Майкл Восс
Рафаэль Асенхо
Джеймс Рейндерс

Параллельное программирование на C++ с помощью библиотеки TBB

Параллельное программирование на C++ с помощью библиотеки TBB

ДМК ПРЕСС

Майкл Восс, Рафаэль Асенхо, Джеймс Рейндерс

Параллельное программирование на C++ с помощью библиотеки TBB

Pro TBB: C++ Parallel Programming with Threading Building Blocks

Michael Voss
Rafael Asenjo
James Reinders

Apress
open

Параллельное программирование на C++ с помощью библиотеки TBB

Майкл Восс
Рафаэль Асенхо
Джеймс Рейндерс

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2020

УДК 004.4
ББК 32.973.202
В76

Восс М., Асенхо Р., Рейндерс Дж.
В76 Параллельное программирование на C++ с помощью библиотеки TBB / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 674 с.: ил.

ISBN 978-5-97060-864-7

Эта книга представляет собой современное руководство для всех пишущих на C++ программистов, которые хотят научиться работать с библиотекой Threading Building Blocks (TBB). Написанная экспертами по TBB и параллельному программированию, она вобрала в себя их многолетний коллективный опыт разработки и преподавания параллельного программирования с помощью TBB. Излагаемый материал представлен в доступной форме. В книге имеются многочисленные примеры и рекомендации, которые помогут вам в полной мере овладеть TBB и задействовать всю мощь параллельных систем.

Книга начинается с описания базовых параллельных алгоритмов и средств распараллеливания, имеющихся в стандартной библиотеке шаблонов C++. Вы узнаете об основах управления памятью, работе со структурами данных и решении типичных проблем синхронизации. Затем эти идеи применяются к более сложным системам, на примере которых объясняются компромиссы во имя производительности, общеупотребительные паттерны параллельного программирования, управление потоками и накладные расходы, а также применение TBB к программированию гетерогенных систем и систем на кристалле.

УДК 004.4
ББК 32.973.202

ISBN 978-1-4842-4397-8 (англ.)
ISBN 978-5-97060-864-7 (рус.)

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторах

Майкл Восс – главный инженер группы архитектуры, графики и программного обеспечения в компании Intel. Он входил в команду разработки TBB еще до выхода версии 1.0 в 2006 году и был первым архитектором API потокового графа TBB. Также является одним из ведущих разработчиков программы Flow Graph Analyzer – графического инструмента для анализа потоков данных на однородных и гетерогенных платформах. Автор и соавтор свыше 40 печатных работ по вопросам параллельного программирования, часто консультирует заказчиков по широкому кругу проблем и помогает им эффективно использовать потоковые библиотеки Intel. До поступления на работу в Intel в 2006 году был доцентом факультета электронной и вычислительной техники имени Эдварда С. Роджерса в Торонтском университете. Получил степень доктора философии в школе электронной и вычислительной техники при университете Пердью в 2001 году.

Рафаэль Асенхо – профессор компьютерной архитектуры в Малагском университете, Испания. Получил степень доктора философии по технике связи в 1997 году и работал доцентом на факультете компьютерной архитектуры с 2001 по 2017 год. В 1996 и 1997 годах был приглашенным преподавателем в Иллинойском университете в Урбана-Шампейне, а в 1998 году – приглашенным научным сотрудником в том же университете. Также работал приглашенным научным сотрудником в Исследовательском центре Томаса Уотсона компании IBM в 2008 году и в компании Cray Inc. в 2011 году. Использует TBB начиная с 2008 года, а в последние пять лет занимается промышленным применением гетерогенных кристаллов, в которых TBB служит в качестве координирующего каркаса. В 2013 и 2014 годах приезжал в Иллинойский университет в Урбана-Шампейне для работы над кристаллами, объединяющими CPU и GPU. В 2015 и 2016 годах начал исследовательскую работу по кристаллам, объединяющим CPU и ППВМ (программируемая пользователем вентильная матрица, англ. FPGA), во время работы в Бристольском университете. Исполнял обязанности председателя на конференции ACM PPoPP'16 (по принципам и практике параллельного программирования) и был членом оргкомитета, а также членом программного комитета на нескольких конференциях по высокопроизводительным вычислениям (PPoPP, SC, PACT, IPDPS, HPCA, EuroPar и SBAC-PAD). В сферу его профессиональных интересов входят модели и архитектуры гетерогенного программирования, распараллеливание нерегулярного кода и оптимизация энергопотребления.

Джеймс Рейндерс – консультант, имеющий за плечами более тридцати лет опыта в области параллельных вычислений. Автор, соавтор или редактор девяти технических книг по параллельному программированию. Принимал участие в разработке ключевых элементов двух самых быстрых в мире компьютеров (номер 1 в списке Top500), а также многих других суперкомпьютеров и средств разработки ПО. В середине 2016 года Джеймс отметил 10 001 день (свыше 27 лет) работы в Intel, но и теперь продолжает писать, преподавать, программировать и консультировать в различных областях, связанных с параллельными вычислениями (высокопроизводительные вычисления и искусственный интеллект).

Благодарности

Два человека поддерживали этот проект с самого начала и до конца – Санджив Шах (Sanjiv Shah) и Херб Хинсторф (Herb Hinstorff). Мы благодарны им за поддержку, ободрение, а временами и ненавязчивое подталкивание.

По-настоящему героические усилия предприняли рецензенты, которые присылали нам содержательные и подробные отзывы на черновики отдельных глав. Благодаря высокому качеству их работы нам пришлось потратить больше времени на редактирование текста, чем первоначально планировалось. Но в результате книга стала лучше.

Круг рецензентов составляла элита пользователей и основных разработчиков TBB. Редко бывает, чтобы в доведении книги до ума принимало участие такое энергичное и благосклонно настроенное сообщество. Читатели книги должны знать этих людей поименно: Эдуapl Айгуаде (Eduard Ayguade), Кристина Бельдика (Cristina Beldica), Константин Бояринов, Хосе Карлос Кабалейро Домингес (José Carlos Cabaleiro Domínguez), Брэд Чемберлен (Brad Chamberlain), Джеймс Джен-Чань Чен (James Jen-Chang Chen), Джим Коуни (Jim Cownie), Сергей Диденко, Алехандро (Алекс) Дюран (Alejandro [Alex] Duran), Михаил Дворский, Рудольф Rudolf (Руди) Эйгенман (Rudolf Eigenmann), Джордж Элькоура (George Elkoura), Андрей Федоров, Алексей Федотов, Томас Фернандес Пена (Tomás Fernández Pena), Элвис Фефей (Elvis Fefey), Евгений Фиксман, Базилио Фрагуэла (Basilio Fraguela), Генри Гэбб (Henry Gabb), Хосе Даниэль Гарсиа Санчес (José Daniel García Sánchez), Мария Хесус Гарзаран (Maria Jesus Garzaran), Александр Гервеши (Alexander Gerveshi), Дарио Суарес Грасиа (Darío Suárez Gracia), Кристина Керманшахче (Kristina Kermanshahche), Янив Клейн (Yaniv Klein), Марк Лубин (Mark Lubin), Антон Малахов, Марк Маклафлин (Mark McLaughlin), Сюзан Мередит (Susan Meredith), Есер Мезиани (Yeser Meziani), Давид Падуа (David Padua), Никита Пономарев, Ануп Мадхусоодханан Прабха (Anoop Madhusoodhanan Prabha), Пабло Ребле (Pablo Reble), Арч Робисон (Arch Robison), Тимми Смит (Timmie Smith), Рубен Гран Техеро (Rubén Gran Tejero), Вазант Товинкере (Vasanth Tovinkere), Сергей Виноградов, Кайл Уилер (Kyle Wheeler) и Флориан Зитцельбергер (Florian Zitzelsberger).

Мы искренне благодарны всем помощникам и приносим извинения тем, кого забыли упомянуть.

Майк (а вместе с ним Рафа и Джеймс!) благодарят всех принимавших участие в работе над TBB на протяжении многих лет: многочисленных разработчиков в Intel, оставивших свой след в библиотеке; Алексея Куканова, который рассказывал нам о ее внутреннем устройстве; сообщество разработчиков программ с открытым исходным кодом; технических писателей и специалистов по маркетингу, которые трудились над документацией и распространением информации о TBB; технических консультантов и прикладных программистов, которые помогают пользователям применять TBB к их задачам; менеджеров, не дававших нам сбиться с пути; а особенно пользователей TBB, которые присылали отзывы о библиотеке и ее функциональности, подсказывавшие нам,

в каком направлении двигаться. А больше всех Майк благодарит свою жену Натали и детей, Ника, Али и Люка, за поддержку и терпение на протяжении вечеров и выходных, проведенных в работе над книгой.

Рафа благодарит своих аспирантов и коллег за советы о том, как понятнее донести идеи TBB: Хосе Карлоса Ромеро (José Carlos Romero), Франсиско Корберу (Francisco Corbera), Алехандро Виллегаса (Alejandro Villegas), Денизу Андреа Константинеску (Denisa Andreea Constantinescu), Анжелес Наварро (Angeles Navarro). Особая благодарность Хосе Даниэлю Гарсиа за увлекательные и информативные беседы по поводу C++11, 14, 17 и 20, а также Алексею Федотову и Пабло Ребле (Pablo Reble) за помощь с примерами применения OpenCL_node, и прежде всего своей жене Анжелес Наварро за поддержку и выполнение некоторых его обязанностей во время работы над книгой.

Джеймс благодарит свою жену Сюзан Мередит – без ее терпеливой и неослабной поддержки книга была бы невозможна. Ко всему прочему детальная правка, когда за красными чернилами иногда не видно было оригинального текста, сделала ее одним из самых ценных наших рецензентов.

Будучи соавторами, мы не находим слов, чтобы воздать должное друг другу. Майк и Джеймс много лет знают друг друга по работе в Intel, и большая удача, что они сошлись в этом проекте. Трудно выразить, как Майк и Джеймс ценят Рафу! Как же повезло его студентам иметь такого энергичного и знающего профессора! Без Рафы читать эту книгу было бы далеко не так приятно. Благодаря знаниям Рафы о TBB книга стала гораздо лучше, а английским он владеет так хорошо, что не раз исправлял ошибки своих англоязычных коллег (Майка и Джеймса). Мы все трое работали над книгой с удовольствием и определенно подстегивали друг друга. Это было прекрасное сотрудничество.

Мы благодарны Тодду Грину (Todd Green), который привел нас в издательство Apress. Спасибо Натали Пао (Natalie Pao) из Apress и Джону Сомоса (John Somoza) из Intel, которые цементировали отношения между Intel и Apress в этом проекте. Мы высоко ценим тяжкий труд всего коллектива Apress над заключением контракта, редактированием и производством.

Спасибо всем!

Майк Восс, Рафаэль Асенхо и Джеймс Рейндерс

Предисловие

Мыслите параллельно

Мы ставили себе целью сделать эту книгу полезной как начинающим, так и искушенным в параллельном программировании. Мы также хотели, чтобы книга была доступна как тем, кто владеет только программированием на C, так и тем, кто уверенно пишет на C++.

Для охвата столь широкой аудитории без «оболванивания» книги мы и написали это предисловие, чтобы уравнять правила игры.

Что такое TBB

TBB – это библиотека для написания параллельных программ на C++, ставшая самым популярным решением и могущая похвастаться отличной поддержкой. Она широко используется – и не без причины. Созданная более десяти лет назад, TBB прошла испытание временем и учитывалась при включении поддержки параллельного программирования в стандарт C++. Хотя C++11 содержит много добавлений, связанных с параллельным программированием, а C++17 и C++2x продвинулись еще дальше в этом направлении, TBB предлагает куда больше, чем стандарт языка. Первая версия TBB была выпущена в 2006 году, поэтому библиотека по-прежнему поддерживает компиляторы, предшествующие выходу C++11. Но мы упростили себе задачу, приняв современный взгляд на TBB и предполагая, что все функции, описанные в C++11, наличествуют. В наши дни дают такой совет: «если у тебя нет компилятора C++11, поставь его». Если сравнивать с книгой о TBB, вышедшей в 2007 году, то, на наш взгляд, C++11, а особенно поддержка лямбда-выражений, расширяют функциональность TBB, а также упрощают ее понимание и использование.

Проще говоря, TBB – лучший способ написать параллельную программу на C++, и мы полагаем, что с TBB ваша продуктивность резко возрастет.

Структура книги и предисловия

В книге четыре основные части.

I. Предисловие. Базовые сведения, полезные для понимания остальной части книги. Содержит обоснование модели параллельного программирования, выбранной в TBB, введение в параллельное программирование, вопросы локальности, кеши, векторные вычисления (набор команд SIMD) и основные средства C++ (сверх имеющихся в языке C), которые поддерживаются или используются TBB.

II. Главы 1–8. Собственно книга о TBB. Включает введение в TBB в объеме, достаточном для эффективного параллельного программирования.
III. Главы 9–20. Включает специальные темы, углубляющие понимание TBB и параллельного программирования. Рассматриваются тонкие нюансы того и другого.
IV. Приложения A и B и глоссарий. Собрание полезных сведений о TBB, которые могут показаться вам интересными, в том числе история (приложение A) и полное справочное руководство (приложение B).

Мыслите параллельно

Для незнакомых с параллельным программированием мы написали это введение, в котором излагаются базовые сведения, делающие книгу более понятной, полезной и независимой. Мы предполагаем только владение языком C на базовом уровне и знакомим с ключевыми элементами C++, которые TBB поддерживает и на которые опирается. Мы рассматриваем параллельное программирование с практической точки зрения, подчеркивая те черты, которые делают параллельные программы более эффективными. Надеемся, что для опытных программистов это предисловие станет полезным напоминанием о теорминологии и способах рассуждений, позволяющих извлекать максимум из параллельного оборудования.

Прочитав предисловие, вы сможете объяснить, что значит «мыслить параллельно» в терминах декомпозиции, масштабирования, корректности, абстрагирования и паттернов. Вы будете понимать, что ключом ко всему параллельному программированию является локальность. Вам раскроется философия программирования на уровне задач, а не на уровне потоков – *революционное достижение концепции параллельного программирования, поддерживаемой TBB*. Вы познакомитесь с теми элементами программирования на C++ сверх известного по программированию на C, которые необходимы для использования TBB.

Предисловие состоит из пяти частей:

1) объяснение мотивов, стоящих за TBB;
2) введение в параллельное программирование;
3) введение в локальность и кеши – аспект оборудования, который, на наш взгляд, неотделим от достижения максимальной производительности средствами параллельного программирования;
4) введение в векторизацию (набор команд SIMD);
5) введение в языковые средства C++ (сверх унаследованных от C), которые используются или поддерживаются библиотекой TBB.

Мотивы, стоящие за библиотекой TBB

Библиотека TBB появилась в 2006 году. Ее создали специалисты по параллельному программированию из компании Intel, и за плечами многих из них были десятки лет работы с моделями параллельного программирования, в т. ч. OpenMP. Многие члены команды TBB потратили годы, чтобы OpenMP могла добиться

впечатляющих успехов, для чего разрабатывали и поддерживали различные реализации OpenMP. Приложение А посвящено истории TBB и ее ключевых концепций, в т. ч. прорывной идее планировщиков с заимствованием задач.

Появившись на заре создания многоядерных процессоров, TBB быстро превратилась в самую популярную у программистов на C++ модель параллельного программирования. На протяжении первого десятилетия после рождения TBB впитывала в себя разнообразные дополнения, сделавшие ее очевидным выбором для параллельного программирования равно у начинающих и опытных пользователей. Будучи проектом с открытым исходным кодом, TBB получала отклики и дополнения со всего мира.

TBB продвигает революционную идею: параллельное программирование должно дать программисту возможность без колебаний выявлять места, подходящие для распараллеливания, а реализация базовой модели программирования (TBB) должна отображать его желания на аппаратные средства во время выполнения.

В основе важности и ценности TBB лежит понимание трех вещей: (1) программирование с применением задач, а не потоков; (2) модели параллельного программирования необязательно должны быть запутанными; (3) как добиться масштабируемости, высокой производительности и переносимости при работе с переносимыми моделями параллельного программирования с низкими накладными расходами, примером которых является TBB. Далее мы рассмотрим все три этих крайне важных аспекта! Можно с уверенностью сказать, что до того, как они стали краеугольными камнями эффективного и структурированного программирования, их важность долгое время недооценивалась.

Программирование с применением задач, а не потоков

Программировать параллельно всегда следует в терминах *задач*, а не *потоков*. В конце этого предисловия мы процитируем авторитетный и глубокий анализ этого положения Эдвардом Ли. В 2006 году он заметил: «Чтобы конкурентное программирование стало обыденностью, необходимо отказаться от потоков как модели программирования».

Параллельное программирование в терминах *потоков* – это упражнение на тему отображения приложения на некоторое число параллельных потоков выполнения на той машине, где выполняется программа. Параллельное программирование в терминах *задач* – это упражнение на тему выявления возможных мест для распараллеливания, после чего исполняющая среда (например, среда TBB) отображает задачи на оборудование во время выполнения, не вынуждая программиста усложнять логику приложения.

Логический *поток* исполняется аппаратным потоком в течение кванта времени, а в будущих квантах времени может быть назначен другому аппаратному потоку. Модель параллельного программирования в терминах потоков терпит провал, потому что часто используется как взаимно однозначное соответствие между логическими потоками и аппаратными потоками (например, процессорными ядрами). Аппаратный поток – это физическое свойство, от машины к машине меняется их количество, равно как и некоторые тонкие характеристики различных реализаций потоков.

Напротив, *задачи* представляют *возможные места* распараллеливания. Разбиение на задачи можно использовать по мере необходимости с учетом количества доступных аппаратных потоков.

Имея в виду эти определения, можно сказать, что программа, написанная в терминах потоков, должна отображать каждый алгоритм на конкретную систему, состоящую из аппаратного и программного обеспечения. Это не только отвлекает внимание, но и порождает целый ряд проблем, из-за которых параллельное программирование оказывается более трудным, менее эффективным и гораздо менее переносимым.

В то же время программа, написанная в терминах задач, допускает наличие механизма времени выполнения, например исполняющей среды TBB, который отображает задачи на реально имеющееся оборудование. Это позволяет не думать о том, каким количеством аппаратных потоков в действительности располагает система. Но важнее то, что на практике это единственный метод, позволяющий эффективно использовать вложенный параллелизм. Это настолько важная возможность, что мы будем возвращаться к ней в нескольких главах.

Компонуемость: параллельное программирование необязательно должно быть запутанным

Библиотека TBB обеспечивает *компонуемость* в параллельном программировании, а это меняет все. Компонуемость означает, что мы можем совместно использовать различные средства TBB без ограничений. А самое главное – она допускает вложенность. В частности, ничто не запрещает поместить один цикл `parallel_for` внутрь другого. Из цикла `parallel_for` можно также вызвать подпрограмму, внутри которой имеется другой цикл `parallel_for`.

Поддержка компонуемого вложенного параллелизма в высшей степени желательна, поскольку открывает больше возможностей для распараллеливания, а это, в свою очередь, позволяет создавать более масштабируемые приложения. Например, система OpenMP не является компонуемой относительно вложенности, т. к. каждый уровень вложенности может приводить к значительным накладным расходам и потреблению ресурсов, что станет причиной истощения ресурсов и аварийного завершения программы. Серьезность этой проблемы становится очевидной при попытке использовать библиотечную подпрограмму, содержащую параллельный код. В TBB подобной проблемы нет, поскольку она поддерживает компонуемость. Отчасти она решается благодаря тому, что TBB позволяет программисту указать места распараллеливания (задачи), а сама во время выполнения решает, как отобразить их на аппаратные средства (потоки).

Это важнейшее преимущество кодирования в терминах задач (доступный, но необязательный параллелизм (см. раздел об «ослабленной последовательной семантике» в главе 2)), а не потоков (принудительный параллелизм). Если бы цикл `parallel_for` считался обязательным, то вложенность привела бы к взрывному росту количества потоков вместе с ворохом проблем неконтролируемого потребления ресурсов, которые легко могут вызвать (и часто вы-

зывают) крах программы. Если же `parallel_for` рассматривается как доступный, но необязательный параллелизм, то исполняющая среда вправе использовать эту информацию при выборе наиболее эффективного отображения на аппаратные средства компьютера.

Мы привыкли ожидать компонуемости от языков программирования, но в большинстве моделей параллельного программирования это свойство утрачено (к счастью, TBB является исключением!). Рассмотрим, к примеру, предложения `if` и `while`. В языках C и C++ они могут сочетаться и вкладываться как угодно. Но представим себе, что это не так – что мы живем в мире, где функция, вызванная из предложения `if`, не может содержать предложения `while`! Сама мысль о таком ограничении кажется нелепой. TBB привносит такого рода компонуемость в *параллельное программирование*, разрешая произвольно сочетать параллельные конструкции без опасения вызвать проблемы.

Масштабируемость, производительность и погоня за переносимой производительностью

Быть может, самым важным преимуществом программирования с использованием библиотеки TBB является то, что она помогает создавать приложения с переносимой производительностью. Мы определяем *переносимую производительность* как характеристику, благодаря которой у программы оказывается похожий «процент пиковой производительности» на различных машинах (с различным оборудованием, разными операционными системами или тем и другим сразу). Мы хотели бы, чтобы высокий процент пиковой производительности имел место на самых разных машинах без необходимости изменять код.

Мы также хотели бы видеть 16-кратный прирост производительности на машине с 64 ядрами по сравнению с четырехъядерной машиной. По различным причинам такое идеальное ускорение на практике почти никогда не наблюдается (но никогда не говори никогда: в некоторых ситуациях благодаря увеличению совокупного размера кеша наблюдается даже более чем идеальное ускорение – это называется *сверхлинейным ускорением*).

Что такое ускорение?

Изначально ускорение определяется как время последовательного выполнения программы, поделенное на время ее параллельного выполнения. Если обычно моя программа работает 3 с, а на четырехъядерном процессоре всего 1 с, то говорят, что ускорение трехкратное. Иногда употребляют термин *эффективность* – это ускорение, поделенное на количество процессорных ядер. Таким образом, трехкратное ускорение эквивалентно эффективности распараллеливания 75 %.

Идеал – 16-кратный прирост производительности при переходе от четырехъядерной машины к 64-ядерной – называется линейной, или идеальной, *масштабируемостью*.

Для его достижения необходимо обеспечить полную занятость всех ядер при увеличении их количества – цель, требующая большого уровня доступного па-

раллелизма. Понятие доступного параллелизма мы более внимательно рассмотрим ниже при обсуждении закона Амдала и следствий из него.

Пока же важно знать, что TBB поддерживает высокопроизводительное программирование и помогает в достижении переносимой производительности. Высокая производительность проистекает из того, что TBB не вносит почти никаких накладных расходов, что позволяет беспрепятственно масштабировать программу. Переносимая производительность позволяет приложению задействовать весь доступный параллелизм, предлагаемый современными компьютерами.

Уверенно делая такие заявления, мы предполагали, что незначительность дополнительных накладных расходов на планирование задач обеспечивает максимальную эффективность выявления и использования возможностей распараллеливания. У этого предположения есть один изъян: если мы напишем программу, идеально соответствующую оборудованию, но без возможности динамической подстройки, то, возможно, сумеем увеличить производительность на несколько процентов. Традиционная модель высокопроизводительных вычислений (High-Performance Computing – HPC), применявшаяся для программирования интенсивных массивно параллельных вычислений на самых больших в мире компьютерах, давно уже обладала такой характеристикой. Разработчик, привыкший к HPC, использующий систему OpenMP со статическим планированием и довольный ее производительностью, вероятно, обнаружит, что TBB со своей динамичностью несколько снижает производительность. Но преимущества такого статического планирования по различным причинам постепенно сходят на нет. По мере усложнения программ на основе модели HPC требуется поддержка вложенного и динамичного параллелизма. Мы видим это во всех аспектах HPC-программирования: появление больших мультифизических моделей, включение искусственного интеллекта (ИИ) и использование методов машинного обучения (МО). Одна из главных причин дополнительной сложности – применение разнообразного оборудования, в результате чего на одной машине появляется гетерогенная вычислительная среда. TBB предлагает средства справиться с этими сложностями, в т. ч. потоковый граф, который мы будем изучать в главе 3.

Очевидно, что для эффективного параллельного программирования необходимо разделять выявление мест распараллеливания в форме задач (функция программиста) и отображение задач на аппаратные потоки (функция реализации модели программирования).

Введение в параллельное программирование

Прежде чем сорвать покров тайны с терминологии и основных понятий параллельного программирования, сделаем смелое заявление: параллельное программирование интуитивно более понятно, чем последовательное. Параллелизм окружает нас в повседневной жизни, делать свои дела шаг за шагом – роскошь, которую мы далеко не всегда можем себе позволить. Параллелизм не является чем-то неизведанным и не должен быть таким в программировании.

Параллелизм вокруг нас

В повседневной жизни мы часто задумываемся о параллелизме. Вот несколько примеров:

- длинные очереди: стоя в длинной очереди, вы наверняка хотели бы, чтобы было несколько очередей покороче (и побыстрее) или чтобы человек, обслуживающий очередь, работал пошустрее. Очереди на кассе в бакалейной лавке, очереди за билетами на электричку, очереди за кофе – все это хорошо знакомые примеры;
- большой объем однообразной работы: когда перед вами стоит большая задача, над которой одновременно могло бы работать много людей, вы, несомненно, хотели бы иметь побольше помощников. Примерами могут служить перевоз пожитков из старой квартиры в новую, раскладывание писем по конвертам для массовой рассылки или установка одной и той же программы на все новые компьютеры в учебном классе. Пословица «Берись дружно, не будет грузно» относится и к компьютерам.

Начав использовать параллелизм, вы станете Мыслить Параллельно. Вы научитесь сначала думать о возможностях распараллеливания в своем проекте и только потом – о кодировании.

> Йель Пат (Yale Pat), знаменитый компьютерный архитектор, как-то заметил:
> Проблема традиционной мудрости заключается в вере, что
> Мыслить параллельно трудно,
> Быть может, потому что мыслить вообще трудно!
> Как нам убедить людей в том, что
> Мыслить параллельно естественно
> (и с этим сложно не согласиться!).

Конкурентность и параллелизм

Стоит отметить, что термины *конкурентный* и *параллельный* связаны, но имеют тонкое отличие. Конкурентный означает «происходящий в течение одного и того же промежутка времени», а параллельный – «происходящий в одно и то же время (по крайней мере, часть времени)». Конкурентность сродни характеру действий человека, пытающегося решать сразу несколько задач, а параллельность – действиям нескольких людей, работающих вместе. На рис. P.1 иллюстрируется различие между конкурентностью и параллелизмом. При создании эффективных параллельных программ нашей целью является не просто достижение конкурентности. Когда говорят о конкурентности, часто не ожидают, что значительная часть работы будет выполняться по-настоящему параллельно – т. е. теоретически два исполнителя необязательно выполнят больше работы, чем один (см. задачи A и B на рис. P.1). Поскольку работа не делается быстрее, конкурентность не уменьшает задержку задачи (время до начала выполнения задачи). Употребление термина *параллельный* подразумевает уменьшение задержки и повышение пропускной способности (объем работы, выполненной в единицу времени). Мы еще вернемся к этому вопросу, когда будем обсуждать пределы параллелизма и очень важный закон Амдала.

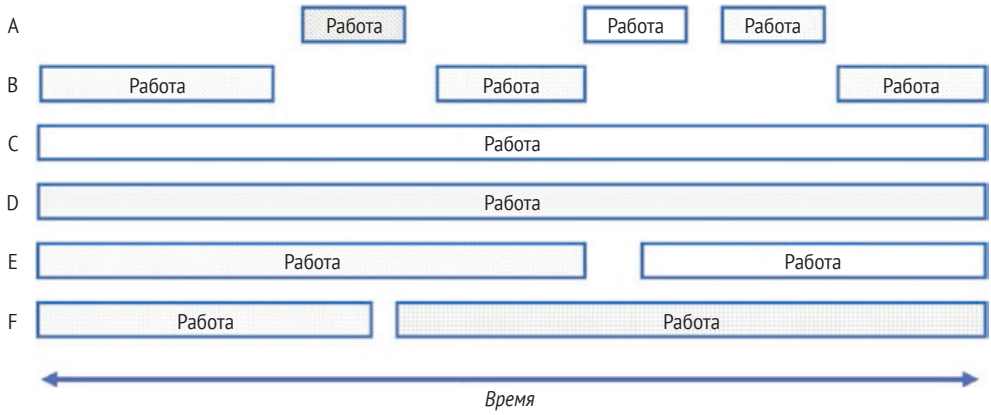


Рис. P.1 ❖ Параллелизм и конкурентность:
задачи (A) и (B) конкурентны друг относительно друга, но не параллельны.
Все остальные комбинации одновременно конкурентны и параллельны

Враги параллелизма: блокировки, разделяемое изменяемое состояние, синхронизация, неумение «мыслить параллельно» и пренебрежение верховенством алгоритма.

Враги параллелизма

Помня о *врагах параллельного программирования*, вы легче воспримете наши аргументы в пользу определенных методов программирования. Вот перечень главных врагов параллельного программирования.

- Блокировки. В параллельном программировании блокировки, или мьютексы, используются, чтобы предоставить потоку монопольный доступ к ресурсу – запретить другим потокам одновременно обращаться к одному и тому же ресурсу. Блокировки – самый распространенный явный способ гарантировать контролируемое (в отличие от хаотического) обновление разделяемых данных несколькими параллельными потоками. Мы ненавидим блокировки, потому что они сериализуют выполнение некоторых участков программы, ограничивая масштабируемость. Это чувство – ненависть к блокировкам – красной нитью проходит через всю книгу. Мы надеемся внедрить эту мантру и в ваши умы, не упуская при этом из виду необходимость надлежащей синхронизации. Поэтому оговоримся: вообще-то мы любим блокировки, когда они необходимы, поскольку без них грядут страшные беды. Эту смесь любви и ненависти к блокировкам необходимо четко осознать.
- Разделяемое изменяемое состояние. Такое состояние возникает всякий раз, как данные сообща используются несколькими потоками, и их разрешено изменять. Такое разделение (или обобществление) либо уменьшает масштабируемость, если необходима синхронизация и она организована правильно, либо приводит к некорректности (состояниям гонки или взаимоблокировкам), если синхронизация (например, блокировка) организована неправильно. Но надо смотреть на жизнь реально – при написании интересных приложений без разделяемого изменяемого со-

стояния не обойтись. Обдумывание аккуратных способов обращения с ним – простой способ понять наше двойственное отношение к блокировкам. В конечном итоге мы научимся «управлять» разделяемым изменяемым состоянием и взаимным исключением (в т. ч. блокировками), так чтобы они работали, как нам нужно.

- Неумение (или нежелание) мыслить параллельно. Хитроумные заплатки и пластыри не смогут компенсировать отсутствие продуманной стратегии использования масштабируемых алгоритмов. Выявлять места, где возможно распараллеливание, следует до начала реализации. Попытка добавить параллелизм в уже написанное приложение чревата рисками. Иногда существующий код сравнительно легко поддается распараллеливанию, но, как правило, необходимо полностью переосмыслить алгоритмы.
- Пренебрежение верховенством алгоритма. Это еще один способ сказать: «Мысли параллельно». От выбора алгоритмов очень сильно зависит масштабируемость приложения. Выбор алгоритма определяет разбиение на задачи, способ доступа к структурам данных и тактику объединения результатов. Оптимальный алгоритм – основа оптимального решения. Оптимальное решение – это сплав подходящего алгоритма, наилучшим образом соответствующего параллельной структуре данных, и наилучшего способа планирования применяемых к данным вычислений. Поиск лучших алгоритмов – нескончаемая задача всех программистов. А параллельные программисты должны включить в определение *лучшего* еще и *масштабируемость*.

Блокировки – и без них никак, и с ними плохо.

Терминология параллелизма

Терминологический словарь параллельного программирования необходимо знать, чтобы общаться с другими программистами. Никаких особенно трудных понятий в нем нет, но очень важно «записать их на подкорку». Любой программист, в т. ч. параллельных приложений (для краткости будем в дальнейшем называть их «параллельными программистами»), тратит годы на овладение своим ремеслом на интуитивном уровне, пусть даже объяснить его основы достаточно просто.

Мы обсудим разложение работы на параллельные задачи, терминологию масштабирования, вопросы корректности и важность локальности, связанную прежде всего с эффектами кеширования.

Как мы находим места для распараллеливания в своем приложении?

На самом верхнем уровне параллелизм существует либо в виде параллельно обрабатываемых данных, либо в виде параллельно исполняемых задач. То и другое *не* является взаимно исключающим. В некотором смысле любой сколько-нибудь важный вид параллелизма – это параллелизм данных. Тем не менее мы вводим оба понятия, поскольку так удобнее рассуждать. При обсуждении масштабируемости и закона Амдала предпочтение, которое мы отдаем *параллелизму данных*, станет более понятным.

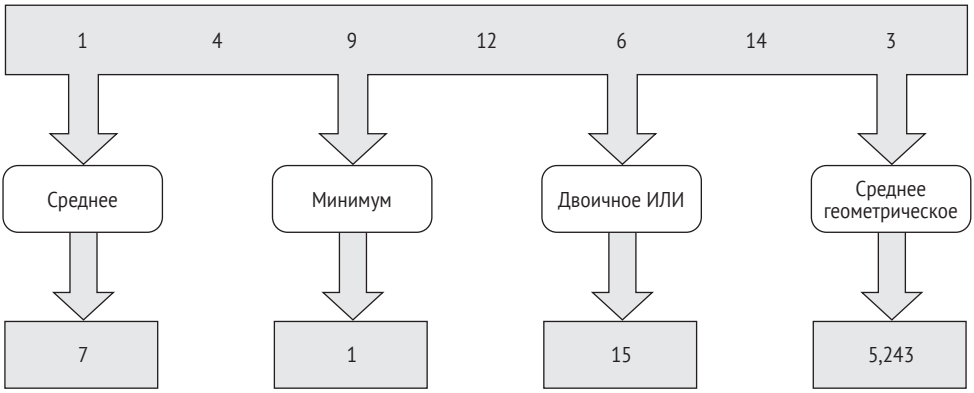


Рис. Р.2 ❖ Параллелизм задач

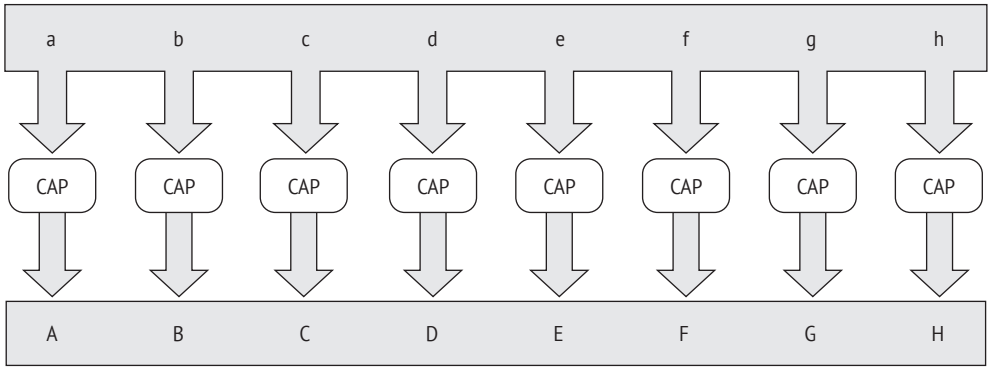


Рис. Р.3 ❖ Параллелизм данных

Терминология: параллелизм задач

Параллелизм задач относится к выполнению различных независимых задач. Это показано на рис. Р.2 на примере математических операций, которые можно применить к одному и тому же набору данных для вычисления независимых значений – в данном случае среднего арифметического, минимума, двоичного ИЛИ и среднего геометрического. При нахождении работ, которые можно распараллелить по задачам, мы ограничены количеством полезных независимых операций.

Выше мы ратовали за *задачи* и против *потоков*. Сейчас, сопоставляя параллелизм *данных* и *задач*, мы оказались в двусмысленной ситуации, потому что в этом контексте слово *задача* употребляется в другом смысле. О каком бы параллелизме ни шла речь – *задач* или *данных*, мы все равно будем программировать в терминах *задач*, а не *потоков*. Так уж устроен словарь параллельных программистов.

Терминология: параллелизм данных

Параллелизм данных (рис. Р.3) изобразить легко: надо взять много данных и применить в каждой порции одно и то же преобразование. На рис. Р.3 каждая

буква преобразуется в верхний регистр. В этом простом примере мы имеем набор данных и операцию, применяемую к каждому его элементу. Программисты, пишущие код для суперкомпьютеров, обожают такого рода задачи, поскольку распараллелить их настолько просто, что они даже получили специальное название – *естественно параллельные*. Совет: если налицо массовый параллелизм данных, не смущайтесь – пользуйтесь и будьте счастливы. Считайте это *счастливым параллелизмом*.

С точки зрения поиска работы, допускающей распараллеливание, подход на основе параллелизма данных ограничен объемом имеющихся данных. А подходы на основе одного лишь параллелизма задач – числом типов задач. Хотя оба вида параллелизма важны и имеют право на существование, для создания истинно масштабируемой параллельной программы критическим аспектом является выявление параллелизма обрабатываемых данных. Масштабируемость означает, что производительность приложения можно повысить путем добавления оборудования (например, дополнительных процессорных ядер) при условии, что данных достаточно. В век больших данных оказывается, что большие данные и параллельное программирование созданы друг для друга. Мы еще вернемся к этому наблюдению при обсуждении закона Амдала.

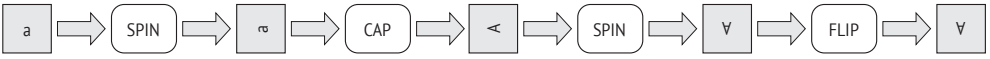


Рис. P.4 ❖ Конвейер

Рис. P.5 ❖ Представьте себе, что каждая позиция – это новая машина на одном из этапов сборки – именно так конвейер воздействует на проходящие через него данные

Терминология: конвейерная обработка

Хотя обнаружить параллелизм задач труднее, чем параллелизм данных, есть один вид такого параллелизма, который стоит отметить особо: *конвейерная обработка*. В этом случае к потоку данных необходимо применить несколько независимых задач. Каждый элемент подвергается обработке на каждом этапе, что обозначено буквой A на рис. P.4. При использовании конвейера поток данных можно обработать быстро, потому что разные элементы могут одновременно находиться на разных этапах, как показано на рис. P.5. В этих примерах время получения окончательного результата (так называемая задержка, измеряемая как время от поступления входных данных до выдачи выходных), возможно, и не уменьшится, но пропускная способность, измеряемая количеством обработанных элементов в единицу времени, возрастет. Благодаря конвейерному параллелизму пропускная способность увеличивается по сравнению с последовательной обработкой. Конвейер может быть устроен и сложнее: он может изменять маршрут прохождения данных или пропускать шаги для некоторых элементов. В TBB имеется специальная поддержка для простых конвейеров (глава 2) и для очень сложных конвейеров (глава 3). Разумеется, на каждом шаге конвейера разрешается использовать параллелизм данных или

задач. Благодаря встроенной в TBB компонуемости с этим не возникает никаких проблем.

Рис. P.6 ❖ Конвейер – у каждого человека своя работа

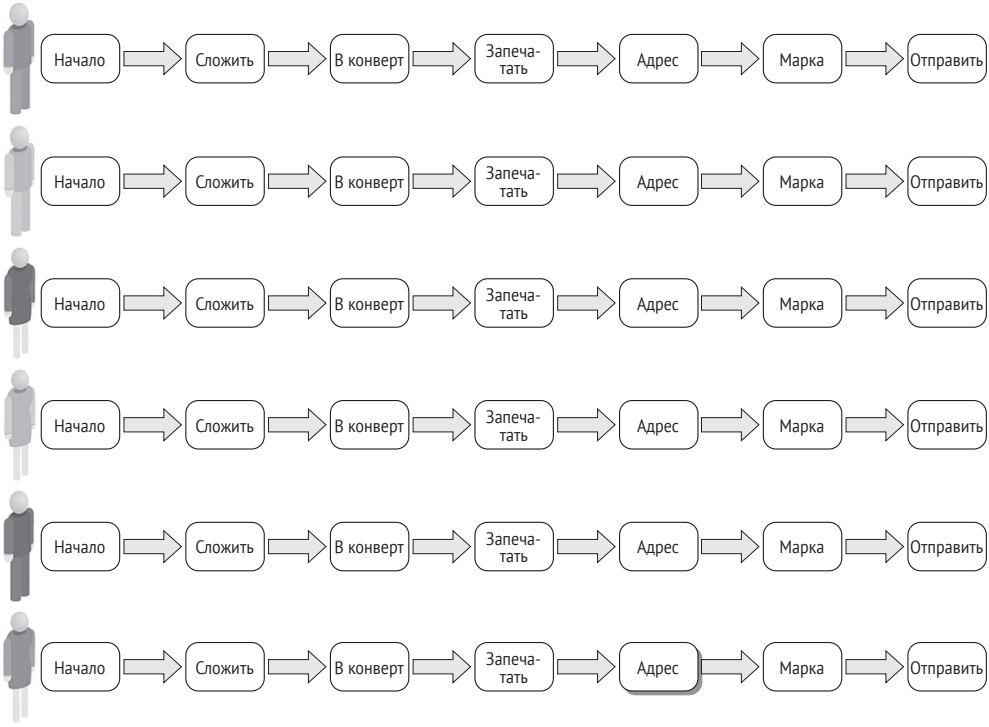


Рис. P.7 ❖ Параллелизм данных – все выполняют одну и ту же работу

Пример использования смешанного параллелизма

Рассмотрим задачу, состоящую из следующих этапов: складывание письма, его помещение в конверт, запечатывание конверта, написание адреса, наклеивание марки и отправка. Если собрать для ее решения шесть человек, то мы можем поручить каждому одну операцию, организовав конвейер (рис. P.6). Это прямая противоположность параллелизму данных, когда мы делим все множество писем на равные части и поручаем каждому человеку одну часть (рис. P.7). После этого каждый человек выполняет все операции с переданными ему материалами.

Ситуацию на рис. Р.7, безусловно, следует предпочесть, если работники находятся далеко друг от друга. Это называется *крупнозернистым* параллелизмом, потому что взаимодействия между задачами происходят редко (работники сходятся вместе, только чтобы получить конверты, а затем расходятся, и каждый занимается своей задачей, включающей и отправку письма). Другой вариант, показанный на рис. Р.6, иллюстрирует *мелкозернистый* параллелизм, когда взаимодействия происходят часто (каждый конверт передается каждому работнику на разных этапах конвейера).

Ни одна из крайностей не отражает реальность точно, хотя иногда аппроксимация может быть достаточно близкой, чтобы оказаться полезной. В нашем примере может случиться, что надписывание конверта занимает так много времени, что для этой операции нужно три человека, тогда как с двумя первыми и двумя последними шагами может справиться один человек. На рис. Р.8 шаги обозначены прямоугольниками, размер которых зависит от трудоемкости работы. Мы можем заключить, что если назначить на каждую операцию ровно одного человека, как на рис. Р.6, то некоторые работники будут простаивать в ожидании работы. Можно сказать, что имеет место скрытая «неполная занятость». Для достижения разумного баланса мы решаем организовать конвейер (рис. Р.9) как гибрид параллелизма данных и задач.

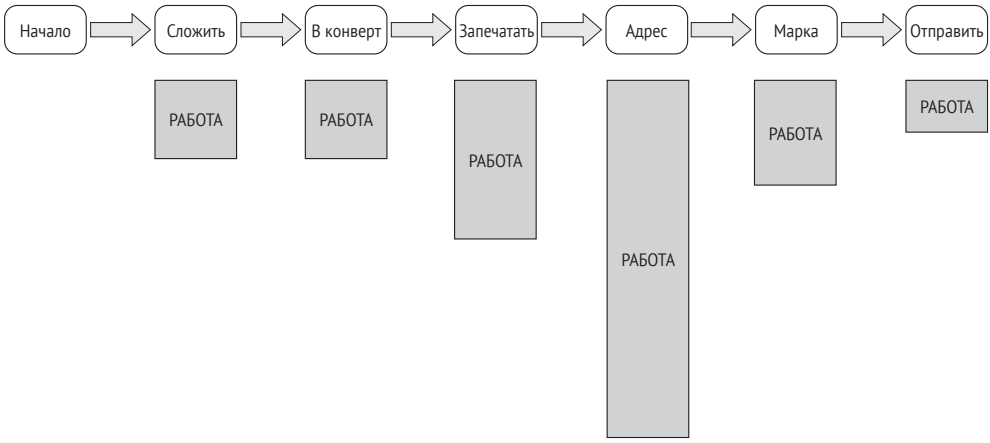


Рис. Р.8 ❖ Неравные по трудоемкости задачи лучше объединить или разделить с учетом наличных трудовых ресурсов

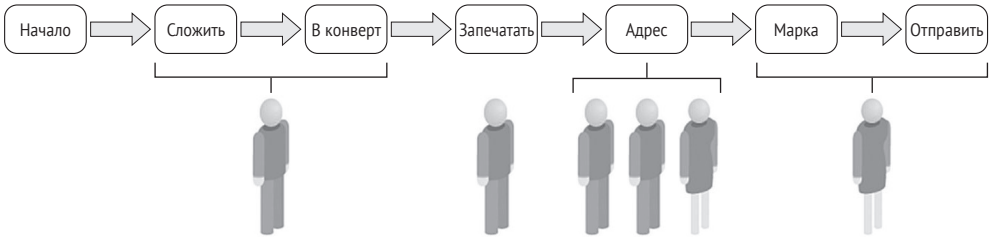


Рис. Р.9 ❖ Поскольку задачи не эквивалентны, направляем больше людей на надписывание конвертов

Достижение параллелизма

Координацию работы людей над подготовкой и отправкой конвертов легко описать следующими двумя концептуальными шагами:

1. Назначить людям задания (и при необходимости перемещать их с одного задания на другое для балансировки загруженности).
2. Вначале назначить на каждое из шести заданий по одному человеку, но быть готовым разбить любую задачу на части, чтобы над ней могло работать несколько людей.

Шесть задач – это складывание письма, помещение его в конверт, запечатывание, надписывание адреса, наклеивание марки и отправка. Для выполнения этой работы в нашем распоряжении имеется шесть человек (ресурсов). Именно так рекомендуется работать с TBB: определить задачи и данные на уровне, допускающем объяснение, а затем разбивать или объединять данные в соответствии с наличными ресурсами.

Первый шаг при написании параллельной программы – определить места для распараллеливания. Во многих учебниках параллелизм задач и данных трактуется так, будто между ними имеется четкое разделение. TBB допускает любую комбинацию.

Мы далеки от осуждения хаоса – мы любим хаос, когда множество некоординируемых задач трудятся над некоторой работой, не утруждаясь взаимной сверкой (синхронизацией). Это так называемое «слабо связанное» параллельное программирование – отличная вещь! Еще больше, чем блокировки, мы ненавидим синхронизацию, т. к. она означает, что задачи должны ждать друг друга. Задачи должны работать, а не простаивать в ожидании!

Если нам повезет, то в программе окажется достаточно параллелизма данных – бери и пользуйся. Чтобы упростить эту работу, TBB требует от нас только одного: определить задачи и способ их разделения. В случае, когда имеется чистый параллелизм данных, мы определяем одну задачу, которой передаем все данные. Затем задача автоматически разделится, чтобы задействовать весь имеющийся аппаратный параллелизм. Неявная синхронизация (в отличие от синхронизации, которую мы явно запрашиваем в коде) зачастую устраняет необходимость в использовании блокировок. Возвращаясь к списку врагов и нашей ненависти к блокировкам, отметим, что неявная синхронизация – это хорошо. Что мы понимаем под «неявной» синхронизацией? Обычно это значит, что синхронизация имеет место, но явно нигде не упоминается в коде. На первый взгляд, это кажется обманом. Ведь синхронизация все-таки произошла – и кто-то должен был ее запросить! В некотором смысле мы рассчитываем на то, что неявные синхронизации планируются и реализуются более тщательно. Чем чаще мы будем использовать стандартные методы TBB и чем реже будем писать собственный код блокировки, тем лучше – в общем случае.

Поручая TBB управлять работой, мы делаем ее ответственной за разбиение работы на части и за синхронизацию. А неявная синхронизация, выполняемая библиотекой, часто устраняет необходимость в написании явного кода синхронизации (см. главу 5).

Мы настоятельно рекомендуем этим и ограничиться и прибегать к явной синхронизации (глава 5) только в случае крайней необходимости или когда это приносит очевидную пользу. По своему опыту мы можем сказать, что даже когда кажется, что нечто подобное необходимо, на деле может оказаться не так. Мы вас предупредили. Но если вы похожи на нас, то иногда игнорируете предупреждения и обжигаетесь. С нами так бывало.

Люди применяют декомпозицию много десятилетий, так что уже выработались определенные паттерны. Мы рассмотрим их позже, когда будем обсуждать паттерны параллельного программирования.

Эффективное параллельное программирование сводится к тому, чтобы обеспечить все задачи полезной работой в каждый момент времени; выявление и исключение холостого простоя – ключ к основной цели: добиться значительного ускорения.

Терминология: масштабируемость и ускорение

Под масштабируемостью программы понимается мера ускорения ее работы при увеличении вычислительной мощности. Ускорением называется отношение времени работы без параллелизма к времени работы распараллеленной программы. Четырехкратное ускорение (4×) означает, что параллельной программе требуется в четыре раза меньше времени. Например, если последовательной программе на однопроцессорной машине нужно 100 с, то той же программе на четырехъядерной машине понадобится 25 с.

Ожидается, что программа, работающая на двух процессорных ядрах, будет быстрее, чем при работе на одном ядре. Аналогично программа, работающая на четырех процессорных ядрах, должна быть быстрее, чем работающая на двух ядрах.

Для любой программы отдача от увеличения степени параллелизма постепенно уменьшается. Нередко производительность не просто выходит на плато, а даже падает, если мы даем программе слишком много вычислительных ресурсов и требуем, чтобы все они использовались. Уровень зернистости, при котором разбиение работы следует прекратить, называется *степенью детализации* (grain size). В TBB это понятие используется, чтобы ограничить разделение данных на части до разумного уровня и тем самым избежать падения производительности. Обычно степень детализации определяется автоматически встроенным в TBB разбивателем, который применяет комбинацию эвристик, чтобы определить начальное значение и затем динамически корректировать его по мере выполнения. Но при желании можно управлять степенью детализации и вручную. Мы не поощряем такой подход в этой книге, поскольку явное задание редко позволяет добиться лучшей производительности по сравнению с автоматическим разбивателем. К тому же оптимальное значение зависит от машины, так что явное задание отрицательно сказывается на переносимости производительности.

Когда параллельно мышление выйдет на интуитивный уровень, структурирование программы с учетом масштабируемости станет для вас обыденным делом.

Сколько параллелизма в приложении?

Вопрос о том, сколько параллелизма можно выдать из приложения, вызывал много споров, а ответ на него зависит от обстоятельств.

Разумеется, это зависит от размера решаемой задачи и от нашей способности найти подходящий алгоритм (и структуры данных), позволяющий задействовать параллелизм. До появления многоядерных процессоров эти споры относились к вопросу о том, как писать эффективные и достойные программы для дорогих и редких параллельных компьютеров. Но с появлением многоядерных процессоров само определение размера, требуемой эффективности и стоимости компьютера изменилось. Мы должны отойти немного назад и оценить, где находимся сейчас. Мир изменился.

Закон Амдала

Известный компьютерный архитектор Джин Амдал (Gene Amdahl) сделал наблюдение, касающееся максимального ожидаемого ускорения компьютерной системы в случае, когда улучшается только часть системы. Это наблюдение, сделанное в 1967 г., получило название закона Амдала. Он утверждает, что если ускорить все части программы в два раза, то можно ожидать, что программа станет работать в два раза быстрее. Но если повысить вдвое производительность только 2/5 программы, то общая производительность возрастет только в 1,25 раза.

Закон Амдала легко проиллюстрировать наглядно. Представьте себе программу, состоящую из пяти равных частей, которая работает 500 с (рис. P.10). Если ускорить две части в 2 и в 4 раза, как показано на рис. P.11, то вместо 500 с мы получим 400 (ускорение 1,25×) и 350 с (ускорение 1,4×) соответственно. Чем дальше, тем больше дают о себе знать ограничения тех частей, которые не ускорились в результате распараллеливания. И сколько бы процессорных ядер ни было, последовательные части образуют барьер в 300 с, преодолеть который невозможно (см. рис. P.12), так что максимальное ускорение составляет всего 1,7×. Если мы можем распараллелить только 2/5 части программы, то никогда не увеличим производительность более чем в 1,7 раза!

Параллельные программисты давно уже используют закон Амдала для предсказания максимального ускорения, которого можно ожидать при использовании нескольких процессоров. По существу, он говорит, что программа никогда не будет работать быстрее суммы частей, работающих последовательно, сколько процессоров ни добавляй.

Многие, опираясь на закон Амдала, предрекали параллельным компьютерам печальную участь, но на вещи можно взглянуть по-другому, и этот взгляд сулит куда более радужные перспективы.

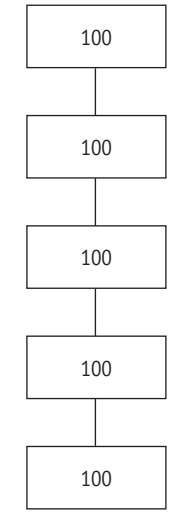


Рис. P.10 ❖ Исходная программа без параллелизма

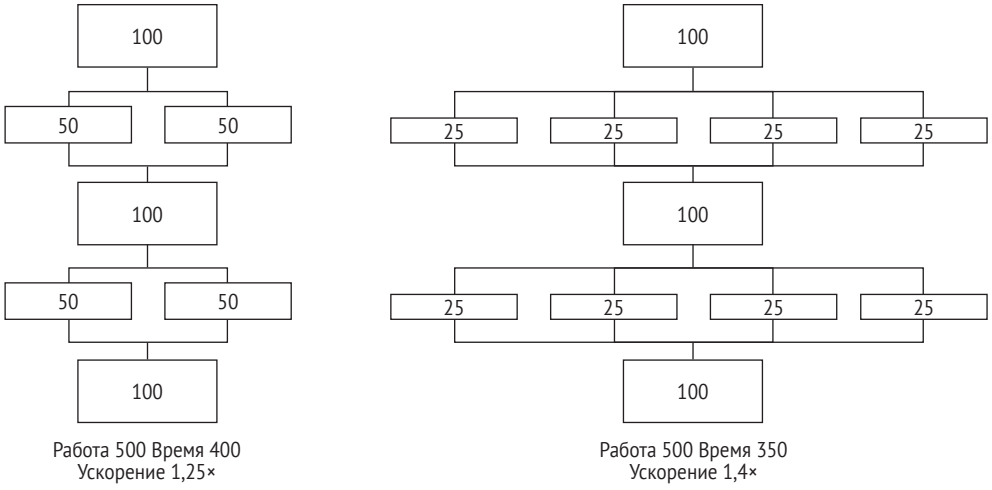


Рис. P.11 ❖ Постепенное увеличение уровня параллелизма

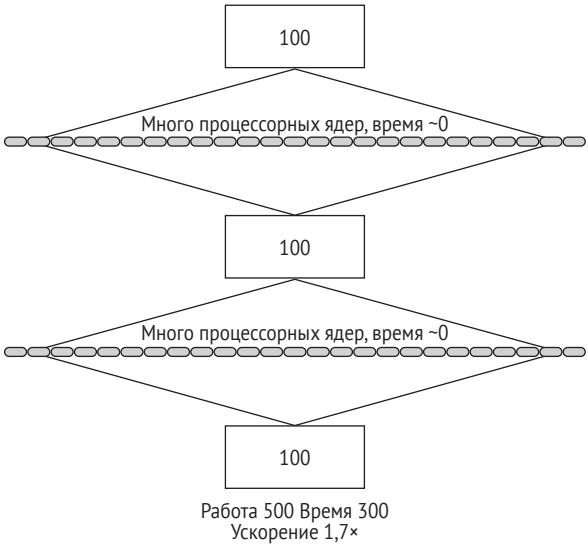


Рис. P.12 ❖ Ограничения, налагаемые законом Амдала

Наблюдения Густафсона, касающиеся закона Амдала

Закон Амдала считает, что программы фиксированы, а изменениям подвергается сам компьютер. Но опыт показывает, что когда компьютеры приобретают новые возможности, приложения изменяются и начинают ими пользоваться. Большинство современных приложений не смогли бы работать на компьютерах десятилетней давности, а многие плохо работали бы уже на машинах, которым всего пять лет. Это относится не только к таким очевидным случаям, как видеоигры, но и к офисным приложениям, веб-браузерам, программам редактирования фотографий и видео.

С появления закона Амдала прошло больше 20 лет, когда Джон Густафсон, работавший в компании Sandia National Labs, взглянул на проблему под другим углом и предложил пересмотреть выводы из закона Амдала. Густафсон заметил, что параллелизм оказывается более полезен, если учитывать рост рабочей нагрузки со временем. Это означает, что по мере того как компьютеры становятся мощнее, мы поручаем им больше работы, т. е. нагрузка не остается неизменной. Во многих случаях с увеличением размера задачи объем работы, возлагаемой на параллельную часть программы, растет быстрее, чем для части, которая не поддается распараллеливанию. Поэтому с увеличением размера задачи доля последовательной части уменьшается, а, стало быть, согласно закону Амдала, масштабируемость растет. Мы можем начать с приложения, как на рис. P.10, но если задача масштабируется вместе с располагаемым уровнем параллелизма, то, вероятно, мы увидим улучшения, представленные на рис. P.13. Если последовательные части занимают столько же времени, сколько раньше, то их процентная доля в общем времени работы программы постепенно снижается. В конечном счете алгоритм достигнет ускорения, показанного на рис. P.14. Производительность растет с такой же скоростью, как число процессоров (n), т. е. мы получаем линейную масштабируемость, что обозначается как $O(n)$.

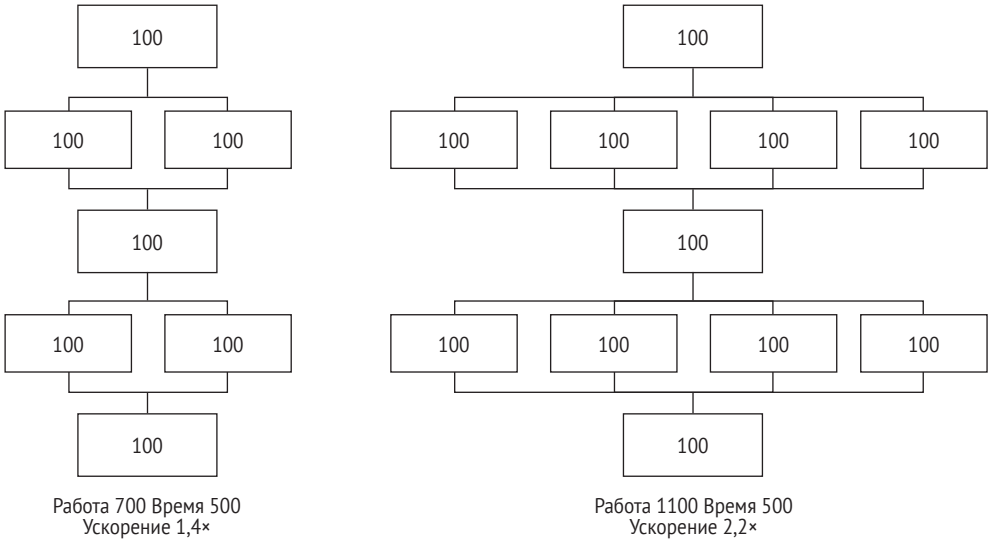


Рис. Р.13 ❖ Масштабирование рабочей нагрузки по мере появления новых возможностей

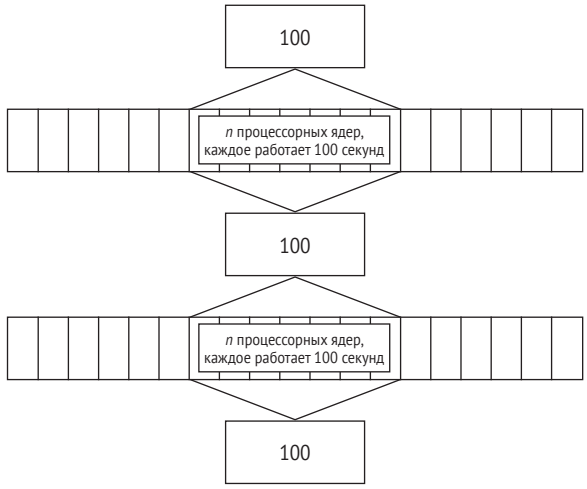


Рис. Р.14 ❖ Густафсон увидел путь к масштабированию

Но все равно в нашем примере эффективность программы ограничена ее последовательными частями. Эффективность использования процессоров составляет примерно 40 % при большом числе процессоров. В суперкомпьютере это может выливаться в нетерпимое растранжирование ресурсов. В системе с многоядерными процессорами можно надеяться, что одновременно с нашим приложением работают другие программы, которые используют не востребованные нами вычислительные мощности. В этом новом мире есть много сложностей. Как бы то ни было, минимизировать объем последовательного кода

хорошо вне зависимости от принятой точки зрения: «стакан наполовину пуст», как в оригинальной трактовке закона Амдала, или «стакан наполовину полон», как с учетом замечания Густафсона.

Правильны обе точки зрения, между ними нет конфликта. Это просто два разных взгляда на одно и то же явление. Закон Амдала предостерегает: попытки сделать существующую программу быстрее на прежней рабочей нагрузке ограничены временем работы последовательной части. Но если мы предвидим большую рабочую нагрузку, то замечание Густафсона дает надежду. История ясно свидетельствует в пользу усложнения программ и решения все более крупных задач, и в этой ситуации параллельное программирование окупается.

Ценность параллелизма легче доказать, если смотреть вперед, а не считать, что мир неизменен.

Заставить современное приложение работать быстрее за счет перехода на параллельный алгоритм, не увеличивая размер задачи, труднее, чем заставить ее работать быстрее на задаче большего размера. Ценность параллелизма проще доказать, если не ограничиваться попыткой ускорить приложение, которое уже хорошо работает на машинах сегодняшнего дня.

Некоторые определяют масштабируемость, нуждающуюся в увеличении размера задачи, как *слабую масштабируемость*. Есть некоторая ирония в том, что термин *естественно параллельный* обычно применяется к масштабируемости другого типа – *сильной масштабируемости*. Поскольку практически всегда масштабируемость имеет место, когда задача масштабируется вместе с уровнем параллелизма, следовало бы говорить просто о *масштабируемости*. Тем не менее принято употреблять термин *естественная*, или *сильная*, *масштабируемость*, говоря о масштабируемости, которая не требует увеличения размера задачи, а масштабируемость, зависящую от роста размера, называть *слабой*. Как и в случае естественного параллелизма, столкнувшись с естественной масштабируемостью, мы радостно пользуемся ей, не испытывая неестественного смущения. Обычно мы ожидаем, что масштабируемость будет *слабой*, но рады любой и просто говорим, что наш алгоритм *масштабируется*.

Масштабируемость приложения сводится к увеличению доли работы, выполняемой параллельно, и уменьшению доли последовательной работы. Закон Амдала побуждает нас сокращать последовательную часть, а наблюдение Густафсона – браться за более крупные задачи.

Что они говорили на самом деле?

Ниже приведены цитаты из знаменитых статей Амдала и Густафсона, которые стали предметом оживленного обсуждения.

> ...Усилия, направленные на достижение высокой скорости параллельной обработки, будут потрачены зря, если не сопровождаются увеличением скорости последовательной обработки примерно во столько же раз.
>
> *— Амдал, 1967*

> …Ускорение следует измерять, масштабируя задачу соответственно количеству процессоров, а не при фиксированном размере задачи.
>
> *— Густафсон, 1988*

Последовательные и параллельные алгоритмы

Одна из истин программирования гласит: лучший последовательный алгоритм редко бывает лучшим параллельным, и наоборот.

Это означает, что написать программу, которая одинаково хорошо работает в системе с одним, двумя и четырьмя процессорными ядрами, труднее, чем хорошую последовательную или хорошую параллельную программу.

Программисты суперкомпьютеров на практике знают, что объем работы является быстро возрастающей функцией от размера задачи. Если объем работы растет быстрее, чем последовательные накладные расходы (например, на передачу данных и синхронизацию), то для исправления плохо масштабируемой программы достаточно увеличить размер задачи. Не так уж необычны случаи, когда программа, плохо масштабируемая на число процессоров свыше 100, начинает прекрасно масштабироваться на 300 и более процессоров, стоит увеличить размер задачи вдвое.

Сильная масштабируемость означает, что ту же самую задачу можно решить быстрее в параллельной системе. Слабая масштабируемость означает, что применение нескольких ядер позволяет решать более интересные задачи за то же время, за какое решались менее интересные задачи на машине с одним ядром.

Чтобы подготовиться к будущему, пишите параллельные программы и отриньте прошлое. Это самый простой и самый хороший совет. Труднее всего писать код, стоя одной ногой в мире эффективной однопоточной производительности, а другой – в мире параллелизма.

Одна из истин программирования гласит: лучший последовательный алгоритм редко бывает лучшим параллельным, и наоборот.

Что такое потоки?

Если вы знаете, что такое поток, то можете смело переходить к разделу «Безопасность в условиях конкурентности». Понятие потока важно хорошо усвоить, пусть даже цель ТВВ – абстрагировать управление потоками. На нижнем уровне мы все равно создаем многопоточную программу и должны понимать, что из этого следует.

Все современные операционные системы многозадачные, и обычно в них используется планирование с вытеснением. Многозадачность означает, что в одно и то же время может быть активно несколько программ. Мы считаем само собой разумеющимся, что почтовая программа и веб-браузер работают одновременно, но еще сравнительно недавно такого не было.

Планирование с вытеснением означает, что операционная система ограничивает время, в течение которого одна программа может занимать ядро

процессора, а по истечении этого времени принудительно передает процессор другой программе. Так операционная система создает иллюзию того, что почтовая программа и браузер работают одновременно, хотя в реальности всю работу выполняет только одно ядро.

Вообще говоря, каждая программа работает более-менее независимо от остальных. В частности, память, в которой размещены переменные нашей программы, полностью отделена от памяти, занятой другими процессами. Почтовая программа не может напрямую присвоить новое значение переменной браузера. Если почтовая программа и умеет взаимодействовать с браузером – например, открывать веб-страницу по ссылке в сообщении, – то реализует это с помощью какого-то механизма коммуникации, который занимает гораздо больше времени, чем доступ к памяти.

Такая изоляция программ друг от друга – весьма ценное свойство, которое является краеугольным камнем современных компьютеров. Внутри одной программы можно организовать несколько потоков выполнения. Операционная система рассматривает программу как *процесс*, а потоки выполнения – как *потоки* (операционной системы).

Все современные операционные системы поддерживают разбиение процессов на несколько потоков выполнения. Потоки, как и процессы, работают независимо, и ни один поток не знает, какие еще потоки работают в той же программе и есть ли они вообще, если только они не осуществляют явную синхронизацию. Главное различие между потоками и процессами заключается в том, что потоки внутри процесса сообща пользуются всеми данными процесса (разделяют данные). Таким образом, для изменения переменной в другом потоке достаточно простого доступа к памяти. Мы называем это «разделяемым изменяемым состоянием» и в этой книге всячески порицаем неприятности, которые сопряжены с разделением. Управление разделением данных – многогранная проблема, которую мы включили в список врагов параллельного программирования. Мы не раз будем возвращаться к этой проблеме и ее решениям.

Заметим, что изменяемое состояние, разделяемое несколькими процессами, – обычное дело. Это может быть память, которую каждый поток явно проецирует в свое адресное пространство, или данные во внешнем хранилище, например в базе данных. Типичный пример – система резервирования авиабилетов, которая позволяет разным пользователям одновременно заказывать билеты, но в конечном итоге все они пользуются одной и той же базой данных и набором мест. Поэтому вы должны понимать, что многие концепции, обсуждаемые нами для одного процесса, встречаются и в более сложных ситуациях. Умение мыслить параллельно дает плоды не только при работе с TBB! Но TBB почти всегда используется внутри одного процесса с несколькими потоками, и лишь немногие потоковые графы (см. главу 3) выходят за рамки одного процесса.

У каждого потока имеется свой *счетчик команд* (регистр, указывающий на команду, исполняемую программой) и *стек* (область памяти, в которой хранятся адреса возврата из подпрограмм и локальные переменные), но в остальном поток разделяет память со всеми другими потоками того же процесса. Даже область стека одного потока доступна другим потокам, хотя при правильном программировании потоки не затирают стеки друг друга.

Тот факт, что потоки одного процесса работают независимо, но сообща используют память, имеет преимущество – потоки могут обмениваться данными быстро, потому что все они обладают равным доступом к памяти процесса. Операционная система может рассматривать несколько потоков как несколько процессов, имеющих одинаковые права доступа к определенным областям памяти. Как уже было сказано, такое «разделяемое изменяемое состояние» является разом и благословением, и проклятием.

Программирование потоков

В начале работы процесс обычно содержит всего один поток выполнения, но может запрашивать запуск дополнительных потоков. Потоки можно использовать для логической декомпозиции программы на множество задач, например пользовательский интерфейс и главная программа. Потоки полезны также для программирования параллелизма, например при наличии многоядерных процессоров.

Стоит начать использовать потоки, как сразу возникает много вопросов. Как следует разбивать программу на потоки и назначать потоки процессорным ядрам, чтобы все ядра были постоянно заняты? Следует ли создавать новый поток всякий раз, как появляется новая задача, или лучше создать пул потоков и управлять им? Должно ли количество потоков зависеть от количества ядер? Что делать с потоком, у которого кончились задачи?

Все эти вопросы важны для реализации многозадачности, но это не значит, что отвечать на них должны мы. Они только отвлекают от выражения целей программы. Точно так же программисты на ассемблере когда-то должны были думать о выравнивании памяти, о размещении данных в памяти, об указателях стека и распределении регистров. Такие языки, как Fortran и C, были созданы, чтобы абстрагировать эти важные детали и оставить их компилятору и библиотекам. Вот и сегодня мы стремимся абстрагировать управление потоками, чтобы программисты могли выражать свои намерения относительно параллелизма прямо.

TBB позволяет программисту выражать намерения относительно параллелизма на более высоком уровне абстракции. При надлежащем использовании код, написанный с применением TBB, неявно параллелен.

Основная идея TBB заключается в том, что мы должны разбить программу на гораздо большее число задач, чем имеется процессоров. Уровень параллелизма следует выбирать из сугубо практических соображений, и пусть исполняющая среда TBB сама решает, какая часть параллелизма реально задействуется.

Что такое SIMD?

Многопоточность – не единственный способ одновременно заниматься несколькими делами! Одна команда может применять одну и ту же операцию к нескольким элементам данных. Такие команды часто называют векторными, а их применение – векторизацией кода (или просто векторизацией). Эта

техника называется «один поток команд, несколько потоков данных» (англ. SIMD). Применение векторных команд для выполнения сразу нескольких вещей – важная тема, которую мы еще обсудим в этом предисловии.

Безопасность в условиях конкурентности

Если код написан таким образом, что при выполнении в условиях конкурентности могут возникнуть проблемы, то говорят, что он не является потокобезопасным. Даже при наличии абстракции, предоставляемой TBB, концепция потокобезопасности важна. Потокобезопасный код написан так, что он правильно работает даже в случае, когда его исполняют несколько потоков. К типичным ошибкам, мешающим достижению потокобезопасности, относятся отсутствие синхронизации доступа при обновлении разделяемых данных (это может повлечь за собой повреждение данных) и неправильное использование синхронизации (может стать причиной взаимоблокировки, которую мы обсудим ниже).

Любая функция, которая сохраняет некоторое состояние между вызовами, должна быть написана с учетом требований потокобезопасности. Правда, это относится только к функциям, которые могут использоваться конкурентно. В общем случае функции, используемые конкурентно, хорошо бы писать так, чтобы у них не было побочных эффектов, тогда конкурентное использование не создаст никаких проблем. Но если глобальные побочные эффекты действительно необходимы, например при изменении единственной на всю программу переменной или при создании файла, следует позаботиться о взаимном исключении, чтобы в каждый момент времени код, содержащий побочные эффекты, мог исполнять только один поток.

В любом коде, где встречается конкурентность или параллелизм, необходимо использовать потокобезопасные библиотеки. Любую библиотеку следует анализировать на предмет потокобезопасности. В библиотеке C++ есть несколько унаследованных от C функций, представляющих проблему, поскольку они сохраняют внутреннее состояние между вызовами, а именно: `asctime`, `ctime`, `gmtime`, `localtime`, `rand` и `strtok`. Прежде чем использовать такие функции, нужно посмотреть в документации, нет ли потокобезопасных вариантов. В стандартной библиотеке шаблонов C++ (STL) контейнерные классы, вообще говоря, не являются потокобезопасными (в главе 6 описаны некоторые имеющиеся в TBB решения этой проблемы, а в главе 5 – использование синхронизации в тех случаях, когда приходится работать с потоконебезопасными функциями).

Взаимное исключение и блокировки

Мы должны все время думать о том, нет ли в нашей программе одновременного доступа к одним и тем же ресурсам. Чаще всего в роли ресурса будут выступать данные, хранящиеся в памяти, но не следует также забывать о файлах и операциях ввода/вывода.

Если точный порядок обновления разделяемых данных имеет значение, то необходима та или иная форма синхронизации. Наилучшая стратегия – выполнить декомпозицию проблемы таким образом, чтобы нужда в синхронизации

возникала нечасто. Для этого следует выделить задачи, которые могут работать независимо, так чтобы единственной синхронизацией было ожидание завершения всех задач в самом конце программы. Синхронизация такого вида называется *барьерной*. Барьеры применяются в случае очень крупнозернистого параллелизма, поскольку они на некоторое время прекращают всякую параллельную работу (программа становится последовательной). Если делать это слишком часто, то, по закону Амдала, мы вообще утратим масштабируемость.

В случае мелкозернистого параллелизма нужна синхронизация доступа к структуре данных, чтобы пока мы пишем в нее, больше никто не мог ни читать, ни изменять. Если обновление памяти производится на основе ранее прочитанного из нее значения, как в случае увеличения счетчика на 10, то следует запретить другим чтение и запись с того момента, как мы начали читать начальное значение, и до того момента, как записано новое значение. Простой способ сделать это показан на рис. P.15. Если мы только читаем, но зато сразу несколько связанных элементов данных, то следует синхронизировать доступ ко всем интересующим нас элементам, чтобы, пока мы читаем, никто не мог их изменить. Эти ограничения, применяемые к другим задачам, называются *взаимным исключением*. Цель взаимного исключения – создать иллюзию *атомарности* (неделимости) набора операций.

В TBB реализованы переносимые механизмы взаимного исключения. Существует два принципиально различных подхода: атомарные операции для очень простых и распространенных действий (например, инкремент) и общий механизм блокировки–разблокировки для более длинных последовательностей команд. Все это обсуждается в главе 5.

Рассмотрим программу с двумя потоками, в начале которой `X = 44`. Поток A выполняет команду `X = X + 10`, а поток B – команду `X = X - 12`. Если добавить блокировку (рис. P.15), так чтобы в каждый момент времени только поток A или поток B мог выполнять команду, то в результате получится `X = 42`. Если оба потока попытаются захватить блокировку одновременно, то одному придется ждать. На рис. P.15 видно, сколько времени будет ждать поток B, запросивший блокировку одновременно с потоком A, но потерпевший неудачу, потому что поток A захватил ее первым.

Вместо блокировок, показанных на рис. P.15, можно было бы использовать небольшой набор операций, атомарность которых гарантирована системой. Мы продемонстрировали блокировки сначала, потому что это общий механизм, позволяющий сделать атомарной произвольную последовательность команд. Такие последовательности следует делать максимально короткими, поскольку, по закону Амдала, они снижают масштабируемость. Если имеется некоторая атомарная операция (например, инкремент), то лучше воспользоваться ей, т. к. это самый быстрый метод, и масштабируемость пострадает меньше.

Какую бы сильную ненависть мы ни испытывали к блокировкам, следует признать, что без них еще хуже. Если отказаться от блокировок в нашем примере, то возникнет *состояние гонки*, в котором возможно по меньшей мере два исхода: `X=32` или `X=54` (рис. P.16). Ниже мы точно определим эту важнейшую концепцию – состояние гонки. Некорректный результат возможен, когда каждый поток сначала читает `X`, затем выполняет вычисление, а потом записывает

новое значение в X. Без блокировки нет гарантии, что один поток не прочитает переменную X после того, как другой изменит ее значение.

Поток A	Поток B	Значение X
LOCK (X)	(*wait*)	44
Read X (44)	(*wait*)	44
add 10	(*wait*)	44
Write X (54)	(*wait*)	54
UNLOCK (X)	(*wait*)	54
	LOCK (X)	54
	Read X (54)	54
	subtract 12	54
	Write X (42)	42
	UNLOCK (X)	**42**

Рис. P.15 ❖ Сериализация, имеющая место при использовании взаимного исключения

Поток A	Поток B	Значение X
Read X (44)		44
add 10	Read X (44)	44
Write X (54)	subtract 12	54
	Write X (32)	**32**

Гонка – сначала A, потом B

Поток A	Поток B	Значение X
	Read X (44)	44
	subtract 12	44
	Write X (32)	32
Read X (32)		32
add 10		32
Write X (42)		**42**

Должно быть

Поток A	Поток B	Значение X
	Read X (44)	44
Read X (44)	subtract 12	44
add 10	Write X (32)	32
Write X (54)		**54**

Гонка – сначала B, потом A

Рис. P.16 ❖ Гонка может приводить к проблемам, если не позаботиться о взаимном исключении. Ошибку легко исправить, заменив каждую последовательность операций чтение/запись подходящей атомарной операцией (атомарным инкрементом или декрементом)

Корректность

Самое трудное, когда учишься мыслить параллельно, – понять, что такое корректность в условиях конкурентности. Конкурентность означает, что имеется несколько потоков управления, которые могут быть активны одновременно. Операционная система вправе планировать эти потоки в разном порядке. При каждом выполнении программы порядок операций может быть различен. Но программист должен гарантировать, что при любом допустимом порядке операций в конкурентной программе получается правильный результат. Высокоуровневая абстракция типа TBB помогает в этом, но есть вещи, о которых мы должны позаботиться сами: потенциальная вариативность результатов в случае параллельного вычисления и новые типы программных ошибок при некорректном использовании блокировок.

Вычисления, выполняемые параллельно, часто дают результаты, отличающиеся от полученных в последовательной программе. Ошибки округления,

возникшие после распараллеливания программы, вызывают удивление у многих программистов. Следует ожидать, что результаты вычислений над числами с плавающей точкой будут отличаться, когда вычисления производятся параллельно, – ведь точность чисел с плавающей точкой ограничена. Например, если выражение `(A+B+C+D)` вычисляется как `((A+B)+(C+D))`, то `A+B` и `C+D` можно вычислять параллельно, но конечная сумма может отличаться от результата вычисления в порядке `(((A+B)+C)+D)`. Результаты параллельного вычисления могут различаться даже при разных прогонах в зависимости от выбранного порядка вычислений. Такое недетерминированное поведение часто можно контролировать, уменьшая гибкость на этапе выполнения. Мы расскажем о соответствующих возможностях, в частности о режиме детерминированной редукции (глава 16). Недетерминированность может сильно осложнить отладку и тестирование, поэтому зачастую желательно принудительно обеспечить детерминированное поведение. В зависимости от обстоятельств это может снизить производительность, т. к. по существу сводится к дополнительной синхронизации.

Некоторые виды ошибок возможны только в конкурентной программе, поскольку касаются координации задач. Это *взаимоблокировки* и *состояние гонки*. Недетерминированность также представляет проблему, потому что в конкурентной программе может существовать много путей выполнения, т. к. имеется много независимо работающих задач.

Хотя TBB упрощает программирование, сводя шансы подобных ошибок к минимуму, они все-таки могут встретиться. Многопоточная программа может оказаться недетерминированной в результате гонки, т. е. одна и та же программа с одними и теми же входными данными может при каждом запуске выбирать различные пути выполнения. Если такое случается, то ошибка будет невоспроизводимой, а вмешательство отладчика легко может изменить ситуацию, так что отладка, мягко говоря, бесит.

Поиск и устранение причин нежелательной недетерминированности – дело непростое. Специальные инструменты типа Intel Advisor могут помочь, но первый шаг – понять, как возникают такие проблемы, и стараться избегать их.

Еще одна широко распространенная проблема при переходе от последовательного кода к параллельному, тоже являющаяся следствием недетерминированности, – нестабильность результатов. Это означает, что мы получаем разные результаты вследствие не всегда очевидного изменения порядка выполнения работы. Некоторые алгоритмы могут демонстрировать нестабильность, тогда как другие просто пользуются возможностью изменять порядок операций в предположении, что все возможные порядки правильны.

Далее мы объясним, в чем суть трех основных ошибок при параллельном программировании, и покажем способы решения.

Взаимоблокировка

Взаимоблокировка имеет место, когда по крайней мере две задачи ждут друг друга и ни одна не может продолжить выполнение. Это легко может случиться, если программа захватывает несколько блокировок. Если задаче `A` нужны блокировки `R` и `X`, то она может захватить `R`, а затем попробовать захватить `X`. Тем

временем задача B, которой нужны те же самые блокировки, сначала захватывает X, а затем пытается захватить R. В результате задача A ждет освобождения X, удерживая R, а задача B ждет R, удерживая X. Возникшую тупиковую ситуацию можно разрешить, только если одна из задач освободит удерживаемую ей блокировку. Если ни одна задача не уступит, то будет иметь место взаимоблокировка, и задачи зависнут навсегда.

Разрешение проблемы взаимоблокировки

Чтобы обойтись без блокировок, используйте неявную синхронизацию. Вообще говоря, следует избегать использования блокировок, особенно захвата нескольких подряд. Захват блокировки с последующим вызовом функции или подпрограммы, в которой тоже используются блокировки, – частый источник взаимоблокировок. Поскольку доступ к разделяемым ресурсам иногда неизбежен, предлагается два общих решения: всегда захватывать блокировки в одном и том же порядке (скажем, сначала A, потом B) или освободить все блокировки, если какую-нибудь захватить не удается, и начать все сначала (после задержки на случайное время).

Состояния гонки

Состояние гонки возникает, когда несколько задач читают и записывают одну и ту же область памяти без надлежащей синхронизации. Иногда гонка заканчивается корректно, а иногда некорректно. На рис. P.16 приведен простой пример, в котором из-за гонки возможно три разных исхода.

Гонки не приводят к таким катастрофическим последствиям, как взаимоблокировки, но они более коварны, поскольку ошибка неочевидна, но может стать причиной повреждения данных (когда читается или записывается неправильное значение). Следствием некоторых гонок может стать неожиданное (и нежелательное) состояние, поскольку в обновлении состояния, включающего более одного элемента, поучаствовало несколько потоков.

Решение проблемы гонок

Управляйте разделяемыми данными дисциплинированно, применяя механизмы синхронизации, описанные в главе 5. Избегайте низкоуровневых методов на основе блокировок, поскольку это легкий способ наделать ошибок. Явные блокировки следует оставить на самый крайний случай. Вообще говоря, лучше при любой возможности использовать синхронизацию, неявно подразумеваемую шаблонами алгоритмов и планировщиком задач. Например, пользуйтесь алгоритмом `parallel_reduce` (глава 2), а не изобретайте свой собственный с разделяемыми переменными. Реализация `parallel_reduce` гарантирует, что операция объединения не начнется, пока не будут завершены подзадачи, результаты которых подлежат объединению.

Нестабильность (недетерминированность) результатов

Параллельная программа обычно вычисляет ответы по-разному при каждом запуске из-за мелких различий в порядке выполнения большого числа конку-

рентных задач, особенно если она запускается на машинах с разным числом процессоров. Мы уже говорили об этом при обсуждении корректности.

Решение проблемы нестабильности результатов

TBB предлагает способ повысить детерминированность за счет уменьшения гибкости на этапе выполнения. Правда, при этом может несколько снизиться производительность, но зачастую это разумная плата за детерминированность. Подробнее данная тема обсуждается в главе 16.

Уровень абстракции

При написании программы важно выбрать подходящий уровень абстракции. В наши дни мало программистов, пишущих на ассемблере. Такие языки программирования, как C и C++, абстрагируют низкоуровневые детали. И вряд ли кто-то скучает по старым временам.

С параллелизмом то же самое. Легко поймать себя на том, что пишешь код слишком низкого уровня. При программировании на уровне потоков мы сами должны управлять потоками, что отнимает время и чревато ошибками.

TBB открывает возможность уйти от управления потоками. Такой код проще создавать и сопровождать, к тому же он получается более элегантным. На практике он оказывается еще и более переносимым, в т. ч. с точки зрения производительности. Однако мы должны продумывать алгоритмы в плане разделения работы и данных между задачами.

Паттерны

Опытные параллельные программисты знают о существовании типичных проблем, для которых имеются типичные же решения. Да и все программирование устроено так же – в результате в нашем словаре существуют такие понятия, как стек, очередь и связные списки. Параллельное программирование привносит понятия отображения, редукции и конвейера.

Мы называем их *паттернами* и стараемся приспособить к своим нуждам. Изучение типичных паттернов – лучший способ учиться у тех, кто шел впереди нас. В TBB реализованы основные паттерны, поэтому мы можем неявно изучать их, изучая TBB. Мы считаем паттерны настолько важной концепцией, что посвятим их обсуждению главу 8.

Локальность и месть кешей

Для эффективного параллельного программирования необходимо отчетливо осознавать важность *локальности*. Для этого нам придется кратко поговорить об оборудовании и, в частности, о *кешах* памяти. Кеш – это просто аппаратный буфер, где хранятся данные, которые недавно читались или модифицировались. Кеш расположен ближе к процессору, так что доступ к нему производится быстрее, чем к основной памяти. Цель кеша – ускорить работу, поэтому программа, которая эффективно использует кеши, работает быстрее.

Мы говорим «кеши», а не «кеш», потому что в современных компьютерах обычно используется несколько уровней аппаратного кеширования. Но нам, как правило, достаточно представлять себе кеш как единый набор данных. Углубляться в устройство кешей необязательно, но на верхнем уровне знать о них нужно, чтобы разобраться в локальности и таких смежных вопросах, как *разделение изменяемого состояния*, а особенно в коварном феномене *ложного разделения данных*.

Важные для нас следствия концепции кеша: локальность, разделение и ложное разделение. Чтобы разобраться в них, нужно понимать, что такое кеши и строки кешей. Это фундаментальные компоненты конструкции любого современного компьютера.

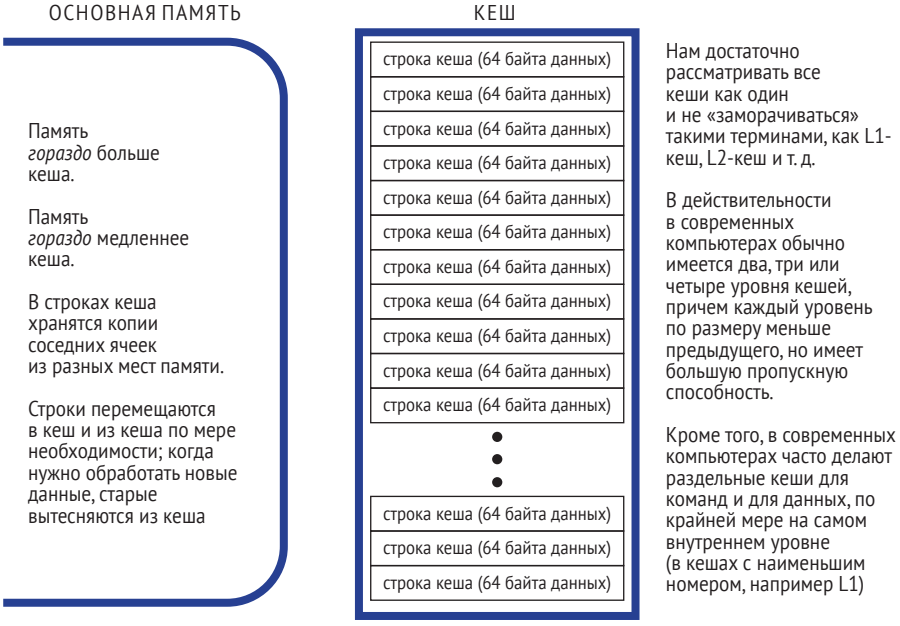


Рис. P.17 ❖ Основная память и кеш

Аппаратное обоснование

Мы хотели бы по возможности игнорировать детали аппаратной реализации, поскольку чем больше мы принимаем во внимание особенности конкретной системы, тем больше утрачиваем переносимость, и переносимость производительности в том числе. Но есть одно заметное и важное исключение: кеши (рис. P.17).

Кеш памяти имеется во всех современных компьютерах. И в большинстве систем есть несколько уровней кеша. Так было не всегда; первоначально компьютеры выбирали данные и команды из памяти только по мере необходимости и записывали результаты непосредственно в память. Простые были времена!

Но быстродействие процессоров росло гораздо быстрее, чем памяти. Сделать всю память такой же быстрой, как процессор, непомерно дорого. Вместо этого конструкторы пошли на компромисс: сравнимой по быстродействию с процессором делается только небольшая часть памяти – *кеши*. Основная память может оставаться более медленной и, стало быть, более дешевой. Оборудование знает, как перемещать данные в кеш и из кеша, в результате чего увеличивается количество «пунктов обработки» на пути между памятью и процессорными ядрами. Кеши критически важны для преодоления несоответствия между скоростями памяти и процессора.

Практически во всех компьютерах кеши используются только для временного хранения копий данных, которые в конечном итоге должны оказаться в памяти. Поэтому функция подсистемы памяти состоит в том, чтобы переместить входные данные в кеш, расположенный ближе к обрабатывающему ядру, а результаты обработки переместить в основную память. По мере того как данные читаются из памяти в кеш, какие-то данные необходимо вытеснять из кеша, чтобы освободить место для недавно запрошенных. Конструкторы кешей стремятся сделать так, чтобы вытеснялись в основном данные, используемые реже всего. Идея в том, что если данные давно не использовались, то, наверное, не будут использоваться и в ближайшем будущем. Таким образом, ценное место в кешах приберегается для данных, которые с большой вероятностью будут использованы снова.

Коль скоро процессор обратился к данным, лучше уж использовать их по максимуму, пока они находятся в кеше. Если данные продолжают использоваться, то они удерживаются в кеше. Напротив, долгое неиспользование приведет к вытеснению данных из кеша, и, когда они вновь понадобятся, придется производить дорогостоящий (более медленный) доступ к памяти, чтобы снова поместить данные в кеш. Кроме того, всякий раз, как запускается новый поток, данные, скорее всего, будут удалены из кеша, чтобы освободить место для данных этого потока.

Локальность ссылок

Итак, первая выборка данных из памяти обходится дорого, но их использование на протяжении некоторого времени после выборки гораздо дешевле. Это потому, что информация сохраняется в кешах – здесь можно провести аналогию с нашей краткосрочной памятью, благодаря которой события, произошедшие сегодня, вспомнить гораздо легче, чем те, что случились несколько месяцев назад.

Простой хрестоматийный пример – умножение матриц, `C=AxB`, где `A`, `B` и `C` – квадратные матрицы размера `nxn` (см. рис. Р.18).

```
for (i=0;i<n;i++)
  for(j=0;j<n;j++)
    for (k=0;k<n;k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Рис. Р.18 ❖ Умножение матриц с плохой локальностью ссылок

В C и C++ массивы хранятся *по строкам*. Это означает, что позиции соседних элементов отличаются последним индексом. Иначе говоря, элементы `c[i][2]` и `c[i][3]` расположены в памяти рядом, а `c[2][j]` и `c[3][j]` далеко друг от друга (в нашем примере между ними `n` элементов).

Поменяв местами итерации по индексам `j` и `k`, как показано на рис. P.19, мы значительно ускорим программу за счет увеличения локальности ссылок. Результат при этом не изменяется, но производительность возрастает благодаря более эффективному использованию кешей. В нашем примере значение *n* должно быть достаточно велико, чтобы суммарный размер матриц превосходил размер кешей. Если это не так, то порядок не играет роли, поскольку матрицы в любом случае поместятся в кеши целиком. При `n=10000` в каждой матрице будет сто миллионов элементов. В предположении, что матрицы содержат числа с плавающей точкой двойной точности, общий размер трех матриц составит примерно 2,4 ГБ. При таком объеме эффекты кеширования начнут проявляться на любой машине, существовавшей в мире на момент публикации книги! Почти все компьютеры выигрывают от изменения порядка индексов, но и почти все не заметят никакой разницы, если `n` настолько мало, что все три матрицы помещаются в кеш.

```
for (i=0;i<n;i++)
  for(k=0;k<n;k++)
    for (j=0;j<n;j++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Рис. P.19 ❖ Умножение матриц с улучшенной локальностью ссылок

Строки кеша, выравнивание, разделение, взаимное исключение и ложное разделение

Кеши организованы в виде набора строк. И процессоры перемещают данные между памятью и кешем порциями, кратными размеру строки кеша. В результате появляются три понятия, нуждающихся в пояснении: выравнивание данных, разделение данных и ложное разделение.

Длина строки кеша вообще-то произвольна, но в настоящее время она чаще всего равна 512 бит, т. е. 64 байта, или восемь чисел с плавающей точкой двойной точности, или шестнадцать 32-разрядных целых.

Выравнивание

Для любого элемента данных (например, типа `int`, `long`, `double` или `short`) лучше, если он помещается в одной строке кеша. Взгляните на рис. P.17 или P.20 и представьте, что было бы, если бы один элемент данных (например, типа `double`) пересекал границу двух строк. В таком случае нужно было бы обратиться (для чтения или записи) к двум строкам кеша вместо одной. Вообще говоря, это займет в два раза больше времени. Выравнивание элементов данных, так чтобы они не пересекали границы строк кеша, очень важно с точки зрения производительности. Отдавая должное конструкторам оборудования, скажем,

что некоторые системы заметно снижают потери из-за невыровненных данных. Но поскольку мы не можем рассчитывать на наличие такой поддержки, настоятельно рекомендуем выравнивать данные на естественные границы в памяти. Массивы обычно занимают несколько строк кеша, если только они не очень малы; как правило, мы рекомендуем выравнивать массив на ту же границу, что и его элемент, так чтобы один элемент массива не оказался сразу в двух строках кеша. Тот же совет относится и к структурам, хотя иногда имеет смысл выравнивать небольшие структуры, так чтобы они целиком помещались в строку кеша.

Недостаток выравнивания – бесполезно расходуемое пространство. При каждом выравнивании данных часть памяти может резервироваться впустую. Вообще говоря, на это можно не обращать внимания, потому что память стоит дешево. Если выравнивание происходит часто, от него можно отказаться или изменить размещение данных в памяти для ее экономии – иногда это все-таки важно. Но выравнивание очень существенно для производительности, поэтому лучше его оставить. Компиляторы автоматически выравнивают переменные, в т. ч. массивы и структуры. При выделении памяти самостоятельно (например, функцией `malloc`) выравнивать приходится тоже самостоятельно; как это сделать, мы покажем в главе 7.

Но настоящая причина, по которой мы упомянули о выравнивании, – разделение данных и опасности ложного разделения.

Разделение данных

Организовать разделение копий неизменяемых данных в памяти легко, потому что любая копия действительна. Разделение неизменяемых данных не влечет никаких особых проблем в параллельном программировании.

Проблемы при параллельной обработке создают изменяемые данные. Мы уже упоминали *разделяемое изменяемое состояние* среди врагов параллелизма! В общем случае следует сводить к минимуму разделение изменяемых данных между задачами. Чем меньше разделения, тем меньше шансов, что возникнет ошибка и понадобится отладка. Но мы знаем, что именно разделение данных позволяет параллельным задачам работать над одной и той же проблемой во имя масштабируемости, поэтому придется обсудить, как делать это корректно.

Разделяемое изменяемое состояние порождает две проблемы: (1) порядок операций и (2) ложное разделение. Первая внутренне присуща параллельному программированию и никак не связана с оборудованием. Мы уже обсудили взаимное исключение и проиллюстрировали, в чем состоит проблема, на рис. P.16. Это важнейшая тема, которую должен хорошо понимать любой параллельный программист.

Ложное разделение

Поскольку данные перемещаются блоками, равными по длине строке кеша, может случиться, что несколько совершенно независимых переменных или динамически выделенных областей памяти полностью или частично окажутся в одной и той же строке кеша (рис. P.20). Такое разделение строки кеша не приводит к отказу программы, но может заметно снизить производительность.

Исчерпывающее объяснение проблем, возникающих при разделении строки кеша изменяемыми данными, которые используются в разных задачах, заняло бы много страниц. А по-простому можно сказать, что обновление данных в любом месте строки кеша может замедлять доступ ко всей строке из других задач.

Но каковы бы ни были причины, по которым ложное разделение замедляет работу компьютера, нужно знать, что в хорошо написанных параллельных программах принимаются меры, чтобы избежать ложного разделения. Даже если для какой-то одной машины последствия ложного разделения не так серьезны, как для других, все равно для достижения переносимой производительности такие меры следует принимать.

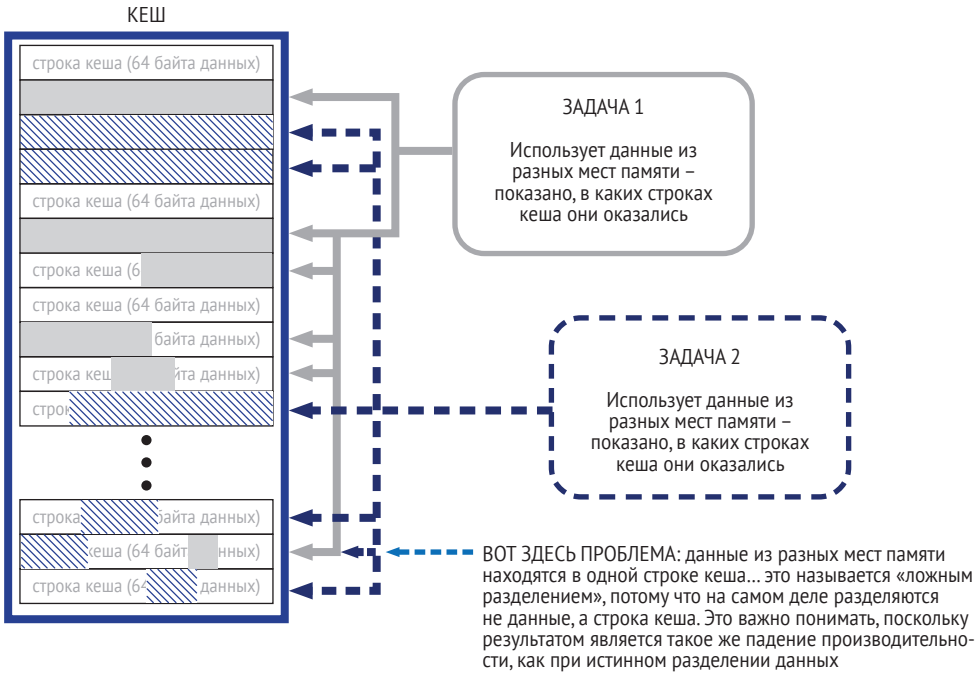


Рис. P.20 ❖ Ложное разделение возникает, когда данные двух разных задач оказываются в одной строке кеша

Чтобы показать, почему **ложное разделение** так вредно для производительности, рассмотрим дополнительные накладные расходы, которые несут кеши и операционная система, когда два потока обращаются к близким участкам памяти. Для определенности будем предполагать, что строка кеша содержит 64 байта, что на процессорах, разделяющих кеш, работает по меньшей мере два потока и что в нашей программе определен массив, к которому каждый поток обращается по индексу, совпадающему с его идентификатором:

```
int my_private_counter[MAX_THREADS];
```

Два соседних элемента массива `my_private_counter`, скорее всего, окажутся в одной строке кеша. Поэтому наша программа будет нести накладные рас-

ходы именно из-за ложного разделения, вызванного тем, что данные, используемые разными потоками, оказались в одной строке кеша. Рассмотрим два потока 0 и 1, исполняемых ядрами 0 и 1 соответственно, и описанную ниже последовательность событий.

Поток 0 увеличивает `my_private_counter[0]`, т. е. выполняет следующие операции: прочитать значение в частный кеш ядра 0, увеличить счетчик и записать результат. Точнее, ядро 0 читает в кеш всю строку (64 байта), включающую этот счетчик, а затем записывает в счетчик новое значение (обычно только в строке кеша).

Если затем поток 1 увеличит `my_private_counter[1]`, то придется заплатить за ложное разделение. Весьма вероятно, что элементы 0 и 1 массива `my_private_counter` попадут в одну строку кеша. В таком случае, когда поток 1 на ядре 1 попытается прочитать свой счетчик, в игру вступит протокол когерентности кешей. Этот протокол работает на уровне строки кеша и осмотрительно предполагает, что поток 1 читает значение, записанное потоком 0 (как если бы потоки 0 и 1 действительно разделяли общий счетчик). Поэтому ядро 1 должно прочитать строку кеша ядра 0 (самый медленный вариант – сбросить всю строку кеша ядра 0 в память, а затем прочитать оттуда в кеш ядра 1). Это само по себе дорого, но становится еще хуже, когда поток 1 увеличивает свой счетчик, делая недействительным копию строки кеша ядра 0.

Если теперь поток 0 снова увеличит `my_private_counter[0]`, то не найдет своего счетчика в локальном кеше ядра 0, поскольку он стал недействительным. Придется заплатить еще за доступ к недавно обновленной версии этой строки в кеше ядра 1. И снова, если затем поток 0 увеличит свой счетчик, копия в кеше ядра 1 станет недействительной. Если такое поведение будет продолжаться, то скорость доступа потоков 0 и 1 к своим счетчикам существенно снизится по сравнению с ситуацией, когда каждый счетчик хранится в отдельной строке кеша.

Это явление называется «ложным разделением», потому что на самом деле у каждого потока имеется свой частный (неразделяемый) счетчик, но из-за протокола когерентности кешей, работающего на уровне строки кеша, и того факта, что оба счетчика «разделяют» одну и ту же строку кеша, создается впечатление, что, с точки зрения оборудования, они разделяются.

Вероятно, у вас возникла мысль, что есть очевидное решение – изменить протокол когерентности кешей, так чтобы он работал на уровне слов, а не строк. Однако эта альтернатива аппаратно неприемлема, так что конструкторы оборудования просят нас решать проблему программно: делайте так, чтобы частные счетчики попадали в разные строки кеша, тогда оборудованию будет ясно, что подключать накладные механизмы обеспечения когерентности кешей не нужно.

Мы видели, что ложное разделение может повлечь гигантские накладные расходы. В нашем простом примере правильным решением было бы использовать копии элементов. Сделать это можно по-разному, например использовать распределитель с выравниванием на границу строки кеша для выделения памяти под каждый элемент данных, нужный потоку, а не выделять всю такую память одним блоком, рискуя получить ложное разделение.

Предотвращение ложного разделения с помощью выравнивания

Чтобы избежать ложного разделения (рис. P.20), нужно следить за тем, чтобы *заведомо различные элементы* изменяемых данных, которые могут обновляться параллельно, не оказались в одной и той же строке кеша. Для этого применяется сочетание выравнивания и дополнения.

Выравнивание таких объектов, как массивы и структуры, на границу строки кеша предотвращает попадание начала объекта в строку кеша, используемую кем-то еще. Обычно для этого применяется версия `malloc`, выравнивающая выделенную область памяти на границу строки кеша. Мы будем обсуждать выделение памяти, в т. ч. распределители с выравниванием на границу строки кеша (например, `tbb::cache_aligned_allocator`), в главе 7. Можно также выравнивать статические и локальные переменные с помощью директив компилятора (прагм), но необходимость в этом возникает гораздо реже. Следует отметить, что зачастую падение производительности, вызванное ложным разделением, носит недетерминированный характер, т. е. может проявляться при одном выполнении приложения и не проявляться при другом. Это может свести с ума, поскольку отладка недетерминированной проблемы – всегда адский труд, т. к. нельзя рассчитывать, что при конкретном запуске программы проблема повторится!

TBB помнит о кешах

При проектировании TBB наличие кешей учитывалось, поэтому библиотека стремится ограничить излишние перемещения задач и данных. Если задачу необходимо переместить на другой процессор, то TBB выбирает задачу, для которой вероятность нахождения данных в кеше старого процессора минимальна. Эти соображения встроены в планировщики с заимствованием работ и потому являются частью шаблонов алгоритмов, обсуждаемых в главах 1–3.

Тема кешей встречается во многих главах, но прежде всего следует отметить:

- главу 5, в которой обсуждается приватизация и редукция;
- главу 7, где приводятся важные замечания о распределителях памяти с поддержкой кешей, в т. ч. выравнивания и дополнения для предотвращения ложного разделения;
- главу 16, где мы возвращаемся к локальности данных и обсуждаем дополнительные настройки для повышения уровня локальности.

Для достижения оптимальной производительности мы должны помнить о локальности, когда проектируем структуру программы. Следует избегать хаотического использования областей данных, если можно построить приложение, так чтобы конкретные наборы данных использовались в определенные промежутки времени.

Введение в векторизацию (SIMD)

Параллельное программирование – это про то, как поставить себе на службу механизмы параллельных вычислений, поддерживаемые оборудованием. В этой книге в центре нашего внимания является использование нескольких

процессорных ядер. Такой аппаратный параллелизм лежит в основе параллелизма на уровне потоков (или, более абстрактно, на уровне задач), который и реализует TBB в наших интересах.

Но есть и другой очень важный класс аппаратного параллелизма – векторные команды, т. е. команды, которые могут выполнять больше вычислений (параллельно), чем обычные. Например, обычная команда сложения принимает два числа, складывает их и возвращает один результат. Восьмипутевая векторная команда сложения умеет обрабатывать сразу восемь пар входных данных, складывать их и возвращать восемь результатов. Вместо C=A+B, как в обычной команде, векторная команда вычисляет $C_0=A_0+B_0$, $C_1=A_1+B_1$, $C_2=A_2+B_2$, ..., $C_7=A_7+B_7$. Производительность при этом возрастает в 8 раз. Но требуется наличие восьми пар данных, к которым можно одновременно применить одну и ту же операцию. Впрочем, в счетных программах такое встречается часто.

Способность выполнять несколько операций в одной команде называется «один поток команд, несколько потоков данных» (Single Instruction Multiple Data – SIMD), это одна из четырех категорий компьютерной архитектуры в так называемой классификации Флинна. Идея о том, что одна команда может применяться к нескольким наборам данных (например, слагаемым для восьми разных операций сложения), открывает возможность параллелизма, которой грех было бы не воспользоваться.

Векторизация – это технология, в которой задействован SIMD-параллелизм. Она зависит от компилятора, потому что именно компилятор решает, какие команды применить.

Мы могли бы игнорировать векторный параллелизм, сказав «это другая тема и другой предмет для изучения». Но не станем! В хорошей параллельной программе найдется место как параллелизму на уровне задач (средствами TBB), так и SIMD-параллелизму (с помощью векторизующего компилятора).

Наш совет: прочитайте главу 4 этой книги. Узнайте о возможностях векторизации своего любимого компилятора и пользуйтесь ими. В современных компиляторах для этой цели предназначена, в частности, `#pragma SIMD`. Разберитесь в библиотеке Parallel STL (глава 4 и приложение B), включая и то, почему она в общем случае не является наилучшим решением для эффективного параллельного программирования (по закону Амдала, распараллеливать программу лучше на более высоком уровне, чем вызовы STL).

Хотя векторизация важна, применение TBB дает более существенное ускорение в большинстве приложений (если приходится выбирать между тем и другим). Это связано с тем, что обычно в системе можно выявить больше возможностей распараллеливания между ядрами, чем позволяет ширина SIMD (количество элементов данных, обрабатываемых векторными командами). К тому же задачи – гораздо более общий механизм, чем ограниченное количество операций, реализованных SIMD-командами.

Хороший совет:	сначала займитесь разбиением программы на задачи (с помощью TBB), а уже затем векторизацией.
Лучший совет:	пользуйтесь тем и другим.

Использование обоих видов параллелизма полезно, когда в программе имеются вычисления, допускающие векторизацию. Рассмотрим 32-ядерный процессор с векторным расширением AVX. Многозадачная программа могла бы надеяться на приближение к теоретическому максимуму ускорения 32× за счет применения TBB. Векторизованная программа могла бы надеяться на приближение к теоретическому максимуму 4× благодаря векторизации математических операций с числами двойной точности. Но если использовать то и другое, то теоретический максимум подскакивает до 256× – именно из-за такого мультипликативного эффекта многие разработчики счетных программ применяют оба подхода.

Введение в средства C++ (в объеме, необходимом для работы с TBB)

Поскольку цель параллельного программирования – добиться масштабирования производительности на машинах с несколькими ядрами, языки C и C++ предлагают идеальное сочетание абстракции и упора на эффективность. В TBB используется C++, но способом, доступным программистам на C.

В каждой дисциплине есть своя терминология, и C++ не исключение. В конце книги имеется глоссарий, помогающий освоить терминологию C++, параллельного программирования, TBB и других дисциплин. Но некоторые вещи, чрезвычайно важные для программистов на C++, мы кратко рассмотрим здесь: лямбда-функции, обобщенное программирование, контейнеры, шаблоны, стандартную библиотеку шаблонов (STL), перегрузку, диапазоны и итераторы.

Лямбда-функции

Мы в восторге от включения лямбда-функций в стандарт C++11, поскольку они позволяют встраивать код вместо вызова в виде анонимной функции. С объяснениями мы подождем до главы 1, где они нам понадобятся, тогда и проиллюстрируем на примерах.

Обобщенное программирование

Парадигма обобщенного программирования предполагает написание алгоритмов, применимых к любому типу данных. Типы в этом случае можно рассматривать как параметры, которые «будут заданы позже». В C++ обобщенное программирование реализовано таким образом, что открывается возможность оптимизации компилятором и не нужно делать зависящий от заданного типа выбор на этапе выполнения. В результате современные компиляторы C++ сводят к минимуму накладные расходы, связанные с активным использованием обобщенного программирования, т. е. шаблонов и STL. Простой пример обобщенного программирования – алгоритм сортировки, который может отсортировать любой список элементов, если мы зададим способ доступа к элементу, а также функции обмена и сравнения двух элементов. Имея такой алгоритм,

мы можем конкретизировать его для сортировки списка целых чисел, чисел с плавающей точкой, комплексных чисел или строк – нужно только определить, как производится обмен и сравнение данных каждого типа.

Еще один пример обобщенного программирования – поддержка комплексных чисел. Части комплексного числа могут иметь тип `float`, `double` или `long double`. Вместо того чтобы объявлять три разных типа комплексных чисел и заводить три набора функций, оперирующих разными типами, мы можем создать обобщенный тип комплексного числа. В объявлении конкретной переменной нужно будет указывать тип частей комплексного числа, как показано ниже:

```
complex<float> my_single_precision_complex;
complex<double> my_double_precision_complex;
complex<long double> my_quad_precision_complex;
```

На самом деле эти типы поддерживаются стандартной библиотекой шаблонов C++ (STL), при условии что включен подходящий заголовочный файл.

Контейнеры

Контейнером в C++ называется структура для организации коллекции элементов данных. Контейнер сочетает объектно-ориентированные средства (изоляция кода, имеющего право манипулировать структурой данных) и механизмы обобщенного программирования (контейнеры могут работать с разными типами данных). Мы обсудим контейнеры, предоставляемые TBB, в главе 6. Контейнеры не критичны для использования TBB, они включены в основном для программистов на C++, которые уже умеют работать с чем-то похожим.

Шаблоны

Шаблоны представляют собой образцы, по которым создаются функции или классы (например, контейнеры). Это делается эффективно и в духе обобщенного программирования, т. е. при задании шаблона типы не специфицируются, но каждый откомпилированный экземпляр шаблона – конкретный класс или функция, что избавляет от накладных расходов, связанных с динамическим определением. Создать эффективный шаблон может быть очень трудно, но использовать его легко. Библиотека TBB основана на шаблонах.

При использовании TBB и других библиотек шаблонов мы можем рассматривать их как собрание вызовов функций. Тот факт, что на самом деле это шаблоны, влияет лишь на выбор компилятора – шаблоны не входят в язык C, так что нужно пользоваться C++. Современные компиляторы C++ сводят к минимуму издержки абстрагирования при интенсивном использовании шаблонов.

STL

Стандартная библиотека шаблонов C++ (STL) входит в стандарт C++. Любой компилятор C++ обязан поддерживать STL. В состав STL входят четыре компонента: алгоритмы, контейнеры, функции и итераторы.

Концепция «обобщенного программирования» снизу доверху пронизывает STL, основанную на шаблонах. Такие алгоритмы STL, как `sort`, не зависят от типов данных (контейнеров). STL поддерживает составные типы данных, включая контейнеры и ассоциативные массивы, которые могут быть основаны на любых встроенных и пользовательских типах, если только они поддерживают некоторые элементарные операции (например, копирование и присваивание).

Перегрузка

Перегрузка операторов – объектно-ориентированная концепция, позволяющая использовать новые типы данных (например, `complex`) в контекстах, где изначально допустимы встроенные типы (например, `int`), например в качестве аргументов функций или таких операторов, как `=` и `+`. Шаблоны C++ принесли с собой обобщенную перегрузку, которую можно рассматривать как распространение перегрузки на имена функций с различными параметрами и (или) возвращаемые значения. Цель обобщенной перегрузки с помощью шаблонов или перегрузки в объектно-ориентированном смысле одна – *полиморфизм*, т. е. способность обрабатывать данные по-разному в зависимости от их типов, но при этом использовать код повторно. TBB справляется с этим хорошо, а мы, пользователи, можем просто наслаждаться плодами.

Диапазоны и итераторы

Если вы хотите в каком-нибудь баре поспорить со знатоками C++, скажите, что «итераторы» намного лучше «диапазонов» или наоборот. Комитет по языку C++ в общем и целом отдает предпочтение диапазонам как интерфейсу более высокого уровня, а итераторы приберегает для будущих версий стандарта. Но и это простое утверждение может развязать войну в баре. Эксперты по C++ говорят нам, что диапазон – это подход к обобщенному программированию, в котором используется абстракция последовательностей, а не абстракция итераторов. Голова еще не пошла кругом?

Для пользователей идея итератора мало чем отличается от идеи диапазона: то и другое – лаконичный способ обозначить множество (и указание на то, как его обойти). Множество целых чисел от `0` до `999999` математически можно обозначить как `[0,999999]` или `[0,1000000)`. Заметим, что квадратные скобки [и] означают, что соответствующий конец включается, а круглые (и) – что не включается. В TBB применяется синтаксическая конструкция `blocked_range<size_t>(0,1000000)`.

Мы любим диапазоны, потому что они точно отвечают нашему желанию определить «потенциальный параллелизм», а не «принудительный параллелизм». Рассмотрим цикл «параллельный for» с миллионом итераций. Можно было бы создать миллион потоков, выполняющих работу параллельно, или один поток с диапазоном `[0,1000000)`. Этот диапазон можно разбить на части в соответствии с располагаемым параллелизмом, потому-то мы и любим диапазоны.

TBB поддерживает и использует итераторы и диапазоны, они часто будут встречаться нам на страницах этой книги. Начиная с главы 2 недостатка

в примерах не будет. Мы покажем, когда использовать одно, а когда другое, не вступая в дискуссию о том, почему в C++ существуют оба механизма. Глубокое понимание того, чем итераторы отличаются от диапазонов в C++, не поможет нам лучше использовать TBB. Пока достаточно простого объяснения – итераторы менее абстрактны, чем диапазоны, и, как минимум, это ведет к тому, что в коде с применением итераторов обычно нужно передавать два параметра: `something.begin()` и `something.end()`, хотя мы всего-то хотим сказать «использовать этот диапазон – от начала до конца».

Резюме

Мы кратко обсудили, как «мыслить параллельно» с точки зрения декомпозиции, масштабируемости, корректности, абстракции и паттернов. Мы познакомились с локальностью как ключевым вопросом всего параллельного программирования. Мы объяснили, что использование задач вместо потоков – главное революционное достижение технологии параллельного программирования, поддерживаемой TBB. Мы познакомились с элементами программирования на C++, выходящими за рамки C, но необходимыми для использования TBB.

Раз эти идеи крутятся у вас в голове, значит, вы уже начали мыслить параллельно. Вы развиваете интуитивное понимание параллелизма, которое вам хорошо послужит.

Это предисловие, предметный указатель и глоссарий помогут вам при чтении книги и в процессе изучения параллельного программирования с помощью TBB.

Дополнительная информация

- Стандарт C++ // https://isocpp.org/std/the-standard.
- «The Problem with Threads» by Edward Lee, 2006. IEEE Computer Magazine, May 2006 // http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.306.9963&rep=rep1&type=pdf или U. C. Berkley Technical Report: www2.eecs.berkeley.edu/Pubs/ TechRpts/2006/EECS-2006-1.pdf.
- Все примеры кода из этой книги доступны по адресу https://github.com/Apress/pro-TBB.

ЧАСТЬ I

Глава 1

Приступаем: «Hello, TBB!»

Спустя 10 лет после выпуска первой версии библиотека Threading Building Blocks (TBB) стала одной из самых широко распространенных библиотек для параллельного программирования на C++. Сохранив базовую философию и средства, она продолжает расширяться, включая новые возможности и отвечая на новые вызовы по мере их возникновения.

В этой главе мы обсудим, почему была создана TBB, дадим краткий обзор ее компонентов, опишем, как установить библиотеку, а затем перейдем к нескольким простым примерам.

Почему именно Threading Building Blocks?

Параллельное программирование имеет долгую историю, уходящую в 1950-е годы и еще раньше. На протяжении многих десятилетий ученые разрабатывали крупномасштабные параллельные модели для суперкомпьютеров, а компании создавали приложения масштаба предприятия для больших многопроцессорных машин. Но лишь чуть больше 10 лет назад на рынке появились первые многоядерные кристаллы, предназначенные для настольных компьютеров и ноутбуков. И это радикально изменило ситуацию.

В первых многоядерных системах количество процессоров было невелико – всего два ядра, – но число программистов, вовлеченных в параллельное программирование, было огромным. Если многоядерным процессорам суждено стать обыденностью, то такая же судьба ждет и параллельное программирование, особенно среди тех, кто ценит высокую производительность.

Первая версия библиотеки TBB вышла в сентябре 2006 года в ответ на уникальные вызовы выходящего на широкую арену параллельного программирования. И тогда, и сейчас ее главная цель – предложить разработчикам простой и эффективный способ создавать приложения, которые хорошо масштабировались бы на новые платформы с другими архитектурами и увеличенным количеством ядер. Эта «ориентация на будущее» принесла плоды, когда количество ядер в наиболее популярных процессорах увеличилось с двух в 2006 году до 64 и более в 2018-м!

Чтобы достичь поставленной цели, в основу философии TBB была положена идея сделать так, чтобы разработчику было легко выразить параллелизм своего приложения, ограничив в то же время средства управления отображением этого параллелизма на имеющееся оборудование. Некоторым опытным парал-

лельным программистам эта философия может показаться противоречащей интуиции. Если вы считаете, что параллельное программирование должно любой ценой выжать всю производительность до последней капли и для этой цели годится программирование «на уровне железа», ручная настройка и оптимизация, то TBB, наверное, не для вас – а для тех разработчиков, которые хотят писать высокопроизводительные приложения на сегодняшних платформах, но готовы поступиться толикой производительности ради того, чтобы эти приложения хорошо работали и в будущих системах.

Для этого интерфейсы TBB позволяют выразить параллелизм приложения, но оставляют библиотеке достаточную гибкость, чтобы она могла эффективно отобразить этот параллелизм на текущие и будущие платформы и адаптироваться к динамическому изменению системных ресурсов во время выполнения.

Производительность: низкие накладные расходы, большое преимущество у C++

Мы не хотим придавать чрезмерно большое значение потере производительности и не собираемся отрицать ее. Для простой программы на C++, написанной в стиле Fortran, когда имеются одноуровневые хорошо сбалансированные параллельные циклы, динамическая природа TBB и не нужна вовсе. Однако ограничения такого стиля программирования – важная причина существования TBB. Библиотека TBB спроектирована так, чтобы эффективно поддерживать композицию вложенных, конкурентных и последовательных конструкций и динамически отображать такой параллелизм на целевую платформу. Используя *компонуемую* библиотеку типа TBB, разработчик может создавать приложения путем комбинирования компонентов и библиотек, содержащих внутри себя параллелизм, не заботясь о возможном негативном влиянии этих компонентов друг на друга. Важно, что TBB не требует от нас ограничить возможные виды параллелизма, чтобы избежать проблем. Поэтому для больших и сложных приложений, написанных на C++, TBB можно рекомендовать без всяких оговорок.

Библиотека TBB развивается уже несколько лет, не только поддерживая новые платформы, но и реагируя на пожелания программистов, которые хотят иметь немного больше контроля над решениями, принимаемыми библиотекой при отображении параллелизма на оборудование. Если в TBB 1.0 было совсем мало средств управления производительностью со стороны пользователя, то в TBB 2019 их намного больше, включая управление привязкой к процессору, конструкции для изоляции работ, методы закрепления ядер за потоками и т. д. Разработчики TBB усердно трудятся, чтобы новые средства обеспечили подходящий уровень контроля, не жертвуя компонуемостью.

Предлагаемые библиотекой интерфейсы образуют несколько уровней. В TBB имеются высокоуровневые шаблоны, отвечающие потребностям большинства программистов и покрывающие наиболее распространенные случаи. Но есть также интерфейсы низкого уровня, чтобы можно было создавать решения, точно настроенные на нужды конкретных приложений. TBB берет лучшее

из обоих миров. Как правило, мы полагаемся на то, что библиотека выбирает по умолчанию, но для достижения максимальной производительности можем углубиться в детали.

Эволюция поддержки параллелизма в TBB и C++

С момента появления первой версии TBB и сама библиотека, и язык C++ прошли большой путь. В 2006 году в C++ не было поддержки параллельного программирования на уровне языка, а многие библиотеки, включая STL, было трудно использовать в параллельных программах, т. к. они не были потокобезопасными.

Комитет по языку C++ озаботился включением средств многопоточности прямо в язык и библиотеку STL. На рис. 1.1 показаны новые и планируемые средства C++, относящиеся к параллелизму.

	C++11/14	C++17	C++2x (предложения)
Верхний уровень (события, сообщения, потоковые графы)	`std::async`, `std::future`	`std::async`, `std::future`	Возобновляемые функции, исполнители
Разветвление–соединение, многопоточность	`std::thread` + синхронизация	STL с политикой `par`	Блок задачи, цикл `for`
SIMD/векторизация	ничего	STL с политикой `par_unseq`	STL с политиками `unseq` и `vec`, векторные типы SIMD

Рис. 1.1 ❖ Средства, определенные в стандарте C++, и новые предложения

Несмотря на всю нашу любовь к TBB, мы предпочли бы, чтобы вся фундаментальная поддержка параллелизма была встроена в сам язык C++. Это позволило бы TBB опереться на прочное основание и уже на нем возводить высокоуровневые абстракции параллелизма. В первых версиях TBB приходилось компенсировать отсутствие поддержки на уровне языка, но в этой области стандарт C++ значительно усовершенствован и теперь заполняет те пробелы, которые раньше приходилось восполнять TBB, например переносимые блокировки и атомарные типы. К сожалению для разработчиков на C++, в стандарте все еще недостает некоторых средств для полной поддержки параллельного программирования. Но к радости читателей, это означает, что TBB по-прежнему актуальна и важна для эффективного программирования многопоточности на C++ и, вероятно, останется таковой еще много лет.

Очень важно понимать, что мы не сетуем по поводу процесса разработки стандарта C++. При добавлении новых средств в стандарт языка следует продвигаться с большой осторожностью и тщательно анализировать все предложения. Например, комитет по стандартизации C++11 потратил очень много усилий на утверждение модели памяти. Этот вопрос критически важен для любой библиотеки параллельного программирования, основанной на стандарте. Существуют также некоторые пределы тому, что должен включать и поддерживать стандарт языка. Мы полагаем, что система задач и потоковые графы, присутствующие в TBB, не должны входить в стандарт напрямую. Даже если мы не правы, случится это очень не скоро.

Недавние добавления в C++, относящиеся к параллелизму

Как показано на рис. 1.1, в стандарт C++11 включены некоторые низкоуровневые средства поддержки многопоточности, в т. ч. `std::async`, `std::future` и `std::thread`. Включены также атомарные переменные, мьютексы и условные переменные. Чтобы с помощью этих расширений создать высокоуровневые абстракции, программисту придется немало поработать, но они все же позволяют выразить простой параллелизм непосредственно на C++. Стандарт C++11, безусловно, стал значительным достижением с точки зрения многопоточности, но он не предоставляет высокоуровневых средств, позволяющих легко писать переносимый и эффективный параллельный код. В нем также отсутствуют задачи и планировщик с заимствованием работ.

Стандарт C++17 поднял уровень абстракции на новую высоту, позволив выражать параллелизм, не беспокоясь о мельчайших низкоуровневых деталях. Но, как мы увидим ниже, по-прежнему имеются существенные ограничения, так что новые средства пока недостаточно выразительны и не могут похвастаться высочайшей производительностью – авторам стандарта C++ предстоит еще много работы.

Из добавлений в C++17 нас больше всего интересуют *политики выполнения*, которые можно использовать совместно с алгоритмами из библиотеки STL. Они позволяют выбрать режим работы алгоритма: безопасно распараллелить, векторизовать, распараллелить и векторизовать или же сохранить первоначальную последовательную семантику. Реализацию STL, поддерживающую эти политики, мы называем Parallel STL.

В будущем в стандарт C++, возможно, будут включены дополнительные средства параллелизма, как то: возобновляемые функции, исполнители, блоки задач, параллельные циклы for, векторные типы SIMD и дополнительные политики выполнения для алгоритмов STL.

Библиотека Threading Building Blocks (TBB)

Threading Building Blocks (TBB) – написанная на C++ библиотека, выполняющая две основные функции:

1) заполняет пробелы в поддержке параллелизма там, где стандарт C++ еще недостаточно развит или новые средства не полностью поддерживаются компиляторами;
2) предоставляет высокоуровневые абстракции параллелизма, которые, вероятно, никогда не будут включены в стандарт C++. На рис. 1.2 показаны средства, реализованные TBB.

Эти средства можно отнести к двум большим группам: интерфейсы для выражения параллельных вычислений и интерфейсы, не зависящие от модели выполнения.

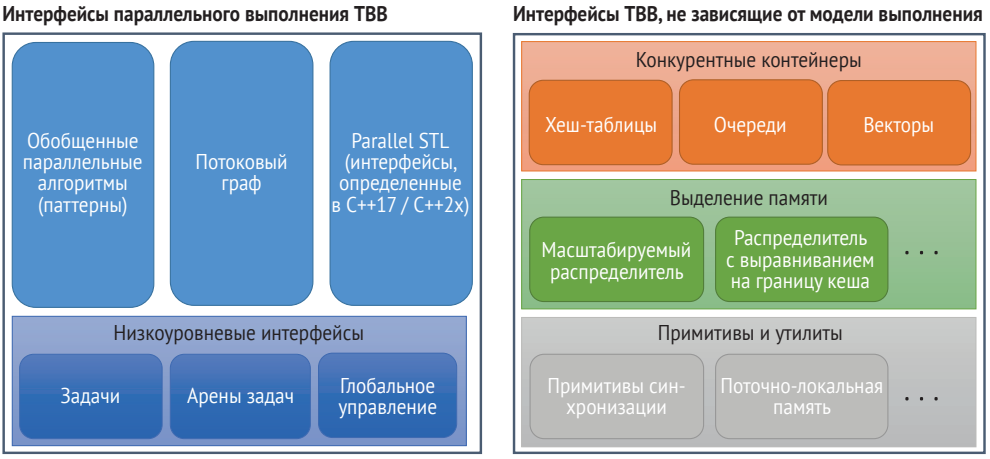


Рис. 1.2 ❖ Средства библиотеки TBB

Интерфейсы параллельного выполнения

Используя TBB для создания параллельных программ, мы выражаем параллелизм приложения с помощью одного из высокоуровневых интерфейсов или непосредственно в виде задач. Задачи мы подробно обсудим ниже, а пока будем рассматривать TBB-задачу как облегченный объект, который определяет небольшое вычисление и связанные с ним данные. Работая с TBB, мы описываем свое приложение с помощью задач – прямо или опосредованно через включенные в TBB алгоритмы, – а библиотека планирует, как выполнить эти задачи, располагая предлагаемыми платформой аппаратными ресурсами.

Важно отметить, что у разработчика может возникнуть желание выразить различные виды параллелизма. На рис. 1.3 показаны три самых распространенных уровня параллелизма в приложениях. Одни приложения включают все три уровня, другие – только один или два. Один из самых «крутых» аспектов TBB – предоставление высокоуровневых интерфейсов для всех трех уровней, что позволяет использовать одну библиотеку во всех случаях.

Уровень обмена сообщениями выражает параллелизм в случае, когда программа структурирована как относительно крупные вычисления, обменивающиеся между собой явно определенными сообщениями. На этом уровне возникают такие паттерны, как потоковые диаграммы, графы потоков данных и графы зависимостей. В TBB все они поддерживаются интерфейсами потокового графа (Flow Graph), описанными в главе 3.

Уровень разветвления–соединения включает паттерны, в которых последовательное вычисление разветвляется на множество параллельных задач и продолжается, после того как все параллельные подвычисления завершатся. Примерами таких паттернов могут служить функциональный параллелизм (параллелизм задач), параллельные циклы, параллельная редукция и конвейеры. TBB поддерживает их с помощью обобщенных параллельных алгоритмов (глава 2).

Наконец, на уровне SIMD параллелизм выражается в применении одной и той же операции одновременно к нескольким элементам данных. Этот тип

параллелизма часто реализуется с помощью таких векторных расширений, как AVX, AVX2 или AVX-512, которые используют устройства векторных вычислений, имеющиеся в каждом процессорном ядре. Библиотека Parallel STL (глава 4), включенная во все дистрибутивы TBB, содержит среди прочего векторные реализации, опирающиеся на эти расширения.

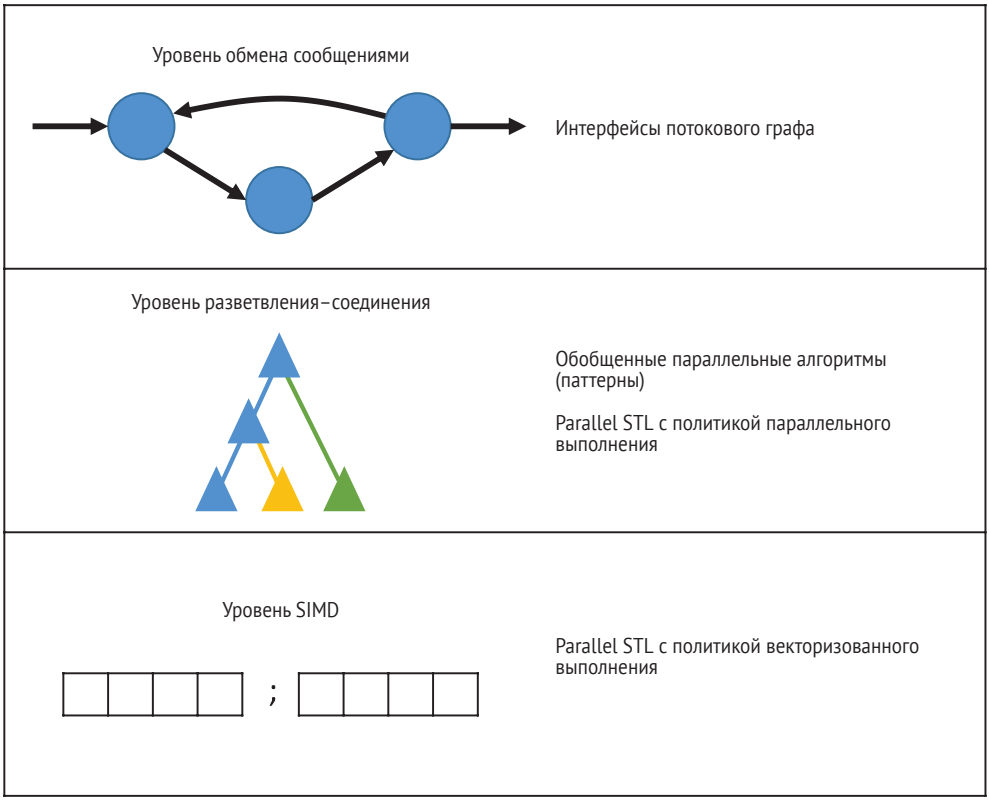


Рис. 1.3 ❖ Три уровня параллелизма, часто встречающихся в приложениях, и их отображение на высокоуровневые интерфейсы параллельного выполнения в TBB

TBB предлагает высокоуровневые интерфейсы для многих распространенных параллельных паттернов, но все же бывают случаи, когда ни один готовый интерфейс не подходит. Тогда можно напрямую использовать в алгоритмах задачи TBB.

Истинная мощь интерфейсов параллельного выполнения в TBB проистекает из возможности комбинировать их, это свойство обычно называют «компонуемостью». Мы можем создавать приложения, на верхнем уровне которых находится потоковый граф, а в его узлах – вложенные обобщенные параллельные алгоритмы. Каждый такой алгоритм может при необходимости включать алгоритмы из библиотеки Parallel STL. Поскольку параллелизм, выражаемый с помощью любого из трех уровней, виден библиотеке TBB, она может эффективно планировать соответствующие задачи так, что они будут допускать композицию и использовать платформенные ресурсы оптимальным способом.

Одно из основных свойств TBB, благодаря которому она компонуема, – поддержка *ослабленной последовательной семантики*. Это означает, что параллелизм, выраженный средствами TBB, – не более чем подсказка библиотеке, и нет никакой гарантии, что любая конкретная задача будет выполняться параллельно с другими. Это открывает перед TBB невероятно гибкие возможности для планирования задач с целью повышения производительности. Благодаря этой гибкости библиотека обеспечивает переносимую производительность на разных системах – с одним, 8 или 80 ядрами. Она также позволяет библиотеке адаптироваться к динамической нагрузке на платформу; например, если одно ядро перегружено работой, то TBB может запланировать больше работы на других ядрах или даже решить, что параллельный алгоритм будет выполняться только одним ядром. Подробнее о том, почему TBB считается компонуемой библиотекой, мы поговорим в главе 9.

Интерфейсы, не зависящие от модели выполнения

В отличие от интерфейсов параллельного выполнения, средства из второй большой группы на рис. 1.2 вообще не зависят от модели выполнения и задач TBB. Эти средства полезны в приложениях, где используются платформенные потоки, например `pthreads` или `WinThreads`, а также в приложениях с задачами TBB.

К числу этих средств относятся конкурентные контейнеры, которые предоставляют потокобезопасные интерфейсы к таким популярным структурам данных, как хеш-таблицы, очереди и векторы. Сюда же относятся средства выделения памяти – масштабируемый распределитель памяти TBB и распределитель с выравниванием на границу кеша (оба описаны в главе 7). В эту группу входят и такие низкоуровневые средства, как примитивы синхронизации и поточно-локальная память.

Использование строительных блоков в TBB

Разработчик может выбирать, какие части TBB полезны в его приложении. Например, можно использовать масштабируемый распределитель памяти (глава 7) и больше ничего. Или воспользоваться конкурентными контейнерами (глава 6) и несколькими обобщенными параллельными алгоритмами (глава 2). И конечно, мы вправе построить приложение, в котором сочетаются все три высокоуровневых интерфейса выполнения, и пользоваться масштабируемым распределителем памяти, конкурентными контейнерами и многими другими возможностями, предлагаемыми библиотекой.

Да начнем же уже!

Получение библиотеки TBB

Первым делом мы должны откуда-то получить копию библиотеки. Путей несколько, перечислим те, что существовали на момент написания книги.

- Перейти по ссылкам на сайте www.threadingbuildingblocks.org или https://software.intel.com/intel-tbb и получить бесплатную библиотеку TBB непосредственно от Intel. Существуют готовые откомпилированные версии для Windows, Linux и macOS. В самые последние пакеты включена как сама TBB, так и реализация алгоритмов из библиотеки Parallel STL, в которой для организации многопоточности используется TBB.
- Зайти на сайт https://github.com/intel/tbb и скачать бесплатную версию TBB с открытым исходным кодом. Эта не облегченная версия, она включает все те же средства, что и коммерческая. Вы можете извлечь исходный код из репозитория и собрать библиотеку самостоятельно или просто щелкнуть по ссылке «releases» и скачать версию, уже собранную и протестированную Intel. На GitHub имеются собранные и протестированные версии для Windows, Linux, macOS и Android. Как и в первом случае, последние пакеты содержат как саму библиотеку TBB, так и реализацию Parallel STL, в которой используется TBB. Если вам нужен исходный код Parallel STL, скачайте его отдельно на странице https://github.com/intel/parallelstl.
- Можно скачать комплект инструментов Intel Parallel Studio XE с сайта https://software.intel.com/intel-parallel-studio-xe. В настоящее время TBB и версия Parallel STL, использующая TBB, включены во все издания этого комплекта, в т. ч. самое маленькое – Composer Edition. Если на вашей машине установлена недавняя версия компилятора Intel C++, то, вероятно, установлена и TBB.

Предлагаем читателям самостоятельно выбрать подходящий способ получения TBB и выполнить инструкции по установке пакета, выложенные на соответствующем сайте.

Получение кода примеров

Код всех примеров, встречающихся в этой книге, выложен по адресу https://github.com/Apress/pro-TBB. В этом репозитории каждой главе соответствует отдельный каталог. Многие исходные файлы названы по номеру соответствующего рисунка, например файл `ch01/fig_1_04.cpp` содержит код, приведенный на рис. 1.4 в этой главе.

Написание первого примера «Hello, TBB!»

На рис. 1.4 приведен небольшой пример, в котором функция `tbb::parallel_invoke` выполняет две функции, одна из которых печатает `Hello`, а вторая параллельно печатает `TBB!`. Пример тривиален и от распараллеливания ничего не выигрывает, но позволяет убедиться, что среда правильно настроена для работы TBB. На рис. 1.4 включен заголовок tbb.h, открывающий доступ к функциям и классам TBB, которые находятся в пространстве имен `tbb`. Обращение к `parallel_invoke` сообщает библиотеке TBB, что обе переданные функции независимы друг от друга, поэтому их можно безопасно выполнять параллельно на разных ядрах или в разных потоках в любом порядке. При таких ограничениях первым может быть напечатано как слово `Hello`, так и слово `TBB!`. Может даже случиться,

что между этими словами нет знака новой строки, а оба таких знака оказались напечатаны в самом конце, поскольку печать строки и следующего за ней символа `std::endl` – не атомарная операция.

```
#include <iostream>
#include <tbb/tbb.h>

int main() {
  tbb::parallel_invoke(
    []() { std::cout << " Hello " << std::endl; },
    []() { std::cout << " TBB! " << std::endl; }
  );
  return 0;
}
```

Рис. 1.4 ❖ Пример Hello TBB

На рис. 1.5 приведен пример, в котором алгоритм Parallel STL `std::for_each` применяет функцию параллельно к двум элементам `std::vector`. Передача политики `pstl::execution::par` алгоритму `std::for_each` подтверждает безопасность применения указанной функции параллельно на разных ядрах или в разных потоках к результату разыменования каждого итератора в диапазоне `[v.begin(), v.end())`. Как и в предыдущем примере, первой может быть напечатана любая из двух строк.

```
#include <pstl/algorithm>
#include <pstl/execution>
#include <iostream>
#include <vector>

int main() {
  std::vector<std::string> v = { " Hello ", " Parallel STL! " };
  std::for_each(pstl::execution::par, v.begin(), v.end(),
    [](std::string& s) { std::cout << s << std::endl; }
  );
  return 0;
}
```

Рис. 1.5 ❖ Пример Hello Parallel STL

На обоих рис. 1.4 и 1.5 для задания функций используются *лямбда-выражения C++*. При работе с библиотеками типа TBB лямбда-выражения очень полезны, когда нужно задать пользовательский код, выполняемый как задача. На врезке «Введение в лямбда-выражения C++» приведен краткий обзор этой важной особенности современного C++.

Введение в лямбда-выражения C++

Поддержка лямбда-выражений впервые появилась в C++11. Они служат для создания анонимных объектов-функций (которые при желании можно присвоить именованным переменным), способных захватывать переменные из объемлющей области видимости.

Базовый синтаксис лямбда-выражений выглядит так:

[*список-захвата* **] (** *параметры* **) ->** *возвр* **{** *тело* **}**

где:

- *список-захвата* – список захватываемых переменных через запятую. Чтобы захватить переменную по значению, ее имя следует указать в *списке-захвата*. Чтобы захватить переменную по ссылке, ее имени нужно предпослать знак амперсанда, например `&v`. Этот способ можно также использовать для захвата по ссылке текущего объекта `this`. Существуют также умолчания: [=] означает захват всех автоматических переменных, встречающихся в теле, по значению, а текущего объекта – по ссылке; [&] означает захват по ссылке всех автоматических переменных, встречающихся в теле, и текущего объекта; [] означает, что не нужно захватывать ничего;
- *параметры* – список параметров, такой же, как для именованной функции;
- *возвр* – тип возвращаемого значения. Если часть ->*возвр* опущена, она выводится из предложений `return` в теле функции;
- *тело* – тело функции.

В следующем примере показано лямбда-выражение C++, захватывающее переменную `i` по значению, а переменную `j` по ссылке. У него также имеется параметр `k0`, передаваемый по значению, и параметр `l0`, передаваемый по ссылке.

```
int i = 1, j = 10, k = 100, l = 1000;
auto lambdaExpression = [i, &j] (int k0, int &l0) -> int {
  j = 2 * j;
  k0 = 2 * k0;
  l0 = 2 * l0;
return i + j + k0 + l0;
  };

printValues(i, j, k, l);
std::cout << "First call returned " << lambdaExpression(k, l) << std::endl;
printValues(i, j, k, l);
std::cout << "Second call returned " << lambdaExpression(k, l) << std::endl;
printValues(i, j, k, l);
return 0;
```

При выполнении этого примера печатается:

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

Лямбда-выражение можно рассматривать как экземпляр объекта-функции, только определение класса создает компилятор. Например, рассмотренное выше лямбда-выражение эквивалентно экземпляру такого класса:

```
class Functor {
  int my_i;
  int &my_jRef;

public:
  Functor(int i, int &j) : my_i{i}, my_jRef{j} { }

  int operator()(int k0, int &l0) {
    my_jRef = 2 * my_jRef;
    k0 = 2 * k0;
    l0 = 2 * l0;
    return my_i + my_jRef + k0 + l0;
  }
};
```

Всякий раз, видя лямбда-выражение C++, мы можем подставить вместо него экземпляр объекта-функции, аналогичного показанному выше. На самом деле библиотека TBB предвосхитила стандарт C++11 и все его интерфейсы, в которых требуется передавать экземпляры объектов пользовательских классов. Лямбда-выражения C++ упрощают работу с TBB, устраняя лишний шаг – определение отдельного класса для каждого использования алгоритма TBB.

Сборка простых примеров

Написав код примеров на рис. 1.4 и 1.5, мы должны собрать для них исполняемые файлы. Инструкции по сборке приложения с использованием TBB зависят от ОС и компилятора. Но всегда есть два шага, необходимых для правильной настройки среды.

Настройка среды

1. Мы должны сообщить компилятору, где находятся заголовки и библиотеки, входящие в состав TBB. Если используются интерфейсы библиотеки Parallel STL, необходимо еще сообщить, где находятся ее заголовки.
2. Следует настроить среду, так чтобы запущенное приложение могло найти библиотеку TBB. TBB поставляется в виде динамически компонуемой библиотеки, т. е. не встраивается непосредственно в приложение; вместо этого приложение находит и загружает ее на этапе выполнения. Для интерфейсов Parallel STL не нужна собственная динамически компонуемая библиотека, но они зависят от библиотеки TBB.

Теперь мы кратко обсудим, как эти шаги выполняются в Windows и в Linux. Инструкции для macOS такие же, как для Linux. В документации, поставляемой вместе с TBB, описаны дополнительные случаи и даны более подробные указания.

Сборка в Windows в Microsoft Visual Studio

Если была скачана версия TBB с коммерческой поддержкой или версия Intel Parallel Studio XE, то библиотеку TBB можно интегрировать с Microsoft Visual Studio на этапе установки, а затем без труда использовать из Visual Studio.

Для создания проекта «Hello, TBB!» создайте проект в Visual Studio как обычно, добавьте cpp-файл с кодом на рис. 1.4 или 1.5, а затем перейдите на страницы свойств проекта, выполните команду **Configuration Properties ➤ Intel Performance Libraries** (Свойства конфигурации ➤ Intel Performance Libraries) и измените флажок **Use TBB** (Использовать TBB) на **Yes** (Да), как показано на рис. 1.6. На этом шаг 1 можно считать выполненным. Теперь Visual Studio будет компоновать проект с библиотекой TBB, поскольку знает о путях к заголовочным файлам и библиотекам. При этом также устанавливаются правильные пути к заголовкам Parallel STL.

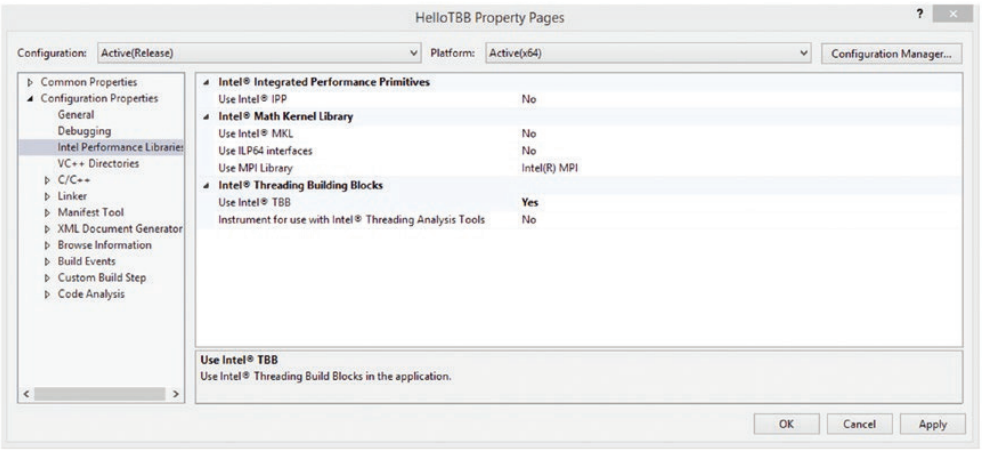


Рис. 1.6 ❖ Задание флажка **Использовать TBB** в страницах свойств проекта в Visual Studio

В системах Windows библиотеки TBB, которые динамически загружаются исполняемым файлом приложения на этапе выполнения, – это dll-файлы. Чтобы выполнить шаг 3 подготовки среды, необходимо добавить путь к этим файлам в переменную среды PATH. Годится как пользовательская, так и системная переменная PATH. Один из способов найти эти параметры – открыть панель управления Windows, пройти по пути **Система ➤ Дополнительные параметры системы ➤ Переменные среды**. Где находятся dll-файлы TBB, можно узнать из документации.

Примечание. Измененная переменная PATH становится видна среде Microsoft Visual Studio только после ее перезагрузки.

Итак, исходный код введен, флажок **Использовать TBB** установлен в **Да**, и путь к dll-файлам TBB прописан в переменной PATH. Теперь можно собрать и выполнить программу, нажав **Ctrl**+**F5**.

Сборка на платформе Linux из терминала

Использование компилятора Intel

При использовании компилятора Intel C++ процесс компиляции упрощается, потому что библиотека входит в дистрибутив компилятора и поддерживается флаг `-tbb`, который правильно устанавливает пути к библиотеке и заголовочным файлам. Поэтому для компиляции примеров нужно только добавить в командную строку флаг `-tbb`:

```
icpc -std=c++11 -tbb -o fig_1_04 fig_1_04.cpp
icpc -std=c++11 -tbb -o fig_1_05 fig_1_05.cpp
```

Скрипты tbbvars и pstlvars

Если компилятор Intel C++ не используется, то для настройки среды можно использовать скрипты, входящие в состав дистрибутивов TBB и Parallel STL. Эти скрипты модифицируют переменные среды `CPATH`, `LIBRARY_PATH` и `LD_LIBRARY_PATH`, включая в них каталоги, необходимые для сборки и выполнения приложений с TBB и Parallel STL. В переменную `CPATH` добавляются каталоги, которые компилятор просматривает, когда ищет заголовочные файлы. В переменную `LIBRARY_PATH` добавляются каталоги, которые компилятор просматривает, когда ищет библиотеки, компонуемые на этапе сборки. А в переменную `LD_LIBRARY_PATH` добавляются каталоги, которые компилятор просматривает, когда ищет библиотеки, динамически загружаемые на этапе выполнения.

Предположим, что корневой каталог нашей установки TBB записан в переменную среды `TBB_ROOT`. В дистрибутив библиотеки TBB входит набор скриптов в каталоге `${TBB_ROOT}/bin`, при выполнении которых правильно настраивается среда. Скрипту нужно передать тип архитектуры [`ia32|intel64|mic`]. Кроме того, на этапе компиляции нужно указать флаг, разрешающий использовать средства из стандарта C++11, в частности лямбда-выражения.

Хотя заголовки Parallel STL включаются во все недавние дистрибутивы библиотеки TBB, все равно необходим дополнительный шаг, чтобы добавить их в среду. Как и для TBB, для Parallel STL имеется набор скриптов в каталоге `${PSTL_ROOT}/bin`. Обычно каталог `PSTL_ROOT` расположен на одном уровне с `TBB_ROOT`. Для использования Parallel STL необходимо также передать тип архитектуры и разрешить использование средств C++11.

Шаги сборки и выполнения примера 1.4 на платформе Linux с 64-разрядным процессором Intel выглядят так:

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp -ltbb
./fig_1_04
```

Шаги сборки и выполнения примера 1.5 на той платформе выглядят так:

```
source ${TBB_ROOT}/bin/tbbvars.sh intel64 linux auto_tbbroot
source ${PSTL_ROOT}/bin/pstlvars.sh intel64 auto_pstlroot
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp -ltbb
./fig_1_05
```

Примечание. Все больше дистрибутивов Linux уже включают библиотеку TBB. На этих платформах компилятор GCC может осуществлять компоновку с платформенной версией TBB, а не с той, которая включена в переменную LIBRARY_PATH скриптом tbbvars. Если возникают проблемы с использованием TBB, то причина может быть именно в этом. В таком случае можно добавить явный путь к библиотеке в командную строку вызова компилятора, указав конкретную версию TBB, например:

```
g++ -L${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb ...
```

Если включить в командную строку g++ флаги -Wl,--verbose, то будет выдан отчет обо всех скомпонованных библиотеках, что поможет диагностировать эту проблему.

Хотя в примерах выше приведены команды для g++, они будут такими же (за исключением названия компилятора) для компиляторов Intel (icpc) и LLVM (clang++).

Задание переменных среды вручную без использования скрипта tbbvars или компилятора Intel

Иногда мы не хотим использовать скрипт tbbvars – то ли потому, что не знаем точно, какие переменные он устанавливает, то ли потому, что нужно интегрироваться с системой сборки. Если к вам это не относится, можете пропустить этот раздел и вернуться к нему, когда возникнет необходимость в ручной настройке.

Но раз уж вы читаете данный раздел, давайте посмотрим, как произвести сборку и выполнение из командной строки, не прибегая к скрипту tbbvars. Ни один компилятор, кроме Intel, не поддерживает флаг -tbb, так что придется задавать пути к заголовкам и разделяемым библиотекам TBB.

Если корневой каталог установки TBB записан в переменную среды TBB_ROOT, то заголовки находятся в каталоге ${TBB_ROOT}/include, а разделяемые библиотеки – в каталоге ${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}, где ARCH – системная архитектура [ia32|intel64|mic], а GCC_LIB_VERSION – версия библиотеки TBB, совместимая с установленным компилятором GCC или clang.

Различие между версиями библиотеки TBB связано с зависимостями от средств библиотек времени выполнения C++ (таких как libstdc++ или libc++).

Обычно для нахождения подходящей версии TBB мы можем использовать команду gcc -version в окне терминала. Затем нужно выбрать ближайшую из доступных версий в каталоге ${TBB_ROOT}/lib/${ARCH}, но не более позднюю, чем установленная версия GCC (обычно это работает даже при использовании компилятора clang++). Но поскольку установка может зависеть от машины, а мы вправе выбирать различные комбинации компилятора и среды выполнения C++, этот простой подход не всегда срабатывает. Если что-то пошло не так, обратитесь к документации по TBB за дополнительными инструкциями.

Например, в системе, где установлен компилятор GCC 5.4.0, мы компилировали пример на рис. 1.4 командой

```
g++ -std=c++11 -o fig_1_04 fig_1_04.cpp    \
   -I ${TBB_ROOT}/include                  \
   -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

И при использовании clang++ использовали ту же самую версию TBB:

```
clang++ -std=c++11 -o fig_1_04 fig_1_04.cpp \
   -I ${TBB_ROOT}/include                   \
   -L ${TBB_ROOT}/lib/intel64/gcc-4.7 -ltbb
```

Для компиляции примера на рис. 1.5 нужно еще указать путь к каталогу заголовочных файлов Parallel STL:

```
g++ -std=c++11 -o fig_1_05 fig_1_05.cpp     \
   -I ${TBB_ROOT}/include                   \
   -I ${PSTL_ROOT}/include                  \
   -L ${TBB_ROOT}/lib/intel64/gcc4.7 -ltbb
```

Вне зависимости от используемого компилятора (Intel, gcc или clang++) необходимо указать путь к разделяемой библиотеке TBB в переменной `LD_LIBRARY_PATH`, чтобы система могла найти библиотеку во время выполнения приложения. Опять же, предполагая, что корневой каталог нашей установки TBB записан в переменную `TBB_ROOT`, мы можем сделать это следующим образом:

```
export LD_LIBRARY_PATH=${TBB_ROOT}/lib/${ARCH}/${GCC_LIB_VERSION}:${LD_LIBRARY_PATH}
```

Откомпилировав приложение компилятором Intel, gcc или clang++ и установив переменную среды `LD_LIBRARY_PATH`, мы можем запустить приложение из командной строки:

```
./fig_1_04
```

В результате должно быть напечатано что-то вроде

```
Hello
Parallel STL!
```

Более полный пример

В предыдущих разделах описаны шаги написания, сборки и выполнения простых приложений TBB и Parallel STL, которые печатают две строки текста. В этом разделе мы напишем программу побольше, в которой воспользуемся всеми тремя высокоуровневыми параллельными интерфейсами, показанными на рис. 1.2. Мы не объясняем все детали алгоритмов и средства, использованные при создании примера, а просто хотим показать в деле различные уровни параллелизма, выразимые с помощью TBB. Пример намеренно искусственный. Он достаточно прост, чтобы объяснения уместились в нескольких абзацах, но довольно сложен, чтобы продемонстрировать описанные выше уровни параллелизма. Окончательный код следует рассматривать как иллюстрацию синтаксиса, а не как руководство по написанию оптимального TBB-приложения. В следующих главах мы подробно рассмотрим все средства, встречающиеся в этом примере, и расскажем, как их использовать для достижения высокой производительности в реальных приложениях.

Начинаем с последовательной реализации

Начнем с последовательной реализации, показанной на рис. 1.7. В этом примере к каждому изображению, хранящемуся в векторе, применяется гамма-коррекция и подкраска, а затем результат записывается в файл. Выделенная функция `fig_1_7` содержит цикл for, в котором обрабатываются элементы вектора путем вызова функций `applyGamma`, `applyTint` и `writeImage` для каждого изображения. На рис. 1.7 показаны также последовательные реализации каждой из этих функций. Определения представления изображений и некоторых вспомогательных функций содержатся в файле `ch01.h`. Этот заголовочный файл, как и весь остальной код примера, имеется по адресу https://github.com/Apress/threading-building-blocks.

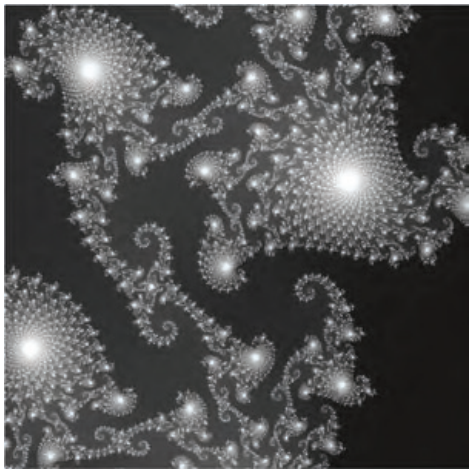
```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include "ch01.h"

using ImagePtr = std::shared_ptr<ch01::Image>;

ImagePtr applyGamma(ImagePtr image_ptr, double gamma);
ImagePtr applyTint(ImagePtr image_ptr, const double *tints);
void writeImage(ImagePtr image_ptr);

void fig_1_7(const std::vector<ImagePtr>& image_vector) {
  const double tint_array[] = {0.75, 0, 0};
  for (ImagePtr img : image_vector) {
    img = applyGamma(img, 1.4);
    img = applyTint(img, tint_array);
    writeImage(img);
  }
}

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
      double res = pow(v, gamma);
      if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
      out_rows[i][j] = ch01::Image::Pixel(res, res, res);
```

Рис. 1.7 ❖ Последовательная реализация примера, в котором к вектору изображений применяется гамма-коррекция и подкраска

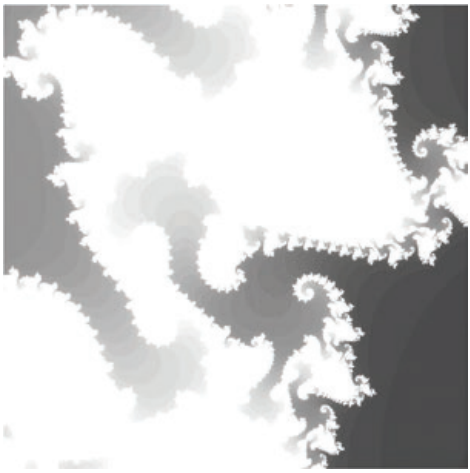
```
    }
  }
  return output_image_ptr;
}

ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      std::uint8_t b = (double)p.bgra[0] +
                       (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
      std::uint8_t g = (double)p.bgra[1] +
                       (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
      std::uint8_t r = (double)p.bgra[2] +
                       (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
      out_rows[i][j] =
        ch01::Image::Pixel(
          (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
          (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
          (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
      );
    }
  }
  return output_image_ptr;
}

void writeImage(ImagePtr image_ptr) {
  image_ptr->write( (image_ptr->name() + ".bmp").c_str());
}

int main(int argc, char* argv[]) {
  std::vector<ImagePtr> image_vector;

  for ( int i = 2000; i < 20000000; i *= 10 )
    image_vector.push_back(ch01::makeFractalImage(i));

  tbb::tick_count t0 = tbb::tick_count::now();
  fig_1_7(image_vector);
  std::cout << "Time : " << (tbb::tick_count::now()-t0).seconds()
            << " seconds" << std::endl;
  return 0;
}
```

Рис. 1.7 (продолжение)

Функции `applyGamma` и `applyTint` перебирают строки изображения во внешнем цикле for и пиксели одной строки – во внутреннем цикле for. Вычисляются и записываются в выходное изображение новые значения пикселей. Функция `applyGamma` применяет гамма-коррекцию, а функция `applyTint` подкрашивает изображение синим цветом. Обе функции получают и возвращают объекты типа `std::shared_ptr`, чтобы упростить управление памятью. Читатели, незнакомые с типом `std::shared_ptr`, могут обратиться к врезке «Замечание об интеллектуальных указателях». На рис. 1.8 показаны примеры входного и выходного изображений.



(a) Исходное (i == 2 000 000)



(b) После гамма-коррекции

(c) После гамма-коррекции и подкраски

Рис. 1.8 ❖ Примеры изображений:
(a) исходное изображение, сгенерированное функцией `ch01::makeFractalImage(2000000)`;
(b) изображение после гамма-коррекции;
(c) изображение после гамма-коррекции и подкраски

Замечание об интеллектуальных указателях

Одна из самых трудных сторон программирования на C/C++ – динамическое управление памятью. Каждому вызову `new` (`malloc`) должен соответствовать вызов `delete` (`free`), чтобы не было утечек памяти и двойного освобождения. В стандарт C++11 включены интеллектуальные указатели `unique_ptr`, `shared_ptr` и `weak_ptr`, поддерживающие автоматическое управление памятью, безопасное относительно исключений. Например, если память для объекта выделена вызовом функции `make_shared`, то мы получаем интеллектуальный указатель на объект. Когда мы присваиваем этот разделяемый указатель другому разделяемому указателю, библиотека C++ производит подсчет ссылок. Если на объект не ссылается ни один интеллектуальный указатель, то объект автоматически освобождается. В большинстве примеров в этой книге, в т. ч. на рис. 1.7, мы используем не простые, а интеллектуальные указатели. Это позволяет не думать обо всех местах, где надо было бы вставить вызов `free` или `delete`, – мы знаем, что интеллектуальные указатели свою работу выполнят.

Добавление уровня обмена сообщениями с помощью потокового графа

Применив нисходящий подход, мы можем заменить внешний цикл в функции `fig_1_07` на рис. 1.7 потоковым графом TBB, в котором изображения проходят через ряд фильтров, как показано на рис. 1.9. Мы признаем, что это самое неестественное из всех принятых в этом примере решений. В этом случае легко было бы использовать параллельный цикл или объединить вложенные циклы гамма-коррекции и подкраски. Но для демонстрации воспользуемся графом отдельных узлов, чтобы показать, как TBB можно использовать для выражения высокоуровневого параллелизма путем обмена сообщениями. В главе 3 мы изучим интерфейсы потокового графа TBB и познакомимся с более естественными применениями обмена сообщениями.

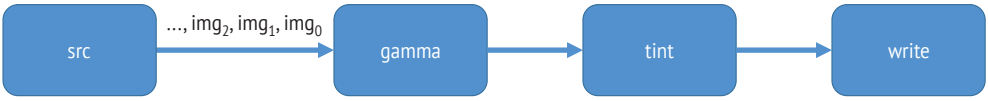


Рис. 1.9 ❖ Граф потока данных с четырьмя узлами:
(1) узел, который получает или генерирует изображения;
(2) узел, который применяет гамма-коррекцию; (3) узел, который применяет подкраску;
(4) узел, который записывает результирующее изображение

С помощью графа потока данных на рис. 1.9 мы можем организовать перекрытие различных этапов конвейера, применяемого к изображениям. Например, когда обработка первого изображения img_0 в узле `gamma` заканчивается, результат передается узлу `tint`, а в узел `gamma` поступает новое изображение img_1. Аналогично после следующего этапа img_0, прошедшее узлы `gamma` и `tint`, передается узлу `write`. Тем временем img_1 передается узлу `tint`, а в узле `gamma` начинается обработка нового изображения img_2. На каждом шаге выполнение фильтров независимо друг друга, поэтому вычисления можно разнести по раз-

ным ядрам или потокам. На рис. 1.10 показан цикл из функции fig_1_7, выраженный в виде потокового графа TBB.

```
void fig_1_10(const std::vector<ImagePtr>& image_vector) {
  const double tint_array[] = {0.75, 0, 0};

  tbb::flow::graph g;

  int i = 0;
  tbb::flow::source_node<ImagePtr> src(g,
    [&i, &image_vector] (ImagePtr &out) -> bool {
      if ( i < image_vector.size() ) {
        out = image_vector[i++];
        return true;
      } else {
        return false;
      }
    }, false);

  tbb::flow::function_node<ImagePtr, ImagePtr> gamma(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) -> ImagePtr {
      return applyGamma(img, 1.4);
    }
  );

  tbb::flow::function_node<ImagePtr, ImagePtr> tint(g,
    tbb::flow::unlimited,
    [tint_array] (ImagePtr img) -> ImagePtr {
      return applyTint(img, tint_array);
    }
  );

  tbb::flow::function_node<ImagePtr> write(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) {
      writeImage(img);
    }
  );

  tbb::flow::make_edge(src, gamma);
  tbb::flow::make_edge(gamma, tint);
  tbb::flow::make_edge(tint, write);
  src.activate();
  g.wait_for_all();
}
```

Рис. 1.10 ❖ Использование потокового графа TBB вместо внешнего цикла for

В главе 3 мы увидим, что построение и выполнение потокового графа TBB состоит из нескольких шагов. Сначала конструируется граф объектов g. Затем мы конструируем узлы, представляющие вычисления в графе потока данных. Узел, который поставляет изображения остальному графу, имеет тип source_

node и называется src. Вычисления выполняются объектами типа function_node, которые называются gamma, tint и write. Объект типа source_node можно рассматривать как узел, не получающий никаких входных данных и продолжающий отправлять данные, пока они не кончатся. Тип function_node – это обертка вокруг функции, которая получает выход и генерирует выход.

Создав узлы, мы соединяем их друг с другом ребрами. Ребра представляют зависимости, или коммуникационные каналы между узлами. В примере на рис. 1.10 мы хотим, чтобы узел src отправлял исходные изображения узлу gamma, поэтому соединяем узлы src и gamma ребром. Затем мы проводим ребро из узла gamma в узел tint. И аналогично проводим ребро из узла tint в узел write. Завершив построение структуры графа, мы вызываем функцию src.activate(), чтобы запустить объект типа source_node, и функцию g.wait_for_all(), которая ждет завершения работы графа.

При выполнении этого приложения каждое изображение, сгенерированное узлом src, проходит по конвейеру узлов, как описано выше. Когда изображение передается узлу gamma, библиотека TBB создает и планирует задачу, применяющую тело узла gamma к изображению. По завершении обработки выход поступит на вход узла tint. Точно так же TBB создаст и запланирует задачу для применения тела узла tint к выходу узла gamma. А когда эта обработка будет завершена, выход узла tint будет передан узлу write. И снова создается и планируется задача для выполнения тела узла – в данном случае записи изображения в файл. Всякий раз, как выполнение узла src завершается и он возвращает true, запускается новая задача для повторного выполнения тела src. И лишь когда узел src перестает генерировать новые изображения, а все уже сгенерированные изображения полностью обработаны узлом write, вызов wait_for_all возвращает управление.

Добавление уровня разветвления–соединения с помощью parallel_for

Теперь обратимся к реализации функций applyGamma и applyTint. На рис. 1.11 внешние циклы по i в последовательных реализациях заменены обращениями к tbb::parallel_for. Обобщенный параллельный алгоритм parallel_for параллельно обрабатывает разные строки. Он создает задачи, которые можно распределить по нескольким процессорным ядрам. Это пример уровня разветвления–соединения на рис. 1.3, который мы подробно опишем в главе 2.

```
ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];
```

Рис. 1.11 ❖ Добавление parallel_for для параллельного применения гамма-коррекции и подкраски

```
  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel &p = in_rows[i][j];
        double v = 0.3*p.bgra[2] + 0.59*p.bgra[l] + 0.11*p.bgra[0];
        double res = pow(v, gamma);
        if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
        out_rows[i][j] = ch01::Image::Pixel(res, res, res);
      }
    }
  );
  return output_image_ptr;
}
```

```
ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];
```

```
  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel& p = in_rows[i][j];
        std::uint8_t b = (double)p.bgra[0] +
                         (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
        std::uint8_t g = (double)p.bgra[1] +
                         (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
        std::uint8_t r = (double)p.bgra[2] +
                         (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
        out_rows[i][j] =
          ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
      }
    }
  }
  return output_image_ptr;
}
```

Рис. 1.11 ❖ Добавление `parallel_for` для параллельного применения гамма-коррекции и подкраски

Добавление уровня SIMD с помощью функции transform из Parallel STL

Мы можем продолжить оптимизацию двух наших вычислительных ядер, заменив внутренние циклы по `j` обращениями к функции `transform` из библиотеки Parallel STL. Алгоритм `transform` применяет функцию к каждому элемен-

ту из входного диапазона и сохраняет результаты в выходном диапазоне. Он получает следующие аргументы: (1) политика выполнения, (2 и 3) входной диапазон элементов, (4) начало выходного диапазона, (5) лямбда-выражение, применяемое к каждому элементу из входного диапазона, результат которого сохраняется в выходном элементе.

На рис. 1.12 задана политика выполнения `unseq`, сообщающая компилятору о необходимости использовать SIMD-версию функции `transform`. Функции из библиотеки Parallel STL подробно описаны в главе 4.

```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include <pstl/algorithm>
#include <pstl/execution>
#include "ch01.h"

using ImagePtr = std::shared_ptr<ch01::Image>;
void writeImage(ImagePtr image_ptr);

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      auto in_row = in_rows[i];
      auto out_row = out_rows[i];
      std::transform(pstl::execution::unseq, in_row, in_row+width,
        out_row, [gamma](const ch01::Image::Pixel &p) {
          double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
          assert(v > 0);
          double res = pow(v, gamma);
          if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
          return ch01::Image::Pixel(res, res, res);
      });
    }
  );
  return output_image_ptr;
}

ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
   std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
```

Рис. 1.12 ❖ Использование `std::transform` для добавления SIMD-параллелизма во внутренние циклы

```
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, tints](int i) {
      auto in_row = in_rows[i];
      auto out_row = out_rows[i];
      std::transform(pstl::execution::unseq, in_row, in_row+width,
        out_row, [tints](const ch01::Image::Pixel &p) {
          std::uint8_t b = (double)p.bgra[0] +
                           (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
          std::uint8_t g = (double)p.bgra[1] +
                           (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
          std::uint8_t r = (double)p.bgra[2] +
                           (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
          return ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
      });
    }
  );
  return output_image_ptr;
}
```

Рис. 1.12 (продолжение)

На рис. 1.12 каждый объект `Image::Pixel` содержит массив из четырех байтов, представляющих значения синего, зеленого, красного и альфа-каналов данного пикселя. Благодаря политике выполнения `unseq` используется векторизованный цикл для применения функции к элементам строки. Это пример уровня распараллеливания SIMD на рис. 1.3, когда мы пользуемся векторными устройствами в процессорном ядре для выполнения кода, но не распределяем вычисления по разным ядрам.

Примечание. Передача политики выполнения алгоритму Parallel STL еще не гарантирует, что выполнение будет распараллелено. Библиотека вправе использовать политику более ограничительную, чем запрошенная. Поэтому важно проверять эффект задания политики выполнения – особенно если он зависит от реализации компилятора!

Хотя примеры на рис. 1.7–1.12 несколько искусственные, они демонстрируют широту и мощь интерфейсов параллельного выполнения в библиотеке TBB. Одна и та же библиотека позволяет выразить параллелизм посредством обмена сообщениями, разветвления–соединения и SIMD, компонуя их в одном приложении.

Резюме

Эту главу мы начали с объяснения того, почему библиотека TBB даже более актуальна сейчас, чем 10 лет назад, когда она только появилась. Затем мы кратко рассмотрели основные средства библиотеки, включая интерфейсы параллельного выполнения и прочие вещи. Мы видели, что к числу высокоуровневых интерфейсов выполнения относятся обмен сообщениями, разветвление–соединение и SIMD, встречающиеся во многих параллельных приложениях. Затем мы рассказали, как получить копию TBB и проверить правильность настройки среды, для чего написали, откомпилировали и выполнили очень простые примеры. И завершили главу разработкой более полного примера, в котором используются все три высокоуровневых интерфейса выполнения.

Теперь мы готовы приступить к систематическому изучению поддержки параллельного программирования в следующих главах: обобщенные параллельные алгоритмы (глава 2), потоковые графы (глава 3), библиотека Parallel STL (глава 4), синхронизация (глава 5), конкурентные контейнеры (глава 6) и масштабируемое выделение памяти (глава 7).

Глава 2

Обобщенные параллельные алгоритмы

Как лучше всего планировать параллельные циклы? Как параллельно обрабатывать структуры данных, не поддерживающие итераторы произвольного доступа? Как лучше распараллелить приложения, похожие на конвейеры? Если бы библиотека TBB предоставляла только задачи и планировщик задач, на все эти вопросы нам пришлось бы отвечать самостоятельно. Но, к счастью, нам не нужно копаться в многочисленных магистерских и докторских диссертациях на эти темы. Разработчики TBB уже проделали за нас всю грязную работу! Они предоставили лучшие из известных методов решения описанных проблем: средства, известные под общим названием «обобщенные параллельные алгоритмы TBB», включающие шаблонные функции и шаблонные классы. Эти алгоритмы реализуют многие паттерны, считающиеся краеугольными камнями многопоточного программирования.

Примечание. Разработчики библиотеки TBB исторически использовали термин «обобщенные параллельные алгоритмы» для описания этого набора средств. Под *алгоритмом* они понимают не конкретное вычисление, например умножение матриц, LU-разложение или даже что-то типа `std::find`, а общие паттерны выполнения. Некоторые рецензенты этой книги возражали, что такого рода средства лучше бы называть более точно: *паттернами*, а не *алгоритмами*. Но мы решили остаться верными терминологии, применяемой в TBB уже много лет.

Мы, безусловно, отдаем предпочтение этим уже написанным алгоритмам (если они применимы), а не написанию своих собственных. Разработчики TBB потратили годы на их тестирование и оптимизацию производительности! Конечно, набор алгоритмов, включенных в TBB, не покрывает все возможные сценарии, но если уж какой-то паттерн обработки подходит, то его следует использовать. Предлагаемые TBB алгоритмы охватывают большую часть масштабируемого параллелизма в приложениях. В главе 8 мы обсудим паттерны проектирования для параллельного программирования, в частности описанные в книге Mattson, Sanders and Massingill «Patterns for Parallel Programming» (издательство Addison-Wesley), и способы их реализации с помощью обобщенных параллельных алгоритмов TBB.

Как показано на рис. 2.1, все обобщенные параллельные алгоритмы TBB начинаются с одного потока выполнения. Когда поток встречает параллель-

ный алгоритм, он распределяет связанную с этим алгоритмом работу по нескольким потокам. Когда все части работы будут сделаны, потоки соединяются, и выполнение продолжается в одном исходном потоке.

Рис. 2.1 ❖ Разветвление–соединение, характерное для параллельных алгоритмов TBB

Алгоритмы TBB предлагают мощную, но сравнительно простую для применения параллельную модель, поскольку зачастую их можно добавлять постепенно в относительно локальные участки кода. Мы можем поискать часть, в которой программа проводит большую часть времени, распараллелить ее, затем найти следующую часть и т. д.

Однако следует понимать, что алгоритмы TBB ***не*** гарантируют параллельности выполнения! Они лишь сообщают библиотеке, что параллельное выполнение разрешено. Если взглянуть на рис. 2.1 с точки зрения TBB, то он означает, что в выполнении частей вычисления могут участвовать все рабочие потоки, только часть потоков или вообще один лишь главный поток. О программах и библиотеках, допускающих факультативность параллелизма, в частности TBB, говорят, что они обладают *ослабленной последовательной семантикой*.

Говорят, что параллельная программа обладает *последовательной семантикой*, если ее выполнение в одном потоке не изменяет семантику программы. В этой книге мы неоднократно будем отмечать, что результаты последовательного и параллельного выполнения не всегда совпадают из-за ошибок округления и других источников неточности. Мы признаем эти потенциальные несемантические различия, используя термин *ослабленная последовательная семантика*. Модель TBB и OpenMP API предлагают ослабленную последовательную семантику, но есть и другие модели, например MPI, которые позволяют писать приложения с циклическими связями, в которых параллельное выполнение обязательно. Ослабленная последовательная семантика в TBB – важная часть, благодаря которой эта библиотека позволяет писать компонуемые приложения. С понятием композиции мы познакомились в главе 1 и вернемся к нему в главе 9. А пока просто запомните, что любой из описываемых в этой главе алгоритмов распределяет работу между одним или несколькими потоками, но необязательно всеми потоками, доступными в системе.

На рис. 2.2 показан набор алгоритмов, имеющихся в дистрибутиве TBB 2019 года. Все они находятся в пространстве имен `tbb` и становятся доступны после включения заголовочного файла `tbb.h`. Основные сведения об алгоритмах, выделенных **полужирным** шрифтом, приводятся в этой главе, а остальные описаны в последующих главах. Во врезке «Лямбда-выражения и поль-

зовательские классы» объясняется, что хотя мы чуть ли не во всех примерах используем лямбда-выражения для передачи исполняемого кода алгоритмам TBB, почти всегда их при желании можно заменить определенными пользователем объектами-функциями.

Категория	*Обобщенный алгоритм*	*Краткое описание*
Функциональный параллелизм	**parallel_invoke**	Параллельно вычисляет несколько функций
Простые циклы	**parallel_for**	Параллельно обходит диапазон значений
	parallel_reduce	Параллельно выполняет редукцию диапазона значений
	parallel_deterministic_reduce	Параллельно выполняет редукцию диапазона значений с детерминированным поведением разветвления–соединения
	parallel_scan	Параллельно вычисляет префикс по диапазону значений
	parallel_for_each	Параллельная реализация std::for_each. Подробно описан в главе 4
Сложные циклы	**parallel_do**	Параллельно обрабатывает рабочие задания в контейнере. Разрешается динамически добавлять новые задания
Конвейеры	pipeline	Класс для конвейерного выполнения фильтров. Описан в приложении B
	parallel_pipeline	Строго типизированная функция для конвейерного выполнения фильтров
Сортировка	parallel_sort	Функция для параллельной сортировки последовательности. Подробно описана в главе 4

Рис. 2.2 ❖ Обобщенные алгоритмы в библиотеке TBB. Полужирным шрифтом выделены алгоритмы, рассматриваемые в этой главе

Лямбда-выражения и пользовательские классы

Поскольку первая версия TBB написана еще до того, как в стандарт C++11 были включены лямбда-выражения, обобщенные алгоритмы TBB не требуют обязательного их использования. Иногда одинакового результата можно достичь как с помощью лямбда-выражений, так и с помощью объектов-функций (функторов). В других случаях алгоритм имеет два набора интерфейсов: более удобный, основанный на лямбда-выражениях, и менее удобный с применением пользовательских объектов.

Например, вместо

```
#include <vector>
#include <tbb/tbb.h>

void f(int v);

void sidebar_pfor_lambda(int N, const std::vector<int>& a) {
  tbb::parallel_for(0, N, 1,
    [&a](int i) {
        f(a[i]);
    }
  );
}
```

мы можем воспользоваться пользовательским классом:

```
class Body {
  const std::vector<int>& myVector;
public:
  Body(const std::vector<int>& v) : myVector{v} {}
  void operator()(int i) const {
    f(myVector[i]);
  }
};

void sidebar_pfor_functor(int N, const std::vector<int>& a) {
  tbb::parallel_for(0, N, 1, Body{a});
}
```

Часто выбор между лямбда-выражением и пользовательским классом – вопрос вкуса и ничего более.

Функциональный параллелизм на уровне задач

Пожалуй, самым простым алгоритмом в библиотеке TBB является `parallel_invoke`. Он позволяет параллельно выполнить две или более функций:

```
template<typename Func0, [...,] typename FuncN>
void parallel_invoke(const Func0& f0, [...,] const FuncN& fN);
```

Паттерн, соответствующий этой идее, называется *отображением* (map) – мы поговорим о нем в главе 8, когда будем обсуждать паттерны непосредственно. Независимость, выражаемая этим алгоритмом-паттерном, обеспечивает ему очень хорошую масштабируемость, поэтому в тех случаях, когда такой вид параллелизма применим, он является предпочтительным. Мы увидим также, что аналогичного эффекта позволяет добиться алгоритм `parallel_for`, поскольку тела цикла на разных итерациях могут быть независимы.

Полное описание интерфейсов `parallel_invoke` можно найти в приложении B. Если имеется набор функций, которые можно безопасно выполнить параллельно, используйте `parallel_invoke`. Например, два вектора, `v1` и `v2`, можно отсортировать, вызвав `serialQuicksort` сначала для первого вектора, а затем для второго:

```
serialQuicksort(serial_v1.begin(), serial_v1.end());
serialQuicksort(serial_v2.begin(), serial_v2.end());
```

Но поскольку эти вызовы независимы, можно вызвать и `parallel_invoke`, дав библиотеке TBB возможность создать задачи, которые будут выполняться разными рабочими потоками. Тогда оба вызова будут совмещены во времени (см. рис. 2.3).

```
#include <iostream>
#include <vector>
#include <limits>
#include <tbb/tbb.h>

struct DataItem {
  int id;
  double value;
  DataItem(int i, double v) : id{i}, value{v} {}
};

using QSVector = std::vector<DataItem>;

void serialQuicksort(QSVector::iterator b, QSVector::iterator e);

void fig_2_3(QSVector &v1, QSVector &v2) {
  tbb::parallel_invoke(
    [&v1]() { serialQuicksort(v1.begin(), v1.end()); },
    [&v2]() { serialQuicksort(v2.begin(), v2.end()); }
  );
}
```

Рис. 2.3 ❖ Использование `parallel_invoke` для параллельного вызова двух экземпляров `serialQuicksort`

Если оба вызова `serialQuicksort` занимают примерно одинаковое время и нет ограничений на ресурсы, то такая параллельная реализация завершится в два раза быстрее последовательной.

Примечание. Разработчик несет ответственность за то, что параллельное выполнение функций действительно безопасно. То есть TBB *не умеет* автоматически выявлять зависимости и применять синхронизацию, приватизацию или другие стратегии распараллеливания, чтобы сделать код безопасным. Мы отвечаем за это, решая применить `parallel_invoke` или любой другой параллельный алгоритм, обсуждаемый в этой главе.

Использование `parallel_invoke` не вызывает трудностей, но одиночный вызов этой функции еще не делает программу *масштабируемой*. Масштабируемый алгоритм эффективно использует дополнительные ядра и аппаратные ресурсы по мере их доступности.

Алгоритм называется *сильно масштабируемым*, если время решения задачи фиксированного размера уменьшается при увеличении количества процессорных ядер. Например, сильно масштабируемый алгоритм сможет завершить обработку имеющегося набора данных в два раза быстрее последовательного, если в наличии два ядра, и в 100 раз быстрее, если в наличии 100 ядер.

Алгоритм называется *слабо масштабируемым*, если при добавлении процессоров время обработки набора данных фиксированного размера *на одном процессоре* остается постоянным. Например, слабо масштабируемый алгоритм за одно и то же время сможет обработать в два раза больше данных, чем последовательный, если имеется два процессора, и в 100 раз больше данных, если имеется 100 процессоров.

Применение parallel_invoke для параллельной сортировки не демонстрирует ни сильной, ни слабой масштабируемости, потому что алгоритм способен использовать не более двух процессоров. Если имеется 100 процессоров, то 98 из них будут простаивать, поскольку мы не загрузили их работой. Вместо написания подобного кода следовало бы разработать масштабируемое приложение так, чтобы не нужно быть переделывать его всякий раз, как появляется новая архитектура с большим числом ядер.

По счастью, TBB умеет эффективно обрабатывать вложенный параллелизм (подробное описание см. в главе 9), поэтому мы можем обеспечить масштабируемое распараллеливание, включив parallel_invoke в рекурсивный алгоритм типа «разделяй и властвуй» (этот паттерн обсуждается в главе 8). В TBB имеются также обобщенные параллельные алгоритмы, рассматриваемые ниже в этой главе, реализующие паттерны, которые доказали свою эффективность для достижения масштабируемого распараллеливания, например циклы.

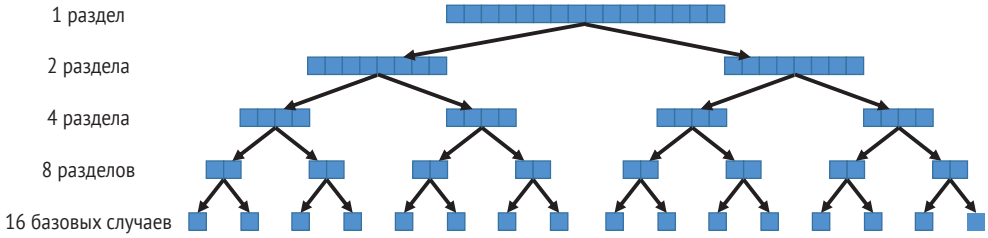
Чуть более сложный пример: параллельная реализация быстрой сортировки

Хорошо известным примером рекурсивного алгоритма типа «разделяй и властвуй» является быстрая сортировка (quicksort), представленная на рис. 2.4. Этот алгоритм рекурсивно обменивает данные по обе стороны от опорных значений, т. е. помещает элементы, меньшие или равные опорному значению, в левую часть массива, а элементы, большие опорного значения, – в правую часть. Когда в результате рекурсивного деления останутся только массивы размера 1, весь массив будет отсортирован.

Мы можем разработать параллельную реализацию быстрой сортировки, как показано на рис. 2.5, заменив два рекурсивных обращения к serialQuicksort вызовом parallel_invoke. Дополнительно можно еще ввести порог отсечения. В оригинальном последовательном алгоритме рекурсивное разбиение продолжается до тех пор, пока не останутся массивы из одного элемента.

Примечание. Запуск и планирование TBB-задачи обходятся не бесплатно – считается, что задача должна работать хотя бы 1 мкс или 10 000 тактов процессора, чтобы оправдать накладные расходы на ее создание и планирование. Мы поставим эксперименты в обоснование этого эвристического правила в главе 16.

Чтобы ограничить накладные расходы параллельной реализации, мы будем рекурсивно вызывать parallel_invoke, только когда на очередном шаге рекурсии сортируется массив, содержащий не менее 100 элементов, а в противном случае вызовем serialQuicksort.



(a) Рекурсивное разбиение, выполняемое алгоритмом быстрой сортировки

```
void serialQuicksort(QSVector::iterator b, QSVector::iterator e) {
  if (b >= e) return;

  // переставить
  double pivot_value = b->value;
  QSVector::iterator i = b, j = e-1;
  while (i != j) {
    while (i != j && pivot_value < j->value) --j;
    while (i != j && i->value <= pivot_value) ++i;
    std::iter_swap(i, j);
  }
  std::iter_swap(b, i);

  // рекурсивный вызов
  serialQuicksort(b, i);
  serialQuicksort(i+1, e);
}
```

(b) Исходный код последовательной реализации быстрой сортировки

Рис. 2.4 ❖ Последовательная реализация быстрой сортировки

```
void parallelQuicksort(QSVector::iterator b, QSVector::iterator e) {
  const int cutoff = 100;

  if (e - b < cutoff) {
    serialQuicksort(b, e);
  } else {
    // переставить
    double pivot_value = b->value;
    QSVector::iterator i = b, j = e - 1;
    while (i != j) {
      while (i != j && pivot_value < j->value) --j;
      while (i != j && i->value <= pivot_value) ++i;
      std::iter_swap(i, j);
    }
    std::iter_swap(b, i);

    // рекурсивный вызов
    tbb::parallel_invoke(
      [=]() { parallelQuicksort(b, i); },
      [=]() { parallelQuicksort(i + 1, e); }
    );
  }
}
```

Рис. 2.5 ❖ Параллельная реализация быстрой сортировки с помощью `parallel_invoke`

Вы, наверное, обратили внимание, что у параллельной реализации быстрой сортировки есть одно серьезное ограничение – перемещение элементов из одной части массива в другую производится последовательно. На верхнем уровне это означает, что в одном потоке выполняется операция со сложностью $O(n)$ еще до начала параллельной работы. Это ограничивает возможное ускорение. Оставляем интересующимся вопрос о том, как это ограничение снимается в известных реализациях параллельного разбиения (см. раздел «Дополнительная информация» в конце главы).

Циклы: parallel_for, parallel_reduce и parallel_scan

Для многих приложений общее время выполнения определяется прежде всего временем, проведенным в циклах. В TBB есть несколько алгоритмов для выражения параллельных циклов, которые позволяют быстро добавить масштабируемый параллелизм в важные циклы. В алгоритмах, отнесенных к категории «Простые циклы» на рис. 2.2, количество итераций можно легко определить в момент, когда цикл начинается.

Например, мы знаем, что в следующем цикле будет ровно `N` итераций, поэтому относим его к простым циклам:

```
for (int i = 0; i < N; ++i)
  f(a[i]);
```

Все алгоритмы простых циклов в TBB основаны на двух важных понятиях: *диапазон* и *тело*. Диапазон представляет рекурсивно разбиваемое множество значений. В случае цикла диапазон обычно описывается индексами в пространстве итераций или значениями, которые будет принимать итератор в процессе обхода контейнера. Тело – это функция, применяемая к каждому значению в диапазоне; в TBB тело обычно представлено лямбда-выражением, но может быть и объектом-функцией.

parallel_for: применение тела к каждому элементу диапазона

Начнем с небольшого последовательного цикла for, в котором функция применяется к элементу массива на каждой итерации:

```
for (int i = 0; i < N; ++i)
  f(a[i]);
```

Параллельную версию этого цикла можно создать, воспользовавшись алгоритмом `parallel_for`:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index last, [Index step,]
                  conts Func& f);
```

Полное описание интерфейсов `parallel_for` имеется в приложении B. В примере цикла ниже диапазоном является полуоткрытый интервал `[0, N)`, шаг равен 1, а тело – вызов `f(a[i])`.

```
#include <tbb/tbb.h>

void f(int v);

void fig_2_6(int N, const std::vector<int>& a) {
  tbb::parallel_for(0, N, 1, [a](int i) {
    f(a[i]);
  });
}
```

Рис. 2.6 ❖ Создание параллельного цикла с применением `parallel_for`

Когда TBB выполняет `parallel_for`, диапазон разбивается на участки итерирования. Каждый участок вместе с телом становится задачей, для которой планируется один из потоков, участвующих в выполнении алгоритма. Планированием задач занимается TBB, а нам остается только вызвать функцию `parallel_for`, выражающую тот факт, что итерации цикла следует выполнять параллельно. В последующих главах мы обсудим, как настроить поведение параллельных циклов в TBB. Пока же будем предполагать, что TBB порождает подходящее количество задач в соответствии с размером диапазона и количеством доступных ядер. В большинстве случаев это разумное предположение.

Важно понимать, что, используя `parallel_for`, мы гарантируем, что выполнять итерации цикла безопасно в любом порядке и параллельно. Библиотека TBB не проверяет, что параллельное выполнение итераций `parallel_for` (да и любого другого обобщенного алгоритма) дает те же результаты, что при последовательном выполнении, – эта ответственность возлагается на программиста. В главе 5 мы будем обсуждать механизмы синхронизации в TBB, позволяющие превратить небезопасный код определенного вида в безопасный. В главе 6 мы обсудим конкурентные контейнеры, предоставляющие потокобезопасные структуры данных, которые тоже иногда позволяют сделать код безопасным. Но в любом случае мы должны как-то гарантировать, что при использовании параллельного алгоритма любые потенциальные изменения порядка чтения и записи не отражаются на правильности результата. Кроме того, в параллельном коде следует использовать только потокобезопасные библиотеки и функции.

Например, следующий цикл ***небезопасно*** выполнять параллельно, поскольку каждая итерация зависит от результата предыдущей. Если изменить порядок итераций, то в массив `a` будут записаны другие элементы:

```
for (int i = 0; i < N; ++i)
  a[i] = a[i-1] + 1;
```

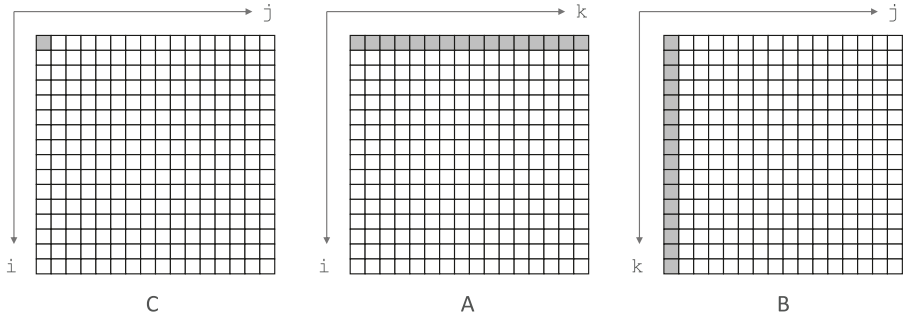
Рассмотрим массив `a={1,0,0,0,...,0}`. По завершении последовательного выполнения этого цикла массив будет содержать значения `{1,2,3,4,...,N}`. Если же итерации выполняются не по порядку, то результаты будут другими. Оценивая безопасность параллельного выполнения цикла, мысленно спросите себя, что будет, если все итерации выполнить одновременно, в случайном по-

рядке или в обратном порядке. Так, если a={1,0,0,0,...,0} и итерации выполняются в обратном порядке, то по завершении цикла массив будет содержать {1,2,1,1,...,1}. Очевидно, порядок в данном случае имеет значение.

Формальное описание анализа зависимости от данных выходит за рамки этой книги, но данный вопрос рассматривается во многих книгах по компиляторам и параллельному программированию, в т. ч. Michael Wolfe «High-Performance Compilers for Parallel Computing» (Pearson) и Allen and Kennedy «Optimizing Compilers for Modern Architectures» (Morgan Kaufmann). Кроме того, для поиска и исправления ошибок многопоточного программирования, в т. ч. в приложениях на основе TBB, можно воспользоваться такими инструментами, как Intel Inspector, входящий в состав Intel Parallel Studio XE.

Более сложный пример: параллельное умножение матриц

На рис. 2.7 показана неоптимизированная последовательная реализация умножения матриц с помощью вложенных циклов – вычисляется произведение C = AB матриц размера M×M. Этот пример служит только для демонстрации – если вам придется перемножать матрицы в реальном приложении и вы не считаете себя экспертом по оптимизации, то почти наверняка лучше воспользоваться хорошо оптимизированной реализацией из какой-нибудь математической библиотеки, включающей подпрограммы линейной алгебры из набора Basic Linear Algebra Subprograms (BLAS), например Intel Math Kernel Library (MKL), BLIS или ATLAS. Умножение матриц подходит нам в качестве примера, поскольку код небольшой и выполняет простую операцию, с которой все хорошо знакомы. А теперь взгляните на рис. 2.7.



(a) Элементы, перебираемые на первой итерации внутреннего цикла по k для вычисления одного элемента матрицы C

```
void fig_2_7(int M, double *a, double *b, double *c) {
  for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
      int c_index = i*M+j;
      for (int k = 0; k < M; ++k) {
        c[c_index] += a[i*M + k] * b[k*M+j];
      }
    }
  }
}
```

(b) Последовательная реализация

Рис. 2.7 ❖ Неоптимизированная реализация умножения матриц

Мы можем быстро реализовать параллельную версию умножения матриц, воспользовавшись алгоритмом parallel_for, как показано на рис. 2.8. В этой реализации внешний цикл по i распараллелен. На одной итерации внешнего цикла выполняются вложенные циклы по j и по k, поэтому если только M не очень мало, работы будет достаточно, чтобы не нарушать правило одной микросекунды. Для снижения накладных расходов часто разумнее распараллеливать внешние циклы.

```
void fig_2_8(int M, double *a, double *b, double *c) {
  tbb::parallel_for( 0, M, [=](int i) {
    for (int j = 0; j < M; ++j) {
      int c_index = i*M+j;
      for (int k = 0; k < M; ++k) {
        c[c_index] += a[i*M + k] * b[k*M+j];
      }
    }
  });
}
```

Рис. 2.8 ❖ Простой цикл parallel_for для реализации умножения матриц

Код на рис. 2.8 являет собой простую параллельную версию умножения матриц. Она корректна, но из-за способа обхода массивов оставляет много возможностей для повышения производительности. В главе 16 мы поговорим о дополнительных средствах parallel_for, позволяющих настраивать производительность.

parallel_reduce: вычисление одного результата для всего диапазона

Еще один распространенный паттерн – редукция, которую часто называют также паттерном «map-reduce», поскольку в нем обычно применяется и паттерн *отображения* (map) (подробнее о терминологии, связанной с паттернами, см. главу 8).

Редукция вычисляет одно значение по коллекции значений. Примерами могут служить вычисления суммы, минимума или максимума.

Рассмотрим цикл, в котором ищется максимальное значение в массиве:

```
int max_value = std::numeric_limits<int>::min();
for (int i = 0; i < a.size(); ++i) {
  max_value = std::max(max_value,a[i]);
}
```

Вычисление максимума множества – ассоциативная операция, т. е. мы можем применить ее к подмножествам, а затем объединить частичные результаты. Операция вычисления максимума также коммутативна, т. е. необязательно объединять результаты в каком-то определенном порядке.

Для циклов, выполняющих ассоциативные операции, TBB предлагает функцию parallel_reduce:

```
template<typename Range, typename Value,
         typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity,
                      const Func& func, const Reduction& reduction);
```

Полное описание интерфейсов `parallel_reduce` приведено в приложении В.

Многие стандартные математические операции ассоциативны, например сложение, умножение, вычисление максимума и минимума. Некоторые операции ассоциативны теоретически, но перестают быть таковыми в реальных системах из-за ограниченной точности представления чисел. Об этом следует помнить, оценивая корректность распараллеливания (см. врезку «Ассоциативность и типы с плавающей точкой»).

Ассоциативность и типы с плавающей точкой

В компьютерной арифметике не всегда возможно представить вещественные числа точно. Вместо этого используются приближенные представления – типы с плавающей точкой: `float`, `double` и `long double`. В результате математические свойства операций с вещественными числами перестают выполняться для чисел с плавающей точкой. Например, операция сложения вещественных чисел ассоциативна и коммутативна, но ни то, ни другое свойство не имеет место для чисел с плавающей точкой.

Так, при вычислении суммы `N` вещественных чисел, равных 1.0, должен бы получиться результат `N`.

```
float r = 0.0;
for (uint64_t i = 0; i < N; ++i)
  r += 1.0;
std::cout << "in-order sum == " << r << std::endl;
```

Но в представлении с плавающей точкой количество значащих цифр ограничено, поэтому не все вещественные числа представимы точно. Например, если выполнить этот цикл при `N == 10e6` (10 млн), то получится результат `10000000`. Но при `N == 20e6` результат будет `16777216`. Переменная `r` просто не может представить число `16777217`, потому что в типе `float` длина мантиссы (количество значащих цифр) равна 24, а для представления `16777217` нужно 25 бит. При прибавлении 1.0 результат округляется с недостатком до `16777216`, и каждое последующее прибавление 1.0 приводит к точно такому округлению. Будем справедливы – для одного шага результат `16777216` можно считать хорошим приближением `16777217`. Проблема возникает из-за накопления этих ошибок округления.

Если разбить суммирование на два цикла и объединить частичные результаты, то получится правильный результат:

```
float tmp1 = 0.0, tmp2 = 0.0;
for (uint64_t i = 0; i < N/2; ++i)
  tmp1 += 1.0;
for (uint64_t i = N/2; i < N; ++i)
  tmp2 += 1.0;
float r2 = tmp1 + tmp2;
std::cout << "associative sum == " << r2 << std::endl;
```

Почему? Потому что `r` может представить большие числа, правда, не всегда точно. Значения `tmp1` и `tmp2` по абсолютной величине близки, поэтому сложение влияет на имеющиеся в представлении значащие цифры, и мы получаем результат, хорошо аппроксимирующий

20 млн. Это крайний пример того, как неассоциативность может исказить результат вычисления, в котором участвуют числа с плавающей точкой.

Из этого обсуждения мы должны извлечь такой урок: при использовании `parallel_reduce` предполагается ассоциативность, только тогда можно объединять параллельно вычисленные частичные результаты. Поэтому при работе с числами с плавающей точкой результаты могут отличаться от полученных при последовательной реализации. Более того, в зависимости от количества привлеченных потоков `parallel_reduce` может при каждом прогоне создавать разное количество частичных результатов. Поэтому мы будем получать разные результаты при разных прогонах, даже если входные данные не изменяются.

Но прежде чем впадать в панику и навсегда отказываться от `parallel_reduce`, следует принять во внимание, что результат операций над числами с плавающей точкой почти всегда приближенный. Поэтому получение разных результатов для одних и тех же входных данных еще не означает, что хотя бы один результат неправильный. Просто ошибки округления накапливаются по-разному. И разработчику решать, существенны ли эти различия в конкретном приложении.

Если мы хотим гарантировать хотя бы получение одинаковых результатов для одних и тех же входных данных при каждом прогоне, то можем воспользоваться функцией `parallel_deterministic_reduce`, описанной в главе 16. Эта детерминированная реализация всегда создает одинаковое количество частичных результатов и объединяет их в одном и том же порядке, поэтому аппроксимация не зависит от прогона.

Как и для всех алгоритмов из категории простых циклов, при использовании `parallel_reduce` необходимо задать диапазон (`range`) и тело (`func`). Но еще нужно передать нейтральное значение (`identity`) и тело редукции (`reduction`).

Для распараллеливания с помощью `parallel_reduce` библиотека TBB разбивает диапазон на участки и создает задачи, применяющие `func` к каждому участку. В главе 16 мы обсудим, как использовать разбиватели для управления размером участков, а пока можем предполагать, что TBB создает участки такого размера, чтобы минимизировать накладные расходы и сбалансировать нагрузку. Каждая задача, выполняющая `func`, начинает со значения `init`, инициализированного значением `identity`, а затем вычисляет и возвращает частичный результат для своего участка. TBB объединяет частичные результаты, вызывая функцию `reduction`, которая вычисляет окончательный результат для всего цикла.

Аргумент `identity` равен нейтральному элементу относительно распараллеливаемой операции. Хорошо известно, что нейтральным элементом для сложения (аддитивным) является 0 (поскольку `x + 0 = x`), а нейтральным элементом для умножения (мультипликативным) – 1 (поскольку `x * 1 = x`). Функция `reduction` принимает два частичных результата и объединяет их.

На рис. 2.9 показано, как функции `func` и `reduction` применяются для вычисления максимума в массиве из 16 элементов, если диапазон разбит на четыре участка. В этом примере ассоциативная операция, применяемая функцией `func` к элементам массива, – `max()`, а нейтральный элемент равен -∞, поскольку `max(x,-∞)=x`. В C++ в качестве операции мы используем `std::max`, а -∞ представляем в программе значением `std::numeric_limits<int>::min()`.

На рис. 2.10 показано, как выразить простой цикл вычисления максимума с помощью `parallel_reduce`.

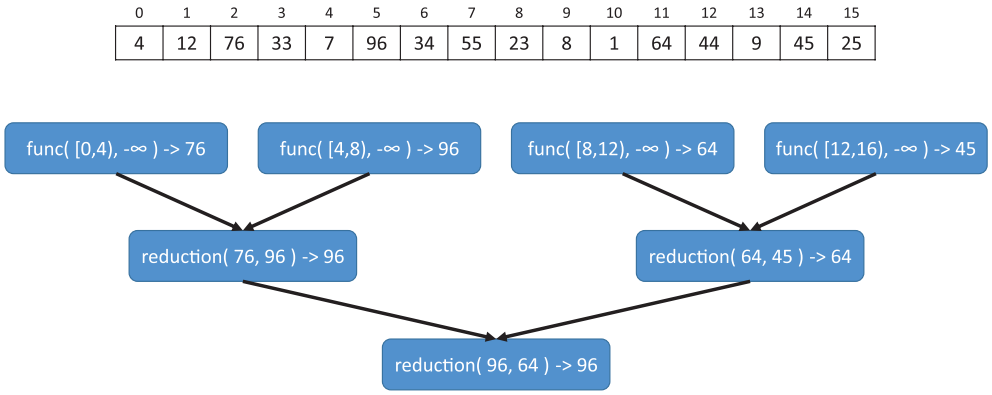


Рис. 2.9 ❖ Как функции `func` и `reduction` вызываются для вычисления максимума

```
int fig_2_10(const std::vector<int> &a) {
  int max_value = tbb::parallel_reduce(
    /* the range = */ tbb::blocked_range<int>(0, a.size()),
    /* identity = */ std::numeric_limits<int>::min(),
    /* func = */
    [&](const tbb::blocked_range<int> &r, int init) -> int {
      for (int i = r.begin(); i != r.end(); ++i) {
        init = std::max(init, a[i]);
      }
      return init;
    },
    /* reduction = */
    [](int x, int y) -> int {
      return std::max(x,y);
    }
  );
  return max_value;
}
```

Рис. 2.10 ❖ Использование `parallel_reduce` для вычисления максимума

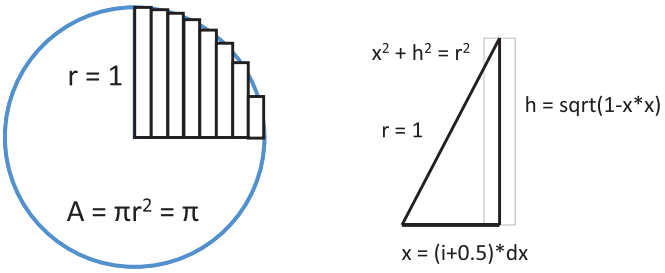
Вы, возможно, заметили, что на рис. 2.10 диапазон представлен объектом `blocked_range`, а не просто своим началом и концом, как в случае `parallel_for`. Для алгоритма `parallel_for` предлагается упрощенный синтаксис, отсутствующий для `parallel_reduce`. В случае `parallel_reduce` мы должны передать объект диапазона непосредственно, но, к счастью, можем использовать один из готовых диапазонов, предоставляемых библиотекой, в т. ч. `blocked_range`, `blocked_range2d` и `blocked_range3d`. Эти объекты будут описаны в главе 16, а их полные интерфейсы см. в приложении В.

Тип `blocked_range` представляет одномерное пространство итераций. Для его конструирования мы задаем начальное и конечное значения. В теле функции `begin()` и `end()` используются для получения начала и конца участка, назначенного данному выполнению тела, а затем производится обход этого поддиапа-

зона. На рис. 2.8 каждое отдельное значение из диапазона передавалось телу `parallel_for`, поэтому не было необходимости в цикле по `i` для обхода диапазона. На рис. 2.10 тело получает объект `blocked_range`, который представляет подлежащий обходу участок, поэтому необходим цикл по `i` для обхода этого участка.

Более сложный пример: вычисление π с помощью численного интегрирования

На рис. 2.11 иллюстрируется подход к вычислению числа π с помощью численного интегрирования. Высота каждого треугольника определяется по теореме Пифагора. Площадь одного квадранта круга вычисляется в цикле. Результат умножается на 4 – получается полная площадь круга, равная π.



(a) Применение численного интегрирования для вычисления π

```
#include <cmath>

double fig_2_11(int num_intervals) {
  double dx = 1.0 / num_intervals;
  double sum = 0.0;
  for (int i = 0; i < num_intervals; ++i) {
    double x = (i+0.5)*dx;
    double h = std::sqrt(1-x*x);
    sum += h*dx;
  }
  double pi = 4 * sum;
  return pi;
}
```

(b) Последовательная реализация

Рис. 2.11 ❖ Последовательное вычисление π с помощью численного интегрирования методом прямоугольников

Код на рис. 2.11 вычисляет сумму площадей всех прямоугольников – это операция редукции. Для применения `parallel_reduce` нужно задать диапазон, тело, нейтральное значение и функцию редукции. В этом примере диапазон равен `[0, num_intervals)`, а тело будет напоминать цикл по `i` на рис. 2.11. Нейтральное значение равно 0.0, поскольку мы вычисляем сумму. А функция редукции, объединяющая частичные результаты, возвращает сумму двух значений. Параллельная реализация с помощью функции `parallel_reduce` показана на рис. 2.12.

```
#include <cmath>
#include <tbb/tbb.h>

double fig_2_12(int num_intervals) {
  double dx = 1.0 / num_intervals;
  double sum = tbb::parallel_reduce(
    /* range = */ tbb::blocked_range<int>(0, num_intervals),
    /* idenity = */ 0.0,
    /* func */
    [=](const tbb::blocked_range<int> &r, double init)
    -> double {
      for (int i = r.begin(); i != r.end(); ++i) {
        double x = (i + 0.5)*dx;
        double h = std::sqrt(1 - x*x);
        init += h*dx;
      }
      return init;
    },
    /* reduction */
    [](double x, double y) -> double {
      return x + y;
    }
  );
  double pi = 4 * sum;
  return pi;
}
```

Рис. 2.12 ❖ Вычисление π с помощью `tbb::parallel_reduce`

Как и для `parallel_for`, существуют дополнительные возможности и параметры, позволяющие настроить поведение `parallel_reduce` и управлять ошибками округления (см. «Ассоциативность и типы с плавающей точкой»). Они рассматриваются в главе 16.

parallel_scan: редукция с промежуточными значениями

Не столь распространенный, но все-таки важный паттерн, встречающийся в приложениях, – сканирование (иногда называется префиксом). Сканирование похоже на редукцию, но вычисляет не только единственное значение по коллекции, но и промежуточный результат для каждого элемента в диапазоне (*префиксы*). Пример – суммирование с нарастающим итогом значений x_0, x_1, ..., x_N. Результат состоит из всех промежуточных сумм y_0, y_1, ..., y_N и конечной суммы y_N.

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$\ldots$$
$$y_N = x_0 + x_1 + \ldots + x_N$$

Вот как в последовательном цикле вычисляется сумма с нарастающим итогом по вектору `v`:

```
int serialImpl(const std::vector<int> &v,
               std::vector<int> &rsum) {
  int N = v.size();
  rsum[0] = v[0];
  for (int i = 1; i < N; ++i) {
    rsum[i] = rsum[i-1] + v[i];
  }
  int final_sum = rsum[N-1];
  return final_sum;
}
```

На первый взгляд, сканирование выглядит как принципиально последовательный алгоритм. Каждый префикс зависит от результатов, вычисленных на всех предыдущих итерациях. Но, как ни удивительно, существуют эффективные параллельные реализации этого алгоритма, и одна из них – функция `parallel_scan` из библиотеки TBB. Ей требуется передать диапазон, нейтральный элемент, тело сканирования и тело объединения:

```
template<typename Range, typename Value,
         typename Scan, typename Combine>
Value parallel_scan( const Range& range, const Value& identity,
                     const Scan& scan, const Combine& combine);
```

Диапазон, нейтральный элемент и тело объединения аналогичны диапазону, нейтральному элементу и телу редукции в алгоритме `parallel_reduce`. Как и в других алгоритмах из категории простых циклов, TBB разбивает диапазон на участки и создает задачи для применения тела сканирования к этим участкам. Полное описание интерфейса `parallel_scan` см. в приложении B.

Отличительной чертой `parallel_scan` является тот факт, что тело сканирования может быть выполнено более одного раза для одного участка – сначала в режиме *предварительного сканирования*, а затем в режиме *окончательного сканирования*.

В режиме окончательного сканирования телу передается точный префикс для итерации, непосредственно предшествующей поддиапазону, связанному с этим телом. Зная это значение, тело вычисляет и сохраняет префиксы для каждой итерации в своем поддиапазоне и возвращает точный префикс для последнего элемента в этом поддиапазоне.

Но когда тело вычисляется в режиме предварительного сканирования, оно получает начальное значение префикса, не являющееся окончательным значением для элемента, предшествующего ассоциированному диапазону. Как и `parallel_reduce`, алгоритм `parallel_scan` нуждается в ассоциативности. В режиме предварительного сканирования начальный префикс может представлять предшествующий ему поддиапазон, но не полный предшествующий диапазон. Зная это значение, тело возвращает (еще не окончательный) префикс для последнего элемента в своем поддиапазоне. Возвращенное значение представляет частичный результат для комбинации начального префикса с поддиапазоном. С помощью режимов предварительного и окончательного сканирования можно задействовать параллелизм, присутствующий в алгоритме сканирования.

Как это работает?

Снова рассмотрим пример суммирования с нарастающим итогом и вычисления префикса при разбиении диапазона на три участка A, B и C. В последовательной реализации мы сначала вычисляем все префиксы для A, затем для B и потом для C (эти три шага выполняются в указанном порядке). Параллельное сканирование позволяет поступить более эффективно, как показано на рис. 2.13.

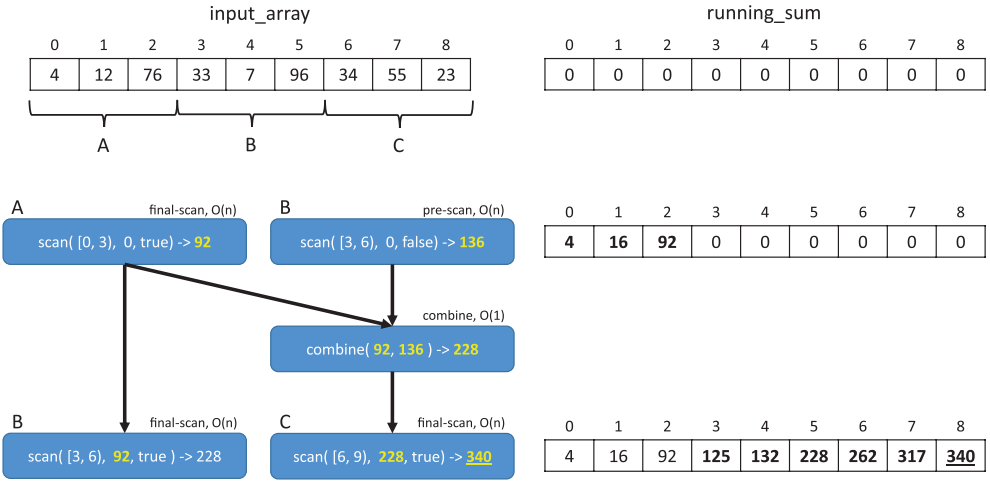


Рис. 2.13 ❖ Параллельное выполнение сканирования при вычислении суммы

Сначала мы вычисляем префикс для A в режиме окончательного сканирования, поскольку это первое множество значений, и, значит, префикс будет вычислен точно, если передать телу нейтральный элемент в качестве начального значения. Одновременно с A мы начинаем обработку B в режиме предварительного сканирования. После того как оба сканирования завершены, можно вычислить точные начальные префиксы для B и C. Для B мы передаем конечный результат, вычисленный для A (92), а для C – конечный результат сканирования A, объединенный с результатом предварительного сканирования B (92+136 = 228).

Операция объединения занимает постоянное время, поэтому обходится значительно дешевле операций сканирования. В отличие от последовательной реализации, которая требует трех больших шагов, выполняемых друг за другом, в параллельной реализации одновременно выполняется окончательное сканирование A и предварительное сканирование B, затем операция объединения, требующая постоянного времени, и, наконец, окончательное параллельное сканирование B и C. Если имеется два ядра и `N` достаточно велико, то для параллельного вычисления префикса понадобится примерно на треть меньше времени, чем для последовательного. И конечно, при выполнении `parallel_scan` количество участков может быть больше, чтобы задействовать больше ядер.

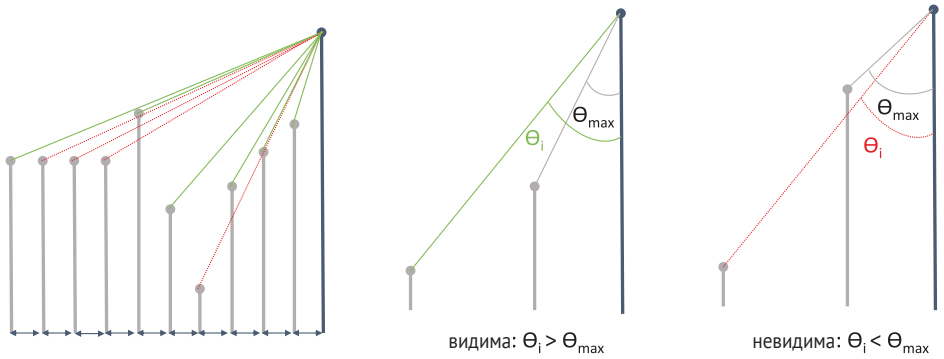
На рис. 2.14 показана реализация вычисления частичных сумм с помощью `parallel_scan`. В качестве `range` передается интервал `[1, N)`, в качестве `identity` – 0, а функция `combine` возвращает сумму двух своих аргументов. Тело сканирования возвращает частичную сумму всех значений в своем поддиапазоне, сложенную с полученной начальной суммой. Однако запись префикса в массив `running_sum` производится, только если аргумент `is_final_scan` равен `true`.

```
int fig_2_14(const std::vector<int> &v, std::vector<int> &rsum) {
  int N = v.size();
  rsum[0] = v[0];
  int final_sum = tbb::parallel_scan(
    /* range = */ tbb::blocked_range<int>(1, N),
    /* identity = */ (int)0,
    /* scan body */
    [&v, &rsum](const tbb::blocked_range<int> &r,
                int sum, bool is_final_scan) -> int {
      for (int i = r.begin(); i < r.end(); ++i) {
        sum += v[i];
        if (is_final_scan)
          rsum[i] = sum;
      }
      return sum;
    },
    /* combine body */
    [](int x, int y) {
      return x + y;
    }
  );
  return final_sum;
}
```

Рис. 2.14 ❖ Реализация суммирования с нарастающим итогом с помощью `parallel_scan`

Более сложный пример: линия прямой видимости

На рис. 2.15 показана последовательная реализация задачи о линии прямой видимости, аналогичной той, что описана в книге Guy E. Blelloch «Vector Models for Data-Parallel Computing» (издательство MIT Press). Пусть даны высота точки наблюдения и высоты точек, отстоящих на фиксированные расстояния от точки наблюдения. Тогда линия прямой видимости определяет, какие точки видны из точки наблюдения. Как показано на рис. 2.15, точка не видна, если между ней и точкой наблюдения `altitude[0]` существует точка с большим углом Θ. При последовательной реализации мы должны выполнить сканирование для вычисления максимума Θ по всем точкам между данной и точкой наблюдения. Если угол Θ для данной точки больше максимального угла, то эта точка видима, иначе невидима.



(a) Вычисление точек, видимых из точки наблюдения

```
void fig_2_15(const std::vector<double> &altitude,
              std::vector<bool> &is_visible, double dx) {
  const int N = altitude.size();

  double max_angle = std::atan2(dx, altitude[0] - altitude[1]);
  double my_angle = 0.0;

  for (int i = 2; i < N; ++i ) {
    my_angle = std::atan2(i * dx, altitude[0] - altitude[i]);
    if (my_angle >= max_angle) {
      max_angle = my_angle;
    } else {
      is_visible[i] = false;
    }
  }
}
```

(b) Последовательная реализация

Рис. 2.15 ❖ Линия прямой видимости

На рис. 2.16 показана параллельная реализация задачи о линии прямой видимости с использованием алгоритма `parallel_scan`. По завершении алгоритма массив `is_visible` будет содержать признаки видимости каждой точки (`true` или `false`). Важно отметить, что код на рис. 2.16 должен вычислять максимальный угол в каждой точке, чтобы определить видимость одной точки, но окончательный результат содержит признаки видимости точек, а не углы. Поскольку `max_angle` нужен, но не является окончательным результатом, он вычисляется в обоих режимах – предварительного и окончательного сканирования, – но массив `is_visible` заполняется только в режиме окончательного сканирования.

```
void fig_2_16(const std::vector<double> &altitude,
              std::vector<bool> &is_visible, double dx) {
  const int N = altitude.size();
  double max_angle = std::atan2(dx, altitude[0] - altitude[1]);

  double final_max_angle = tbb::parallel_scan(
    /* range = */ tbb::blocked_range<int>(1, N),
    /* identity */ 0.0,
    /* scan body */
    [&altitude, &is_visible, dx](const tbb::blocked_range<int> &r,
                                 double max_angle,
                                 bool is_final_scan) -> double {
      for (int i = r.begin(); i != r.end(); ++i) {
        double my_angle = atan2(i*dx, altitude[0] - altitude[i]);
        if (my_angle >= max_angle)
          max_angle = my_angle;
        if (is_final_scan && my_angle < max_angle)
          is_visible[i] = false;
      }
      return max_angle;
    },
    [](double a, double b) {
      return std::max(a,b);
    }
  );
}
```

Рис. 2.16 ❖ Реализация линии прямой видимости с помощью `parallel_scan`

Варить до готовности: parallel_do и parallel_pipeline

В некоторых приложениях простых циклов достаточно для полезного распараллеливания. Но в других случаях мы должны выражать параллелизм в циклах, для которых диапазон невозможно вычислить до начала цикла. Рассмотрим, к примеру, цикл while:

```
while(auto i = get_image()) {
  f(i);
}
```

Этот цикл продолжает читать изображения, пока они не кончатся. Каждое прочитанное изображение обрабатывается функцией `f`. Использовать `parallel_for` нельзя, потому что мы не знаем, сколько всего изображений будет, и, значит, не можем подготовить диапазон.

Более тонкая ситуация возникает, когда имеется контейнер, не предоставляющий итераторов произвольного доступа:

```
std::list<image_type> my_images = get_image_list();
for (auto &i : my_list) {
  f(i);
}
```

Примечание. В C++ итератором называется объект, который указывает на элемент в некотором диапазоне и предоставляет операторы для обхода элементов диапазона. Существуют различные категории итераторов: однонаправленные, двунаправленные и произвольного доступа. Оператор произвольного доступа можно переместить к любому элементу за постоянное время.

Поскольку список `std::list` не поддерживает произвольный доступ к своим элементам, мы можем получить ограничители диапазона `my_images.begin()` и `my_images.end()`, но не можем добраться до элементов между ними, не обойдя список последовательно. Поэтому TBB не может быстро (за постоянное время) создать участки итерирования и распределить их между задачами, поскольку не в состоянии быстро найти концевые точки этих участков.

Для работы с такими сложными циклами библиотека TBB предлагает два обобщенных алгоритма: `parallel_do` и `parallel_pipeline`.

parallel_do: применять тело, пока имеются элементы

Алгоритм `parallel_do` применяет тело к элементам, пока они не кончатся. Часть элементов можно подать заранее в момент начала цикла, а остальные добавляются при исполнении тела во время обработки им других элементов.

У алгоритма `parallel_do` есть два интерфейса: один принимает итераторы на начало и на конец, другой – контейнер. Полное описание интерфейсов `parallel_do` приведено в приложении B. В этом разделе мы рассмотрим версию, принимающую контейнер:

```
template<typename Container, typename Body>
void parallel_do(Container c, Body body);
```

Начнем с простого примера. Пусть имеется список `std::list` элементов `std::pair<int, bool>`, каждый из которых содержит случайное целое число и `false`. Для каждого элемента мы вычисляем, является ли число простым; если да, в булево значение записывается `true`. Предполагается, что заданы функции, которые добавляют новый элемент в контейнер и определяют простоту числа. Ниже показана последовательная реализация:

```
using PrimesValue = std::pair<int, bool>;
using PrimesList = std::list<PrimesValue>;

bool isPrime(int n);

void serialImpl(PrimesList &values) {
  for (PrimesList::reference v : values) {
    if (isPrime(v.first))
      v.second = true;
  }
}
```

Параллельную реализацию этого цикла можно написать с помощью функции `parallel_do`, как показано на рис. 2.17.

```
void fig_2_17(PrimesList &values) {
  tbb::parallel_do(values,
    [](PrimesList::reference v) {
      if (isPrime(v.first))
        v.second = true;
    }
  );
}
```

Рис. 2.17 ❖ Реализация цикла нахождения простых чисел с помощью `parallel_do`

Алгоритм `parallel_do` из TBB безопасно обходит контейнер последовательно и создает задачи для применения тела к каждому элементу. Поскольку контейнер приходится обходить последовательно, степень масштабируемости `parallel_do` не такая высокая, как у `parallel_for`, но если тело достаточно большое (выполняется дольше 100 000 тактов), то накладные расходы на обход пренебрежимо малы по сравнению с параллельным выполнением тела для элементов.

Помимо работы с контейнерами, не предоставляющими произвольного доступа, `parallel_do` позволяет добавлять новые элементы в теле функции. Если тела выполняются параллельно и действительно добавляют новые элементы, то добавление тоже можно организовать параллельно, снимая ограничение `parallel_do` на последовательное порождение задач. На рис. 2.18 приведена последовательная реализация, в которой вычисляется простота числа, но значения хранятся не в списке, а в дереве.

```
using PrimesValue = std::pair<int, bool>;

struct PrimesTreeElement {
  using Ptr = std::shared_ptr<PrimesTreeElement>;

  PrimesValue v;
  Ptr left;
  Ptr right;
  PrimesTreeElement(const PrimesValue &_v);
}

bool isPrime(int n);

void serialCheckPrimesElem(PrimesTreeElement::Ptr e) {
  if (e) {
    if (isPrime(e->v.first))
      e->v.second = true;
    if (e->left) serialCheckPrimesElem(e->left);
    if (e->right) serialCheckPrimesElem(e->right);
  }
}
```

Рис. 2.18 ❖ Проверка простоты чисел в дереве

Параллельная реализация этой версии с применением `parallel_do` приведена на рис. 2.19. Чтобы наглядно продемонстрировать различные способы порождения элементов, здесь используется контейнер, содержащий одно дерево значений. Выполнение `parallel_do` начинается, когда в контейнере находится один элемент, но при каждом выполнении добавляется еще два элемента: один в левое поддерево, другой в правое. Для добавления элементов в пространство итерирования мы используем метод `parallel_do_feeder.add`. Класс `parallel_do_feeder` определен библиотекой TBB и передается телу в качестве второго аргумента.

```
template<typename Item>
struct parallel_do_feeder {
  void add( const Item& item );
  // поддерживается начиная с C++11
  void add( Item&& item );
};
```

Количество доступных для обработки элементов экспоненциально возрастает по мере спуска по дереву. На рис. 2.19 мы добавляем новые элементы еще до того, как проверили текущий элемент на простоту, так чтобы другие задачи запускались как можно быстрее.

```
using PrimesValue = std::pair<int, bool>;

struct PrimesTreeElement {
  using Ptr = std::shared_ptr<PrimesTreeElement>;

  PrimesValue v;
  Ptr left;
  Ptr right;
  PrimesTreeElement(const PrimesValue &_v) : left{}, right{} {
    v.first = _v.first;
    v.second = _v.second;
  }
};

bool isPrime(int n);

void fig_2_19(PrimesTreeElement::Ptr root) {
  PrimesTreeElement::Ptr tree_array[] = {root};
  tbb::parallel_do(tree_array,
    [](PrimesTreeElement::Ptr e,
      tbb::parallel_do_feeder<PrimesTreeElement::Ptr>& feeder)
{
        if (e) {
          if (e->left) feeder.add(e->left);
          if (e->right) feeder.add(e->right);
          if (isPrime(e->v.first))
            e->v.second = true;
        }
      }
  );
}
```

Рис. 2.19 ❖ Проверка простоты чисел в дереве с помощью `parallel_do`

Отметим, что в двух рассмотренных случаях причины потенциальной масштабируемости `parallel_do` различны. В реализации без пополнения на рис. 2.17 хорошей производительности можно добиться, если при каждом выполнении тела достаточно работы для амортизации накладных расходов на последовательный обход списка. В реализации с пополнением на рис. 2.19 вначале имеется только один элемент, но их число быстро растет по мере выполнения тел и добавления новых элементов.

Более сложный пример: прямая подстановка

Прямая подстановка – это метод решения системы уравнений `Ax = b`, где `A` – нижнетреугольная матрица размера `n×n`. В матричном виде систему можно записать так:

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

и решать построчно:

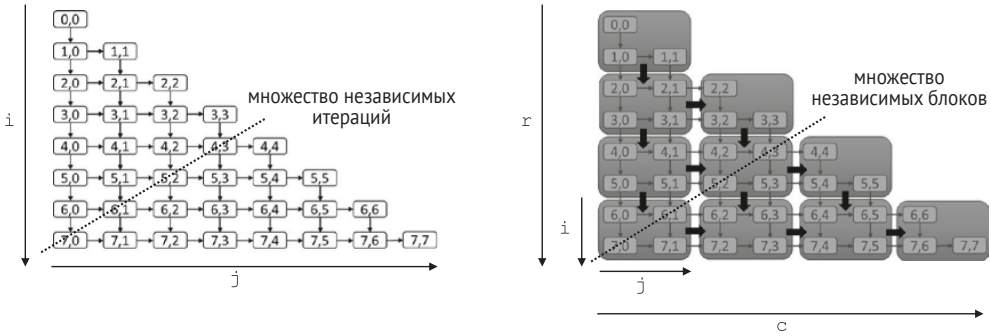
$$x_1 = b_1/a_{11}$$
$$x_2 = (b_2 - a_{21}x_1)/a_{22}$$
$$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$$
$$\vdots$$
$$x_m = (b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1})/a_{nn}.$$

Последовательный код для реализации этого алгоритма в лоб показан на рис. 2.20. Здесь вектор `b` затирается и используется для хранения суммы по каждой строке.

```
void fig_2_20(std::vector<double> &x,
              const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < i; ++j) {
      b[i] -= a[j + i*N] * x[j];
    }
    x[i] = b[i] / a[i + i*N];
  }
}
```

Рис. 2.20 ❖ Последовательный код для бесхитростной реализации прямой подстановки. Эта реализация призвана прояснить алгоритм, не ставя целью добиться наилучшей производительности

На рис. 2.21(а) показаны зависимости между итерациями тела во вложенном цикле по `i`, `j` на рис. 2.20. На каждой итерации внутреннего цикла по `j` (строки на рисунке) выполняется редукция в `b[i]`, и эта итерация зависит от всех элементов `x`, которые были вычислены на предыдущих итерациях цикла по `i`. Мы могли бы использовать алгоритм `parallel_reduce` для распараллеливания внутреннего цикла по `j`, но на предыдущих итерациях цикла по `i` может не хватить работы, чтобы сделать это предприятие выгодным. Пунктирная линия на рис. 2.21(а) показывает другой способ распараллелить внутренний цикл – нужно только взглянуть на пространство итераций по диагонали. Задействовать этот параллелизм позволит алгоритм `parallel_do`, который будет добавлять новые итерации только по мере удовлетворения их зависимостей – аналогично тому, как мы добавляли новые элементы в дерево по мере их обнаружения на рис. 2.19.



(a) Зависимости между итерациями (b) Зависимости между блоками

Рис. 2.21 ❖ Зависимости в методе прямой подстановки для небольшой матрицы 8×8. На рисунке (a) показаны зависимости между итерациями. На рисунке (b) итерации сгруппированы в блоки, чтобы снизить накладные расходы на планирование. На обоих рисунках каждый блок должен дождаться завершения своих соседей сверху и слева и только потом может быть безопасно выполнен

Если выражать параллелизм для каждой итерации по отдельности, то созданные задачи будут слишком малы, чтобы оправдать накладные расходы на планирование, – в каждой будет всего несколько операций с плавающей точкой. Но можно модифицировать вложенные циклы и создать блоки итераций, как показано на рис. 2.21(b). Схема зависимости остается такой же, но теперь мы можем планировать в виде задач эти более крупные блоки. Версия последовательного кода с блоками итераций показана на рис. 2.22.

```
void fig_2_22(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;

  for ( int r = 0; r < num_blocks; ++r ) {
    for ( int c = 0; c <= r; ++c ) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
    }
  }
}
```

Рис. 2.22 ❖ Последовательная реализация прямой подстановки с блоками итераций

Параллельная реализация с применением parallel_do показана на рис. 2.23. Здесь мы воспользовались интерфейсом parallel_do, который позволяет задать итераторы на начало и на конец, а не контейнер в целом. Подробности имеются в приложении В.

```
void fig_2_23(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;
  std::vector<tbb::atomic<char>> ref_count(num_blocks*num_blocks);
  for (int r = 0; r < num_blocks; ++r) {
    for (int c = 0; c <= r; ++c) {
      if (r == 0 && c == 0)
        ref_count[r*num_blocks + c] = 0;
      else if (c == 0 || r == c)
        ref_count[r*num_blocks + c] = 1;
      else
        ref_count[r*num_blocks + c] = 2;
    }
  }

  using BlockIndex = std::pair<size_t, size_t>;
  BlockIndex top_left(0,0);

  tbb::parallel_do( &top_left, &top_left+1 ,
    [&](const BlockIndex &bi,
        tbb::parallel_do_feeder<BlockIndex> &feeder) {
      size_t r = bi.first;
      size_t c = bi.second;
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
      // добавить преемника справа, если готов
      if (c + 1 <= r && --ref_count[r*num_blocks + c + 1] == 0) {
        feeder.add(BlockIndex(r, c + 1));
      }
      // добавить преемника снизу, если готов
      if (r + 1 < (size_t)num_blocks
          && --ref_count[(r+1)*num_blocks + c] == 0) {
        feeder.add(BlockIndex(r+1, c));
      }
    }
  );
}
```

Рис. 2.23 ❖ Реализация прямой подстановки с помощью parallel_do

В отличие от дерева простых чисел на рис. 2.19, здесь мы не хотим просто передавать каждый соседний блок пополнителю. Вместо этого мы инициализировали массив счетчиков `ref_count`, где храним количество блоков, которые должны завершиться, перед тем как блоку будет разрешено начать выполнение. Атомарные переменные будут обсуждаться в главе 5, а пока достаточно знать, что эти переменные можно безопасно модифицировать параллельно, в частности их декремент потокобезопасен. Счетчики инициализируются так, что левый верхний элемент ни от чего не зависит, у блоков в первом столбце и на диагонали есть одна зависимость, а у всех остальных по две зависимости – то же самое, что количество предшественников на рис. 2.21.

При обращении к `parallel_do` на рис. 2.23 сначала передается только левый верхний блок, `[&top_left, &top_left+1)`. Но при выполнении каждого тела в предложении if внизу производится декремент атомарных счетчиков блоков, зависящих от только что обработанного. Если счетчик обратился в ноль, значит, все зависимости соответствующего блока удовлетворены, и он передается пополнителю.

Как и предыдущий пример с простыми числами, этот пример демонстрирует отличительную особенность приложений, в которых используется `parallel_do`: параллелизм ограничен необходимостью осуществлять последовательный доступ к контейнеру или необходимостью динамически искать новые элементы и подавать их на вход алгоритму.

parallel_pipeline: обработка несколькими фильтрами

Второй из имеющихся в TBB алгоритмов для обработки сложных циклов – `parallel_pipeline`. Конвейером называется линейная последовательность *фильтров*, которые преобразуют проходящие через них *элементы*. Конвейеры часто используются для обработки данных, передаваемых приложению потоком, как, например, видео- либо аудиокадры или финансовые данные. В главе 3 мы обсудим интерфейсы потокового графа, которые позволяют строить более сложные графы, в т. ч. с несколькими входными или выходными фильтрами.

На рис. 2.24 показан простой цикл, который читает массивы символов, преобразует символы, инвертируя регистр, а затем записывает результаты в том же порядке в выходной файл.

Символы в каждом буфере должны обрабатываться строго по порядку, но выполнение фильтров, применяемых к разным буферам, может перекрываться во времени. На рис. 2.25(a) этот пример представлен в виде конвейера, в котором фильтр «записать буфер» применяется к буферу $\texttt{buffer}_i$, а параллельно с ним фильтр «обработать» применяется к $\texttt{buffer}_{i+1}$, а фильтр «получить буфер» читает $\texttt{buffer}_{i+2}$.

Как показано на рис. 2.25(b), в стационарном состоянии каждый фильтр занят, а интервалы их выполнения перекрываются. Но если выполнение фильтров не сбалансировано, как на рис. 2.25(c), то ускорение становится меньше. Производительность конвейера последовательных фильтров ограничена скоростью самого медленного этапа.

```
using CaseStringPtr = std::shared_ptr<std::string>;
CaseStringPtr getCaseString(std::ofstream &f);
void writeCaseString(std::ofstream &f, CaseStringPtr s);

void fig_2_24(std::ofstream &caseBeforeFile,
              std::ofstream &caseAfterFile) {
  while (CaseStringPtr s_ptr = getCaseString(caseBeforeFile)) {
    std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
      [](char c) -> char {
        if (std::islower(c))
          return std::toupper(c);
        else if (std::isupper(c))
          return std::tolower(c);
        else
          return c;
      }
    );
    writeCaseString(caseAfterFile, s_ptr);
  }
}
```

Рис. 2.24 ❖ Последовательное изменение регистра

(a) Три буфера параллельно обрабатываются разными фильтрами

	номер буфера							
получить буфер	0	1	2	3	4	5		
обработать		0	1	2	3	4	5	
записать буфер			0	1	2	3	4	5

время

(b) В стационарном состоянии, если каждый фильтр может работать только с одним элементом, максимально достижимый уровень параллелизма равен 3

	номер буфера													
получить буфер	0	1	2	3	4	5								
обработать		0	0	1	1	2	2	3	3	4	4	5	5	
записать буфер				0		1		2		3		4		5

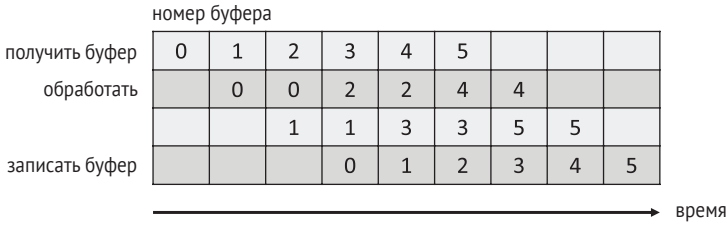
время

(c) Здесь фильтр «обработать» занимает в два раза больше времени, чем остальные, поэтому остальные фильтры иногда простаивают

Рис. 2.25 ❖ Изменение регистра с помощью конвейера

Библиотека TBB поддерживает как последовательные, так и параллельные фильтры. Параллельный фильтр можно параллельно применять к различным элементам для повышения пропускной способности. На рис. 2.26(a) приведен тот же пример, в котором средний фильтр «обработать» применяется сразу к двум элементам. А на рис. 2.26(b) показано, что если средний фильтр для любого элемента работает в два раза дольше остальных, то назначение двух потоков этому фильтру уравняет его пропускную способность с остальными.

(a) Параллельный фильтр может параллельно обрабатывать более одного элемента



(b) Параллельному фильтру можно сопоставить больше потоков и тем самым уравнять его пропускную способность с остальными

Рис. 2.26 ❖ Изменение регистра с помощью конвейера с параллельным фильтром. Благодаря двум экземплярам параллельного фильтра пропускная способность конвейера увеличилась

Полное описание интерфейсов алгоритма `parallel_pipeline` приведено в приложении B. В этом разделе мы воспользовались следующим интерфейсом:

```
void parallel_pipeline( size_t max_number_of_live_tokens,
                        const filter_t<void,void>& filter_chain);
```

Первый аргумент `max_number_of_live_tokens` – это максимальное число элементов, которым разрешено находиться в конвейере в любой момент времени. Это необходимо, чтобы ограничить потребление ресурсов. Рассмотрим, к примеру, простой конвейер с тремя фильтрами. Что, если средний фильтр последовательный и работает в 1000 раз дольше, чем фильтр, получающий новые буферы? Тогда первый фильтр сможет выделить память для 1000 буферов, но лишь для того, чтобы они встали в очередь ко второму фильтру. Очевидно, что это пустая трата памяти.

Второй аргумент parallel_pipeline – filter_chain, последовательность, полученная сцеплением фильтров, созданных функцией make_filter:

```
template<typename T, typename U, typename Func>
filter_t<T,U> make_filter(filter::mode mode, const Func& f );
```

Аргументы шаблона T и U задают типы входа и выхода фильтра. Аргумент mode может принимать значения serial_in_order, serial_out_of_order или parallel. А аргумент f задает тело фильтра. На рис. 2.27 показана реализация задачи об изменении регистра с помощью алгоритма parallel_pipeline. Полное описание интерфейсов parallel_pipeline приведено в приложении B.

Заметим, что первый фильтр, для которого входной тип равен void, получает специальный аргумент типа tbb::flow_control. Этот аргумент используется, когда нужно сигнализировать, что первый фильтр в конвейере должен прекратить порождение новых элементов. Например, в первом фильтре на рис. 2.27 мы вызываем stop(), когда указатель, возвращенный getCaseString(), равен null.

В этой реализации первый и последний фильтры созданы в режиме serial_in_order. Это означает, что каждый из них обрабатывает по одному элементу за раз и что последний фильтр обрабатывает элементы в том же порядке, в каком первый их порождает. Фильтру типа serial_out_of_order разрешено обрабатывать элементы в любом порядке. Средний фильтр создан в режиме parallel, т. е. может обрабатывать несколько элементов параллельно. Все режимы, поддерживаемые алгоритмом parallel_pipeline, описаны в приложении B.

Более сложный пример: создание трехмерных стереоскопических изображений

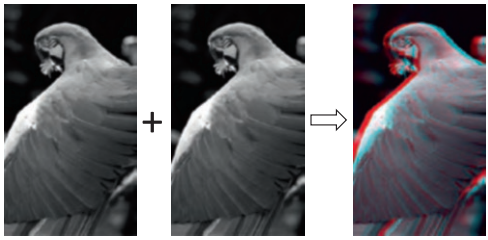
На рис. 2.28 приведен пример более сложного конвейера. В цикле while читаются номер кадров, а затем для каждого кадра читаются левое и правое изображения, и в левое изображение добавляется красный цвет, а в правое – голубой. Затем оба получившихся изображения объединяются в одно красно-голубое трехмерное стереоскопическое изображение.

Как и в примере с изменением регистра, у нас имеется последовательность изображений, пропускаемая через несколько фильтров. Мы выделяем важные функции и преобразуем их в фильтры конвейера: getNextFrameNumber, getLeftImage, getRightImage, increasePNGChannel (для левого изображения), increasePNGChannel (для правого изображения), mergePNGImages и right.write(). На рис. 2.29 рассматриваемый пример представлен в виде конвейера. Фильтр increasePNGChannel применяется дважды – сначала к левому, а затем к правому изображению.

На рис. 2.30 приведена параллельная реализация с помощью алгоритма parallel_pipeline.

```
fig_2_27(int num_tokens, std::ofstream &caseBeforeFile,
         std::ofstream &caseAfterFile) {
tbb::parallel_pipeline(
  /* tokens */ num_tokens,
  /* the get filter */
  tbb::make_filter<void, CaseStringPtr>(
    /* узел фильтра */ tbb::filter::serial_in_order,
    /* тело фильтра */
    [&](tbb::flow_control &fc) -> CaseStringPtr {
      CaseStringPtr s_ptr = getCaseString(caseBeforeFile);
      if (!s_ptr)
        fc.stop();
      return s_ptr;
    }) & // операция конкатенации
  /* построить фильтр изменения регистра */
  tbb::make_filter<CaseStringPtr, CaseStringPtr>(
    /* узел фильтра */ tbb::filter::parallel,
    /* тело фильтра */
    [](CaseStringPtr s_ptr) -> CaseStringPtr {
      std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
        [](char c) -> char {
          if (std::islower(c))
            return std::toupper(c);
          else if (std::isupper(c))
            return std::tolower(c);
          else
            return c;
        });
      return s_ptr;
    }) & // операция конкатенации
  /* построить фильтр записи */
  tbb::make_filter<CaseStringPtr, void>(
    /* узел фильтра */ tbb::filter::serial_in_order,
    /* тело фильтра */
    [&](CaseStringPtr s_ptr) -> void {
      writeCaseString(caseAfterFile, s_ptr);
    })
```

Рис. 2.27 ❖ Изменение регистра с помощью конвейера с параллельным средним фильтром



(a) Изображения объединяются в одно
красно-голубое трехмерное стереоскопическое изображение.
Оригинальная фотография сделана Еленой Адамс

```
class PNGImage {
public:
  uint64_t frameNumber = -1;
  unsigned int width = 0, height = 0;
  std::shared_ptr<std::vector<unsigned char>> buffer;
  static const int numChannels = 4;
  static const int redOffset = 0;
  static const int greenOffset = 1;
  static const int blueOffset = 2;

  PNGImage() {}
  PNGImage(uint64_t frame_number, const std::string& file_name);
  PNGImage(const PNGImage &p);
  virtual ~PNGImage() {}
  void write() const;
};

int getNextFrameNumber();
PNGImage getLeftImage(uint64_t frameNumber);
PNGImage getRightImage(uint64_t frameNumber);
void increasePNGChannel(PNGImage& image, int channel_offset,
                        int increase);
void mergePNGImages(PNGImage& right, const PNGImage& left);

void fig_2_28() {
  while (uint64_t frameNumber = getNextFrameNumber()) {
    auto left = getLeftImage(frameNumber);
    auto right = getRightImage(frameNumber);
    increasePNGChannel(left, PNGImage::redOffset, 10);
    increasePNGChannel(right, PNGImage::blueOffset, 10);
    mergePNGImages(right, left);
    right.write();
  }
}
```

(b) Последовательная реализация,
которая читает изображения и применяет эффект трехмерности

Рис. 2.28 ❖ Пример построения
красно-голубого трехмерного стереоскопического изображения

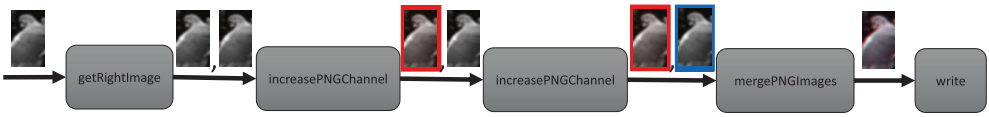


Рис. 2.29 ❖ Построение трехмерного стереоскопического изображения в виде конвейера

```
void fig_2_30() {
  using Image = PNGImage;
  using ImagePair = std::pair<PNGImage, PNGImage>;
  tbb::parallel_pipeline(
    /* количество элементов */ 8,
    /* построить фильтр левого изображения */
    tbb::make_filter<void, Image>(
      /* тип фильтра */ tbb::filter::serial_in_order,
      [&](tbb::flow_control &fc) -> Image {
        if (uint64_t frame_number = getNextFrameNumber()) {
          return getLeftImage(frame_number);
        } else {
          fc.stop();
          return Image{};
        }
      }) &
    tbb::make_filter<Image, ImagePair>(
      /* тип фильтра */ tbb::filter::serial_in_order,
      [&](Image left) -> ImagePair {
        return ImagePair(left, getRightImage(left.frameNumber));
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* тип фильтра */ tbb::filter::parallel,
      [&](ImagePair p) -> ImagePair {
        increasePNGChannel(p.first, Image::redOffset, 10);
        return p;
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* тип фильтра */ tbb::filter::parallel,
      [&](ImagePair p) -> ImagePair {
        increasePNGChannel(p.second, Image::blueOffset, 10);
        return p;
      }) &
    tbb::make_filter<ImagePair, Image>(
      /* тип фильтра */ tbb::filter::parallel,
      [&](ImagePair p) -> Image {
        mergePNGImages(p.second, p.first);
        return p.second;
      }) &
    tbb::make_filter<Image, void>(
      /* тип фильтра */ tbb::filter::parallel,
      [&](Image img) {
        img.write();
      })
  );
}
```

Рис. 2.30 ❖ Реализация построения трехмерного стереоскопического изображения с помощью `parallel_pipeline`

Функция `parallel_pipeline` линеаризует фильтры конвейера. Фильтры применяются один за другим по мере того, как входное изображение, прочитанное на первом этапе, перемещается по конвейеру. На самом деле в данном примере это является ограничением. Обработки левого и правого изображений независимы до фильтра `mergeImageBuffers`, однако интерфейс `parallel_pipeline` устроен так, что они должны выполняться в линейном порядке. Но даже при этом ограничении последовательными являются только фильтры, читающие изображения, поэтому реализация все же может оказаться масштабируемой, если время выполнения этого фильтра намного меньше времени выполнения последующих параллельных этапов.

В главе 3 мы познакомимся с потоковыми графами TBB, которые позволяет более естественно описывать приложения, выигрывающие от нелинейного выполнения фильтров.

Резюме

В этой главе приведен краткий обзор обобщенных параллельных алгоритмов в библиотеке TBB, включая паттерны функционального параллелизма, простых и сложных циклов и конвейерного параллелизма. Эти готовые алгоритмы представляют собой хорошо протестированные и оптимизированные реализации, которые можно применять к одному и тому же приложению, поступательно увеличивая его производительность.

Мы привели код простых примеров, показывающих, как используются эти алгоритмы. Во второй части книги (начиная с главы 9) мы обсудим, как выжать максимум из TBB путем композиции этих алгоритмов и настройки библиотечных средств для оптимизации локальности, минимизации накладных расходов и задания приоритетов. Во второй части мы обсудим также обработку исключений и отмену при работе с обобщенными параллельными алгоритмами.

В следующей главе мы рассмотрим еще одно высокоуровневое средство TBB – потоковый граф.

Дополнительная информация

Ниже приведен перечень рекомендуемых дополнительных материалов к этой главе.

- Мы обсуждали паттерны проектирования в параллельном программировании и их связь с обобщенными параллельными алгоритмами TBB. Подборку паттернов проектирования можно найти в книге Timothy Mattson, Beverly Sanders, and Berna Massingill «Patterns for Parallel Programming (1-е изд.), 2004, Addison-Wesley Professional.
- Обсуждая параллельную реализацию быстрой сортировки, мы отметили, что разбиение на две части по-прежнему производится последовательно и является узким местом. В следующих статьях описываются параллельные реализации разбиения:

  - *P. Heidelberger, A. Norton, and J. T. Robinson.* Parallel Quicksort using fetch-and-add. IEEE Transactions on Computers. Vol. 39. № 1. P. 133–138. Jan 1990;
  - *P. Tsigas, and Y. Zhang.* A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10 000. In 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003), p. 372–381, 2003.
- Дополнительные сведения об анализе зависимостей можно найти в ряде книг по компиляторам и параллельному программированию, в том числе:
  - *Michael Joseph Wolfe.* High-Performance Compilers for Parallel Computing. 1995, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA;
  - *Kennedy, and John R. Allen.* Optimizing Compilers for Modern Architectures. 2001, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- При обсуждении умножения матриц мы заметили, что, не являясь экспертом по оптимизации, лучше отдать предпочтение готовым пакетам линейной алгебры, если таковые существуют.

  К числу таких пакетов относятся:
  - The Basic Linear Algebra Subprograms (BLAS) по адресу www.netlib.org/blas/;
  - The Intel Math Kernel Library (Intel MKL) по адресу https://software.intel.com/mkl;
  - Automatically Tuned Linear Algebra Software (ATLAS) по адресу http://math-atlas.sourceforge.net/.

  Проект FLAME посвящен исследованиям и разработкам плотных библиотек линейной алгебры. Созданный в его рамках программный каркас BLIS можно использовать для создания высокопроизводительных библиотек BLAS. Сайт проекта FLAME размещен по адресу www.cs.utexas.edu/~flame.
- Пример с линией прямой видимости реализован с помощью параллельного сканирования на основе описания в книге Guy E. Blelloch «Vector Models for Data-Parallel Computing» (MIT Press).

Фотография на рис. 2.28а, 2.29 и 3.7 сделана Еленой Адамс и используется с разрешения авторов учебных пособий из проекта Halide по адресу http://halide-lang.org.

Глава 3

Потоковые графы

В главе 2 мы познакомились с алгоритмами, отвечающими типичным ситуациям, которые часто встречаются в приложениях. Это отличные алгоритмы! Их следует использовать всегда, когда представляется такая возможность. К сожалению, не все приложения так удобно структурированы, бывают и более сложные случаи. Встретившись с запутанной ситуацией, мы можем поддаться искушению контролировать все на микроуровне или просто решим «плыть по течению» и реагировать на события по мере поступления. TBB позволяет выбрать и тот, и другой путь.

В главе 10 мы обсудим, как работать с задачами напрямую и создавать собственные алгоритмы. Существуют высокоуровневые и низкоуровневые интерфейсы задач, так что, решив пойти по этому пути, мы действительно сможем контролировать все до мельчайших деталей, если захотим.

Но в этой главе мы рассмотрим интерфейс потоковых графов в TBB. Большинство алгоритмов в главе 2 ориентированы на приложения, в которых с самого начала имеется большой массив данных, а мы хотим создать задачи, чтобы разбить эти данные на части и обрабатывать их параллельно. Потоковый граф ориентирован на приложения, которые реагируют по мере поступления данных, а также на приложения, в которых зависимости сложнее, чем можно выразить с помощью простой структуры. Интерфейсы потокового графа успешно использовались в самых разных областях, включая обработку изображений, искусственный интеллект, финансовые услуги, здравоохранение и игры.

Интерфейсы потокового графа позволяют описать программы, содержащие параллелизм, который может быть выражен в форме графа. Часто поток данных проходит через ряд фильтров или вычислений. В этих случаях употребляется термин *графы потоков данных*. Графы могут выражать также связи «до–после» между операциями, что позволяет описывать структуры, для которых параллельного цикла или конвейера недостаточно. Для некоторых линейно-алгебраических вычислений, например разложения Холецкого, возможна эффективная параллельная реализация без дорогостоящих точек синхронизации, которые заменены отслеживанием зависимостей среди меньших операций. Графы, выражающие такие связи «до–после», называются *графами зависимостей*.

В главе 2 мы описали два обобщенных параллельных алгоритма, которые, подобно потоковому графу, не нуждаются в априорном задании всех данных: `parallel_do` и `parallel_pipeline`. Эти алгоритмы очень эффективны, но у обоих

есть ограничения, отсутствующие у потокового графа. В алгоритме `parallel_do` имеется единственная функция-тело, применяемая к каждому входному элементу по мере его появления. Алгоритм `parallel_pipeline` применяет линейную последовательность фильтров к входным элементам по мере их поступления в конвейер. В конце главы 2 мы рассмотрели пример построения трехмерного стереоскопического изображения, в котором параллелизм имел форму, не допускающую выражения с помощью линейной последовательности фильтров. API потокового графа позволяет выразить более сложные структуры, чем `parallel_do` или `parallel_pipeline`.

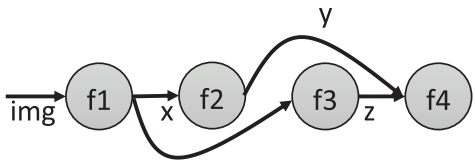
Мы начнем эту главу с обсуждения вопроса о том, почему параллелизм на основе графов так важен, а затем рассмотрим основы API потоковых графов в TBB. После этого мы представим примеры для каждого из двух основных типов графов: потокового графа данных и графа зависимостей.

Зачем использовать графы для выражения параллелизма?

Приложение, допускающее выражение в виде графа вычислений, раскрывает информацию, которую можно эффективно использовать во время выполнения для планирования параллельных вычислений. Пример показан на рис. 3.1(a).

```
while (img = getImage()) {
  x = f1(img);
  y = f2(x);
  z = f3(x);
  f4(y,z);
}
```

(a) Исходный код



(b) Граф потока данных

Рис. 3.1 ❖ Приложение, выраженное графом потока данных

На каждой итерации цикла while на рис. 3.1(a) изображение читается и передается серии фильтров: f1, f2, f3 и f4. Поток данных между этими фильтрами показан на рис. 3.1(b). На этом рисунке переменные, которые в коде использовались для передачи данных, возвращенных каждой функцией, заменены ребрами, соединяющими узел, генерирующий значение, с узлом (или узлами), потребляющим это значение.

Пока предположим, что на графе на рис. 3.1(b) отражены все данные, которыми обмениваются функции. Если это так, то мы (а значит, и библиотека типа TBB) можем многое узнать о том, какие операции допустимо выполнять параллельно. Эти выводы показаны на рис. 3.2.

На рис. 3.2 мы видим типы параллелизма, которые можно вывести из потокового графа данных в нашем небольшом примере. На этом рисунке показано прохождение четырех изображений через граф. Поскольку между узлами f2 и f3 нет ребер, их можно выполнять параллельно. Параллельное выполнение двух разных функций над одними и теми же данными – пример *функционального параллелизма* (параллелизма задач). Если предположить, что функции не

содержат побочных эффектов, т. е. не обновляют глобальное состояние, а только читают входящие сообщения и пишут в исходящие сообщения, то обработку различных сообщений в графе тоже можно совместить во времени, задействовав тем самым *конвейерный параллелизм*. Наконец, если функции *потокобезопасны*, т. е. несколько экземпляров каждой функции можно выполнять (с разными входными данными) параллельно, то можно еще совмещать обработку двух разных изображений в одном узле, задействовав *параллелизм данных*.

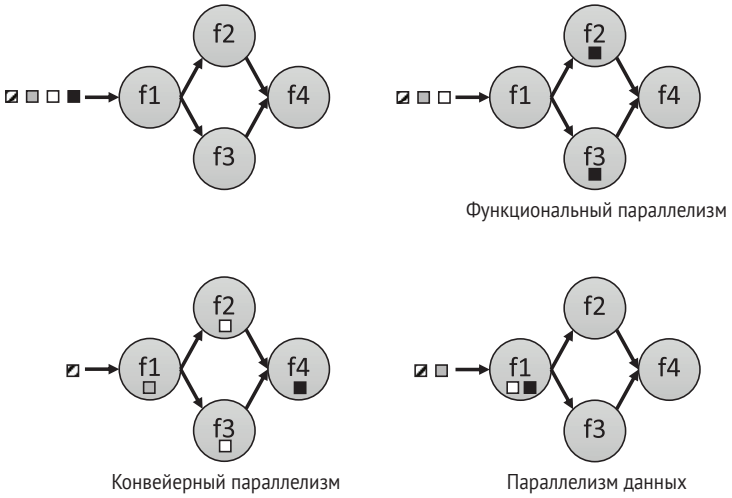


Рис. 3.2 ❖ Виды параллелизма, выводимые из графа

Выражая приложение в виде потокового графа, мы предоставляем библиотеке информацию, с помощью которой она сможет воспользоваться имеющимся параллелизмом и отобразить вычисления на платформенное оборудование в целях повышения производительности.

Основы интерфейса потоковых графов в TBB

Класс и функции потоковых графов определены в файле `flow_graph.h` и находятся в пространстве имен `tbb::flow`. Всеохватывающий заголовок `tbb.h` включает и `flow_graph.h`, так что если мы используем этот заголовок, то больше ничего включать не нужно.

Для работы с потоковым графом первым делом создается объект *граф*. Затем создаются *узлы* для выполнения операций над сообщениями, проходящими по графу, например: применение пользовательских вычислений, соединение, расщепление, буферизация или изменение порядка сообщения. Для выражения каналов сообщений, или зависимостей между узлами, служат *ребра*. Наконец, собрав граф из узлов и ребер, мы подаем ему *сообщения*. В роли сообщений могут выступать примитивные типы, объекты или указатели на объекты. Если мы хотим дождаться завершения обработки, то можем использовать объект графа как описатель.

На рис. 3.3 приведен небольшой пример, в котором выполняются все пять шагов, необходимых для использования потокового графа. В этом разделе мы подробно обсудим их.

(a) Граф с двумя узлами

```
#include <iostream>
#include <tbb/tbb.h>

void fig_3_3() {
  // шаг 1: сконструировать граф
  tbb::flow::graph g;

  // шаг 2: создать узлы
  tbb::flow::function_node<int, std::string> my_first_node(g,
    tbb::flow::unlimited,
    []( const int &in ) -> std::string {
      std::cout << "first node received: " << in << std::endl;
      return std::to_string(in);
    }
  );

  tbb::flow::function_node<std::string> my_second_node(g,
    tbb::flow::unlimited,
    []( const std::string &in ) {
      std::cout << "second node received: " << in << std::endl;
    }
  );

  // шаг 3: добавить ребра
  tbb::flow::make_edge(my_first_node, my_second_node);

  // шаг 4: отправить сообщения
  my_first_node.try_put(10);

  // шаг 5: ждать завершения графа
  g.wait_for_all();
}
```

(b) Исходный код графа с двумя узлами

Рис. 3.3 ❖ Пример потокового графа с двумя узлами

Шаг 1: создать объект графа

Первым шагом создания потокового графа является конструирование объекта графа. Этот объект служит для выполнения всех операций над графом, например ожидание завершения всех связанных с графом операций, сброс состояния всех узлов графа или отмена выполнения во всех узлах графа. Каждый узел принадлежит ровно одному графу, а ребра проводятся между узлами одного

и того же графа. Сконструировав граф, мы должны сконструировать узлы, реализующие вычисления в графе.

Шаг 2: создать узлы

В интерфейсе потокового графа определен богатый набор типов узлов (рис. 3.4), который можно грубо разбить на три группы: функциональные узлы, узлы управления потоком (включая узлы соединения) и узлы буферизации. Полные

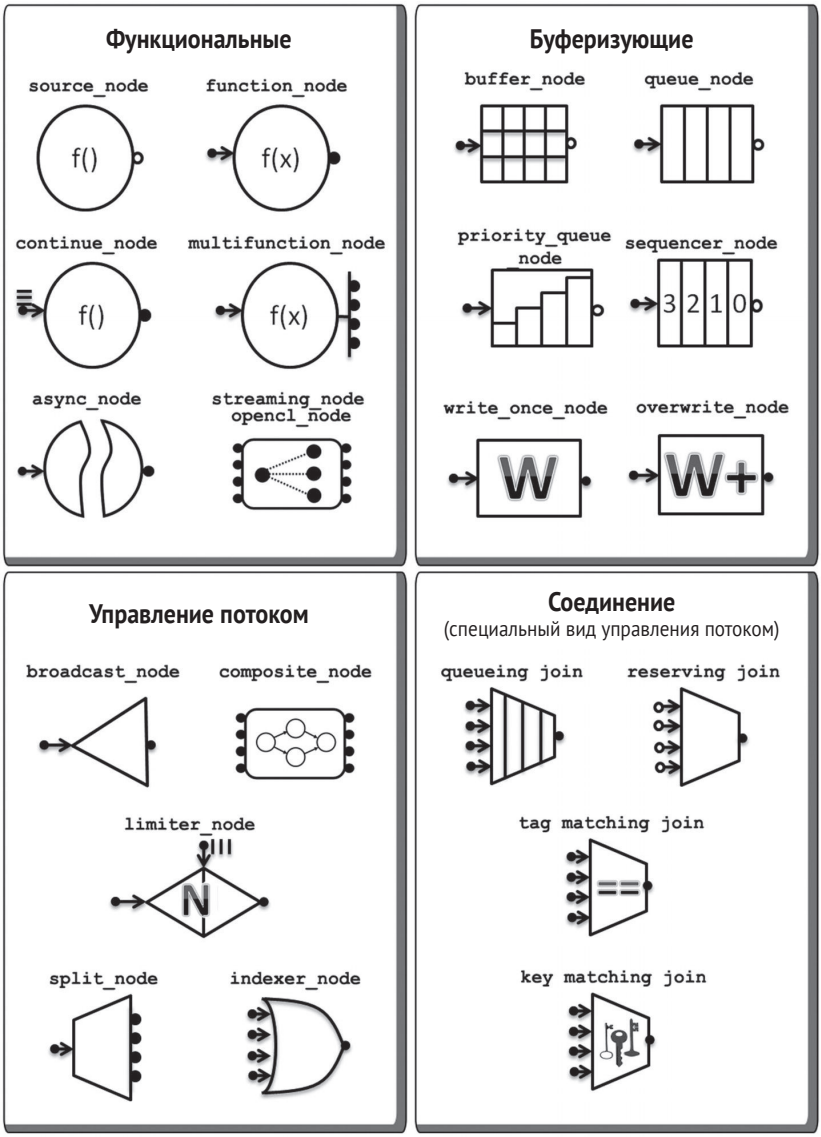


Рис. 3.4 ❖ Типы узлов потокового графа
(см. главы 3, 17, 18, 19 и описание интерфейсов в приложении В)

сведения об интерфейсах класса graph и всех типах узлов см. в разделе «Потоковый граф: узлы» приложения В. Мы не ожидаем, что вы сейчас досконально изучите эти таблицы, нужно только знать, где искать справочную информацию, когда узлы будут использоваться в этой и последующих главах.

Как и все функциональные узлы, function_node принимает лямбда-выражение в качестве одного из аргументов. Такие аргументы-тела используются в функциональных узлах для передачи кода, который узел должен применять к входящим сообщениям. На рис. 3.3 первый узел принимает значение типа int, печатает его, а затем преобразует в std::string и возвращает преобразованное значение. Ниже воспроизведено определение этого узла.

```
tbb::flow::function_node<int, std::string> my_first_node(g,
  tbb::flow::unlimited,
  [](const int &in) -> std::string {
    std::cout << "first node received: " << in << std::endl;
    return std::to_string(in);
  }
);
```

Узлы обычно соединяются друг с другом ребрами, но можно и явно послать сообщение узлу. Например, чтобы послать сообщение узлу my_first_node, нужно вызвать его метод try_put:

```
my_first_node.try_put(10);
```

В результате TBB запустит задачу для выполнения тела узла my_first_node с сообщением, содержащим целое число 10, и напечатает такой результат:

```
first node received: 10
```

В отличие от функциональных узлов, которым передается аргумент-тело, узлы управления потоком выполняют предопределенные операции: соединение, расщепление или перенаправление входящих сообщений. Например, можно создать узел join_node, который соединяет сообщения, поступающие в несколько входных портов, и создает выходное сообщение типа std::tuple<int, std::string, double>. Для этого нужно задать тип кортежа, политику соединения и ссылку на объект графа:

```
join_node<tuple<int, std::string, double>,
          queueing> j(g);
```

Этот узел j типа join_node имеет три входных порта и один выходной. Входной порт 0 принимает сообщения типа int, входной порт 1 – сообщения типа std::string, а входной порт 2 – сообщения типа double. Один выходной порт выдает сообщения типа std::tuple<int, std::string, double>.

Для узла типа join_node можно задать одну из четырех политик: queueing, reserving, key_matching и tag_matching. В случае политик queueing, key_matching и tag_matching узел буферизует входящие сообщения, поступающие через любой из входных портов. Политика queueing сохраняет входящие сообщения в очередях, связанных с портами, и объединяет сообщения в кортеж, применяя дисциплину «первым пришел – первым обслужен». Политики key_matching и tag_matching сохраняют сообщения в отображениях, связанных с портами, и объединяют сообщения на основе совпадения ключей или тегов.

Узел join_node с политикой reserving не буферизует входящие сообщения. Вместо этого он отслеживает состояние предшествующих буферов; если он полагает, что для каждого из его выходных портов есть сообщение, то пытается зарезервировать элемент для каждого входного порта. Пока резервирование действует, никакой другой узел не может потребить элемент. Только если узел типа join_node сумел успешно зарезервировать элемент для каждого своего входного порта, он потребляет сообщения; в противном случае все резервирования освобождаются и сообщения остаются в предыдущих буферах. Если узлу join_node с политикой reserving не удалось зарезервировать элементы для всех своих входов, он повторяет попытку. Примеры применения такой политики мы увидим в главе 17.

Буферизующие узлы запоминают сообщения в буферах. Поскольку функциональные узлы, function_node и multifunction_node, содержат буферы для каждого входа, а source_node – для выхода, то буферизующие узлы имеют ограниченное применение – обычно они используются совместно с резервирующим узлом (см. главу 17).

Шаг 3: добавить ребра

Сконструировав объект графа и узлы, мы вызываем функцию make_edge, чтобы задать каналы сообщений или зависимости:

```
make_edge(predecessor_node, successor_node);
```

Если в узле более одного входного или выходного порта, то для выбора порта используются шаблонные функции input_port и output_port:

```
make_edge(output_port<0>(predecessor_node),
          input_port<1>(successor_node));
```

На рис. 3.3 было проведено ребро между my_first_node и my_second_node в нашем простом графе с двумя узлами. На рис. 3.5 показан чуть более сложный потоковый граф с четырьмя узлами.

Первые два узла на рис. 3.5 порождают результаты, которые объединяются в кортеж узлом my_join_node с политикой queueing. При создании ребер, ведущих во входные порты узла типа join_node, необходимо задать номер порта:

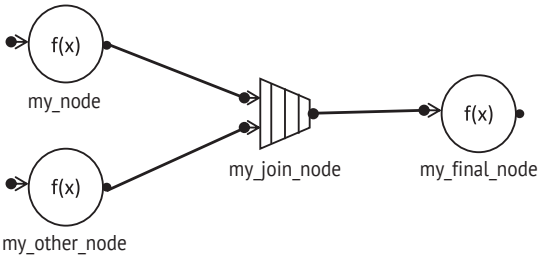
```
make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
```

Выход my_join_node, значение типа std::tuple<std::string, double>, посылается узлу my_final_node. Указывать номер порта не нужно, потому что порт всего один.

```
make_edge(my_join_node, my_final_node);
```

Шаг 4: запустить граф

Четвертый шаг создания и использования потокового графа – запустить выполнение графа. Есть два основных пути попадания сообщений в граф: (1) явный вызов try_put с указанием узла или (2) выход узла типа source_node. На рис. 3.3 и 3.5 мы вызывали try_put, чтобы запустить поток сообщений.



(a) Граф с четырьмя узлами

```
void fig_3_5() {
  // шаг 1: сконструировать граф
  tbb::flow::graph g;
  // шаг 2: создать узлы
  tbb::flow::function_node<int, std::string> my_node(g,
    tbb::flow::unlimited,
    []( const int &in ) -> std::string {
      std::cout << "received: " << in << std::endl;
      return std::to_string(in);
    }
  );
  tbb::flow::function_node<int, double> my_other_node(g,
    tbb::flow::unlimited,
    [](const int &in) -> double {
      std::cout << "other received: " << in << std::endl;
      return double(in);
    }
  );
  tbb::flow::join_node<std::tuple<std::string, double>,
                       tbb::flow::queueing> my_join_node(g);
  tbb::flow::function_node<std::tuple<std::string, double>,
                           int> my_final_node(g,
    tbb::flow::unlimited,
    [](const std::tuple<std::string, double> &in) -> int {
      std::cout << "final: " << std::get<0>(in)
                << " and " << std::get<1>(in) << std::endl;
      return 0;
    }
  );
  // шаг 3: добавить ребра
  make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
  make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
  make_edge(my_join_node, my_final_node);
  // шаг 4: отправить сообщения
  my_node.try_put(1);
  my_other_node.try_put(2);
  // шаг 5: ждать завершения графа
  g.wait_for_all();
}
```

(b) Исходный код графа с четырьмя узлами, содержащего узел `join_node`

Рис. 3.5 ❖ Пример потокового графа с четырьмя узлами

Узел типа `source_node` по умолчанию конструируется в активном состоянии. Всякий раз, как создается исходящее ребро, он сразу же начинает посылать через него сообщения. На наш взгляд, такая схема чревата ошибками, поэтому мы

всегда конструируем узлы source_node в неактивном состоянии, т. е. передаем false в аргументе is_active. Чтобы запустить поток сообщений по завершении построения графа, мы вызываем функцию activate() для всех неактивных узлов.

На рис. 3.6 показано использование узла типа source_node в качестве замены последовательного цикла для подачи сообщений в граф. На рис. 3.6(a) в цикле вызывается try_put для узла my_node, чтобы отправлять ему сообщения. На рис. 3.6(b) для той же цели используется узел типа source_node.

```
void loop_with_try_put() {
  const int limit = 3;
  tbb::flow::graph g;
  tbb::flow::function_node<int> my_node(g, tbb::flow::unlimited,
    [](int i) {
      std::cout << i << std::endl;
    }
  );
  for (int count = 0; count < limit; ++count) {
    int value = count;
    my_node.try_put(value);
  }
  g.wait_for_all();
}
```

(a) Использование цикла for и явных вызовов try_put

```
void fig_3_6() {
  tbb::flow::graph g;
  int count = 0;
  tbb::flow::source_node<int> my_src(g,
    [&count](int &i) -> bool {
      const int limit = 3;
      if (count < limit) {
        i = count++;
        return true;
      } else {
        return false;
      }
    },
    false);
  tbb::flow::function_node<int> my_node(g,
    tbb::flow::unlimited,
    [](int i) {
      std::cout << i << std::endl;
    }
  );
  tbb::flow::make_edge(my_src, my_node);
  my_src.activate();
  g.wait_for_all();
}
```

(b) Использование source_node вместо цикла

Рис. 3.6 ❖ В (a) цикл посылает значения 0, 1 и 2 узлу my_node. В (b) source_node посылает значения 0, 1 и 2 узлу my_node

Значение, возвращенное узлом source_node, играет роль булева условия в последовательном цикле – если оно равно true, тело цикла выполняется еще раз, иначе цикл прекращается. Узел source_node в этой роли возвращает булево значение в зависимости от текущего счетчика и дополнительно обновляет сам счетчик. На рис. 3.6(b) узел source_node заменяет цикл по count рис. 3.6(a).

Основное преимущество source_node по сравнению с циклом состоит в том, что он может реагировать на другие узлы графа. В главе 17 мы обсудим, как source_node можно использовать в сочетании с резервирующим узлом join_node или с узлом типа limiter_node для управления количеством сообщений, поступающих в граф. При использовании простого цикла можно затопить граф входными данными, вынуждая узлы буферизовать много сообщений, если они не успевают их обработать.

Шаг 5: ждать завершения выполнения графа

Поскольку мы посылали графу сообщения с помощью try_put или source_node, то должны дождаться завершения выполнения, вызвав метод wait_for_all() объекта графа. Эти вызовы присутствуют на рис. 3.3, 3.5 и 3.6.

После выполнения графа на рис. 3.3 мы увидим такой результат:

```
first node received: 10
second node received: 10
```

После выполнения графа на рис. 3.5 результат будет таким:

```
other received: received: 21

final: 1 and 2
```

Результат выполнения графа на рис. 3.5 выглядит несколько беспорядочно. Дело вот в чем. Первые два функциональных узла работают параллельно, и оба пишут на std::cout. В показанном выше выводе результаты двух узлов перемешаны, потому что мы нарушили предположение, сделанное в начале главы при обсуждении параллелизма на основе графов: наши узлы содержат побочные эффекты! Оба узла изменяют состояние глобального объекта std::cout. В данном примере это не страшно, потому что печать нужна, только чтобы показать прохождение сообщений по графу. Однако этот важный момент следует запомнить.

Последний узел типа function_node на рис. 3.5 выполняется, только когда оба значения из предшествующих функциональных узлов соединяются узлом join_node и передаются ему. Он выводит на std::cout ожидаемый результат: «final: 1 and 2».

Более сложный пример потокового графа данных

В главе 2 мы рассматривали применение трехмерного стереоскопического эффекта к парам изображений: левому и правому. Этот пример был распараллелен с помощью алгоритма parallel_pipeline, но мы признали, что не весь потенциал параллелизма был задействован, поскольку этому мешала линеаризация этапов конвейера. Пример вывода показан на рис. 3.7.

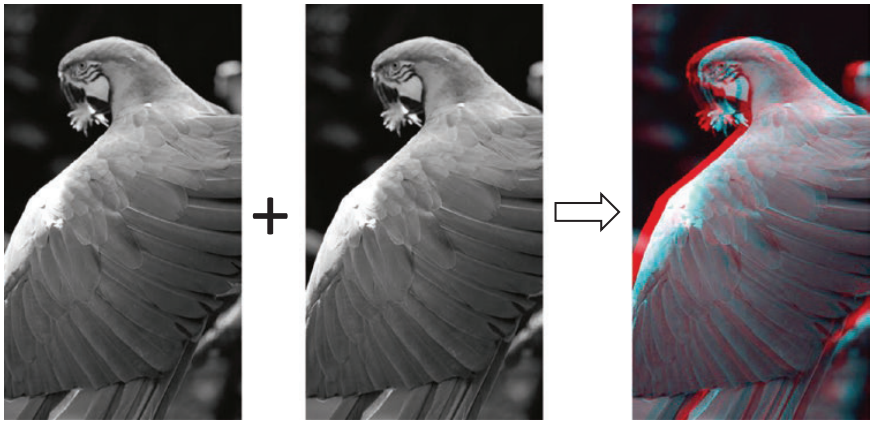


Рис. 3.7 ❖ Левое и правое изображения,
использованные для создания красно-голубого стереоскопического изображения

На рис. 3.8 показаны зависимости по данным и по управлению, существующие в последовательном коде на рис. 2.28. Зависимости по данным показаны сплошными линиями, а по управлению – пунктирными. По рисунку видно, что вызовы `getLeftImage`, сопровождаемые `increasePNGChannel`, не зависят от вызовов `getRightImage`, сопровождаемых `increasePNGChannel`. Следовательно, эти две цепочки вызовов можно выполнять параллельно. Видно также, что `mergePNGImages` не может начаться до завершения вызовов `increasePNGChannel` для левого и правого изображений. И наконец, с вызовом `write` придется подождать до завершения `mergePNGImages`.

В отличие от главы 2, где использовался линейный конвейер, потоковый граф позволяет выразить эти зависимости более точно. Для этого нужно сначала понять, какие ограничения в приложении обеспечивают правильный порядок выполнения. Например, следующая итерация цикла while начинается не раньше завершения предыдущей, но, возможно, это просто побочный эффект использования последовательного цикла. Нам необходимо решить, какие ограничения действительно необходимы.

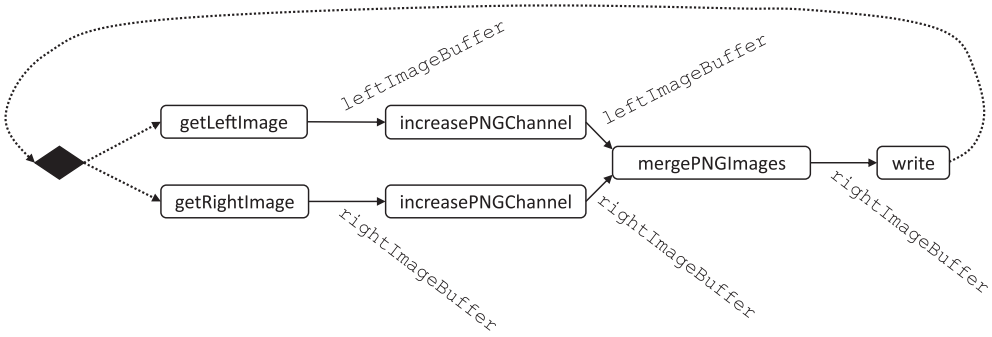


Рис. 3.8 ❖ Зависимости по данным и по управлению для кода на рис. 2.28.
Сплошными линиями показаны зависимости по данным, пунктирными – по управлению

В этом примере предположим, что изображения – это кадры, читаемые по порядку из файла или с камеры. Поскольку изображения читаются по порядку, мы не вправе выполнять несколько вызовов `getLeftImage` или `getRightImage` параллельно – это последовательные операции. Но мы можем совместить вызов `getLeftImage` с вызовом `getRightImage`, потому что эти функции не мешают друг другу. Сверх того будем предполагать, что функции `increasePNGChannel`, `mergePNGImages` и `write` безопасно выполнять параллельно для разных входных данных (они потокобезопасны и не имеют побочных эффектов). Таким образом, итерации цикла while нельзя выполнять полностью параллельно, но и внутри, и между итерациями имеется кое-какой параллелизм, который мы можем задействовать, при условии что перечисленные выше ограничения соблюдаются.

Реализация примера в виде потокового графа TBB

Теперь разберем по шагам построение потокового графа, реализующего пример с добавлением стереоскопического эффекта. Структура интересующего нас потокового графа показана на рис. 3.9. Она похожа на изображенную на рис. 3.8, что и неудивительно, поскольку узлы теперь соответствуют узлам графа, а ребра – ребрам графа.

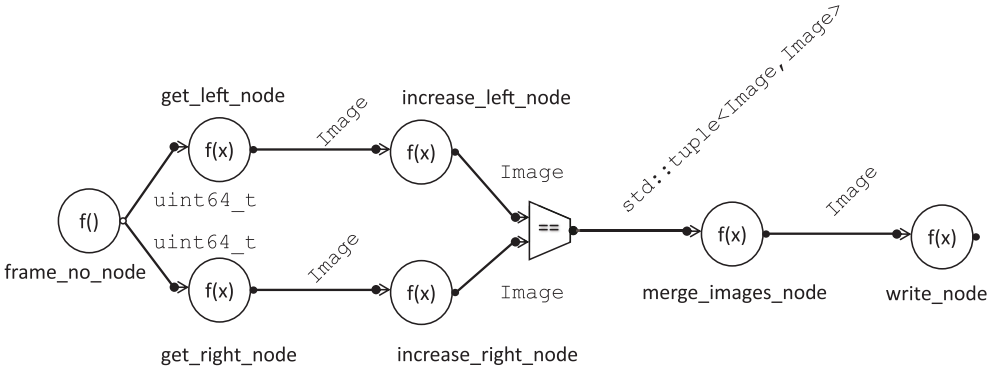


Рис. 3.9 ❖ Граф, представляющий вызовы на рис. 2.28. Кружочки инкапсулируют функции, ребра представляют промежуточные значения, а трапеция – узел, объединяющий сообщения в кортеж из двух элементов

На рис. 3.10 приведен код, реализующий потоковый граф в рассматриваемом примере. Пять основных шагов обведены рамкой. Сначала мы создаем объект графа, затем – все восемь узлов, включая `source_node`, несколько узлов `function_node` и `join_node`. Далее мы вызываем `make_edge` для соединения узлов между собой. Создав все ребра, мы активируем узел-источник. И наконец, ждем завершения графа.

На диаграмме на рис. 3.9 видно, что `frame_no_node` – источник входных данных для графа, а на рис. 3.10 этот узел реализован с помощью `source_node`. Пока тело `source_node` возвращает `true`, библиотека продолжает запускать новые задачи для его выполнения, и при этом вызывается функция `getNextFrameNumber()`.

Выше мы отмечали, что функции getLeftImage и getRightImage должны выполняться последовательно. В коде на рис. 3.10 мы сообщаем об этом ограничении библиотеке, задавая для этих узлов ограничение конкурентности flow::serial. Оба этих узла реализованы с помощью класса function_node. Дополнительные сведения о function_node имеются в приложении В. Если для узла задан режим flow::serial, то библиотека не будет запускать следующую задачу для выполнения его тела, пока не завершится тело работающей задачи.

С другой стороны, при конструировании объектов increase_left_node и increase_rigt_node задано ограничение конкурентности flow::unlimited. Во время выполнения библиотека запустит задачу для выполнения тела этих узлов, как только поступит входящее сообщение.

На рис. 3.9 мы видим, что функции merge_images_node необходимы и правое, и левое изображения. В первоначальном последовательном коде мы были уверены, что оба изображения относятся к одному кадру, поскольку цикл while в каждый момент времени работал только с одним кадром. Но в варианте с потоковым графом в конвейере может находиться одновременно несколько кадров. Поэтому необходимо как-то гарантировать, что мы объединяем изображения, относящиеся к одному и тому же кадру.

Чтобы на вход merge_images_node подавались соответственные изображения, мы создаем узел join_images_node с политикой tag_matching (см. приложение В). На рис. 3.10 join_images_node конструируется с двумя входными портами и создает кортеж объектов Image с одинаковыми данными-членами frameNumber. Теперь конструктору передаются два лямбда-выражения, которые получают значения тега из входящих сообщений в обоих портах. Узел merge_images_node принимает кортеж и порождает одно объединенное изображение.

Последним на рис. 3.10 создается узел write_node. Это узел типа function_node с ограничением flow::unlimited, который получает объекты Image и вызывает функцию write для записи входящего буфера в выходной файл.

Сконструированные узлы соединяются между собой с помощью вызовов make_edge, в результате чего создается топология, показанная на рис. 3.9. Следует отметить, что для узлов, имеющих только один вход или выход, задавать порт не нужно. Но для узлов с несколькими входными портами, как, например, join_images_node, используются функции доступа к портам, чтобы указать при вызове make_edge конкретные порты.

Наконец, мы активируем узел frame_no_node и обращаемся к wait_for_all, чтобы дождаться завершения выполнения графа.

Производительность потокового графа данных

Важно отметить, что, в отличие от некоторых других систем для работы с потоками данных, узлы в потоковом графе TBB не реализованы как потоки. Задачи в TBB запускаются в ответ на поступление сообщения с соблюдением ограничений конкурентности. Запущенные задачи отображаются на рабочие потоки TBB планировщиком с применением того же подхода на основе заимствования работ, что в обобщенных алгоритмах TBB (подробнее о планировщиках с заимствованием работ см. главу 9).

```
tbb::flow::graph g;
```

```
tbb::flow::source_node<uint64_t> frame_no_node(g,
  [] (uint64_t &frame_number) -> bool {
    if (frame_number = getNextFrameNumber())
      return true;
    else
      return false;}, false);
tbb::flow::function_node<uint64_t, Image> get_left_node(g, tbb::flow::serial,
  [] (uint64_t frame_number) -> Image {
    return getLeftImage(frame_number);});
tbb::flow::function_node<uint64_t, Image> get_right_node(g, tbb::flow::serial,
  [] (uint64_t frame_number) -> Image {
    return getRightImage(frame_number);});
tbb::flow::function_node<Image, Image> increase_left_node(g,
  tbb::flow::unlimited,
  [] (Image left) -> Image {
      increasePNGChannel(left, Image::redOffset, 10);
      return left;});
tbb::flow::function_node<Image, Image> increase_right_node(g,
  tbb::flow::unlimited,
  [] (Image right) -> Image {
      increasePNGChannel(right, Image::blueOffset, 10);
      return right;});
tbb::flow::join_node<std::tuple<Image, Image>, tbb::flow::tag_matching >
  join_images_node(g, [] (Image left) { return left.frameNumber; },
                      [] (Image right) { return right.frameNumber; } );
tbb::flow::function_node<std::tuple<Image, Image>, Image>
merge_images_node(g, tbb::flow::unlimited,
  [] (std::tuple<Image, Image> t) -> Image {
    auto &l = std::get<0>(t);
    auto &r = std::get<1>(t);
    mergePNGImages(r, l);
    return r;});
tbb::flow::function_node<Image> write_node(g, tbb::flow::unlimited,
  [] (Image img) {
    img.write();});
```

```
tbb::flow::make_edge(frame_no_node, get_left_node);
tbb::flow::make_edge(frame_no_node, get_right_node);
tbb::flow::make_edge(get_left_node, increase_left_node);
tbb::flow::make_edge(get_right_node, increase_right_node);
tbb::flow::make_edge(increase_left_node,
                     tbb::flow::input_port<0>(join_images_node));
tbb::flow::make_edge(increase_right_node,
                     tbb::flow::input_port<1>(join_images_node));
tbb::flow::make_edge(join_images_node, merge_images_node);
tbb::flow::make_edge(merge_images_node, write_node);
```

```
frame_no_node.activate();
```

```
g.wait_for_all();
```

Рис. 3.10 ❖ Пример создания стереоскопического эффекта с помощью потокового графа

Есть три основных фактора, потенциально способных ограничить производительность потокового графа: (1) последовательные узлы, (2) количество рабочих потоков и (3) накладные расходы, связанные с параллельным выполнением задач TBB.

Рассмотрим, как граф для рассматриваемого примера можно отобразить на задачи TBB и какова возможная производительность этих задач. Узлы `frame_no_node`, `get_left_node` и `get_right_node` последовательные (работают в режиме `flow::serial`). Для остальных узлов установлен режим конкурентности `flow::unlimited`.

Последовательные узлы могут приводить к простою рабочих потоков, поскольку ограничивают доступность задач. В нашем примере изображения читаются по порядку. Как только изображение прочитано, его обработка может начаться немедленно и может быть совмещена с любой другой работой в системе. Стало быть, именно эти три последовательных узла ограничивают доступность задач в нашем графе. Если время чтения изображения преобладает в общем времени обработки, то ускорение будет очень мало. Но если время обработки гораздо больше времени чтения, то ускорение может оказаться значительным.

Если чтение изображения не является лимитирующим фактором, то производительность ограничена количеством рабочих потоков и накладными расходами на параллельное выполнение. При использовании потокового графа мы передаем данные между узлами, которые могут исполняться в разных потоках и на разных процессорных ядрах. Кроме того, выполнение различных функций перекрывается во времени. И передача данных между потоками, и одновременное выполнение функций разными потоками могут влиять на поведение памяти и кеша. Мы обсудим локальность и оптимизацию накладных расходов во второй части книги.

Частный случай – графы зависимостей

Интерфейсы потокового графа TBB поддерживают как графы потоков данных, так и графы зависимостей. Ребра графа потоков данных – это каналы, по которым данные передаются между узлами. Рассмотренная выше задача построения стереоскопического изображения дает пример графа потоков данных – объекты `Image` передаются вдоль ребер, соединяющих узлы.

Ребра в графе зависимостей представляют связи «до–после», которые должны поддерживаться для правильного выполнения. В графе зависимостей данные передаются от узла к узлу с использованием разделяемой памяти, а не в сообщениях, циркулирующих по ребрам графа. На рис. 3.11 показан граф зависимостей для приготовления сэндвича с арахисовым маслом и джемом; ребра означают, что выполнение узла не может начаться, пока не завершатся ***все*** его предшественники.

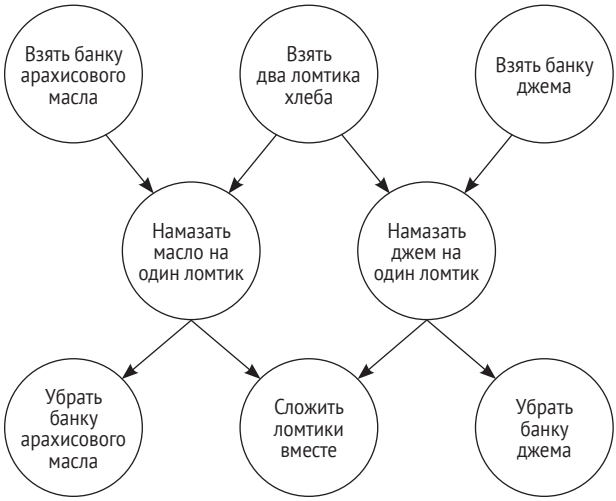


Рис. 3.11 ❖ Граф зависимостей для приготовления сэндвича с арахисовым маслом и джемом. Ребра представляют связи «до – после»

Для выражения графов зависимостей используется класс `continue_node` для узлов и передаваемые сообщения типа `continue_msg`. Главное различие между `function_node` и `continue_node` заключается в реакции на сообщения. Когда узел типа `function_node` получает сообщение, он применяет к нему свое тело – либо немедленно запускает задачу, либо буферизует сообщение в ожидании момента, когда будет разрешено запустить задачу. С другой стороны, `continue_node` подсчитывает количество полученных сообщений. Когда оно станет равно количеству предшествующих узлов, запускается задача для исполнения тела, и затем счетчик сообщений сбрасывается. Например, если бы мы захотели реализовать граф на рис. 3.11 с помощью узлов типа `continue_node`, то узел «Сложить ломтики вместе» выполнялся бы каждый раз после получения двух объектов `continue_msg`, поскольку в графе у него два предшественника.

Объекты типа `continue_node` подсчитывают сообщения, но не следят за тем, какой конкретно предшественник отправил данное сообщение. Так, если у узла два предшественника, то он будет выполняться после получения двух сообщений вне зависимости от их происхождения. В результате накладные расходы этих узлов гораздо ниже, но зато требуется, чтобы граф зависимостей был ациклическим. Кроме того, хотя граф зависимостей можно выполнять повторно до завершения, подавать ему объекты `continue_msg` потоком небезопасно. Если имеются циклы или элементы подаются в граф зависимостей потоком, то простой механизм подсчета не годится, поскольку узел может по ошибке активироваться, посчитав сообщения, полученные от одного и того же преемника, хотя должен был бы ждать входные данные от разных преемников.

Реализация графа зависимостей

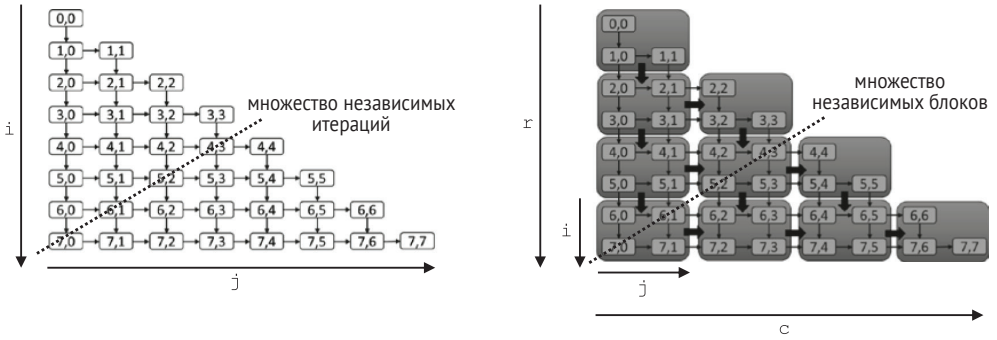
Шаги при работе с графом зависимостей такие же, как при работе с графом потока данных: создать объект графа, создать узлы, добавить ребра и подавать графу сообщения. Основные различия следующие: используются только классы `continue_node` и `broadcast_node`, граф должен быть ациклическим, и перед тем как подавать графу новое сообщение, необходимо дождаться завершения обработки предыдущего.

А теперь построим пример графа зависимостей. Мы реализуем тот же метод прямой подстановки, который в главе 2 осуществили с помощью `parallel_do`. Подробное описание последовательного алгоритма можно найти там. А ниже повторена его реализация.

```
void fig_3_12(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 512;
  const int num_blocks = N / block_size;

  for ( int r = 0; r < num_blocks; ++r ) {
    for ( int c = 0; c <= r; ++c ) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
    }
  }
}
```

Рис. 3.12 ❖ Последовательный код реализации прямой подстановки с блоками итераций. Эта реализация призвана прояснить алгоритм, не ставя целью добиться наилучшей производительности

В главе 2 мы обсуждали зависимости между операциями в этом примере и заметили, что, как повторно показано на рис. 3.13, параллелизм распространяется фронтом, и это можно заметить, глядя на вычисления по диагонали. Используя алгоритм `parallel_do`, мы создали двумерный массив атомарных счетчиков и должны были вручную следить за тем, какой блок безопасно подать `parallel_do` для выполнения. Это эффективно, но громоздко и чревато ошибками.



(a) Зависимости между итерациями

(b) Зависимости между блоками

Рис. 3.13 ❖ Зависимости в методе прямой подстановки для небольшой матрицы 8×8. На рисунке (a) показаны зависимости между итерациями. На рисунке (b) итерации сгруппированы в блоки, чтобы снизить накладные расходы на планирование. На обоих рисунках каждый блок должен дождаться завершения своих соседей сверху и слева и только потом может быть безопасно выполнен

В главе 2 было отмечено, что для выражения параллелизма в этом примере можно было бы также использовать алгоритм `parallel_reduce`. Такая реализация показана на рис. 3.14.

```
void fig_3_14(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  for (int i = 0; i < N; ++i) {
    b[i] -= tbb::parallel_reduce(tbb::blocked_range<int>(0,i), 0.0,
      [&a, &x, i, N] (const tbb::blocked_range<int> &b, double init)
        -> double {
        for (int j = b.begin(); j < b.end(); ++j) {
          init += a[j + i*N] * x[j];
        }
        return init;
      },
      std::plus<double>()
    );
    x[i] = b[i] / a[i + i*N];
  }
}
```

Рис. 3.14 ❖ Применение `parallel_reduce` для распараллеливания прямой подстановки

Однако, как видно на рис. 3.15, главный поток должен ждать завершения каждого `parallel_reduce`, прежде чем переходить к следующему. Эта синхронизация между строками вообще-то излишняя. Например, после того как блок 1,0 обработан, вполне можно безопасно начинать обработку блока 2,0, но мы вынуждены ждать, пока алгоритм разветвления–соединения `parallel_reduce` закончится, прежде чем переходить к этой строке.

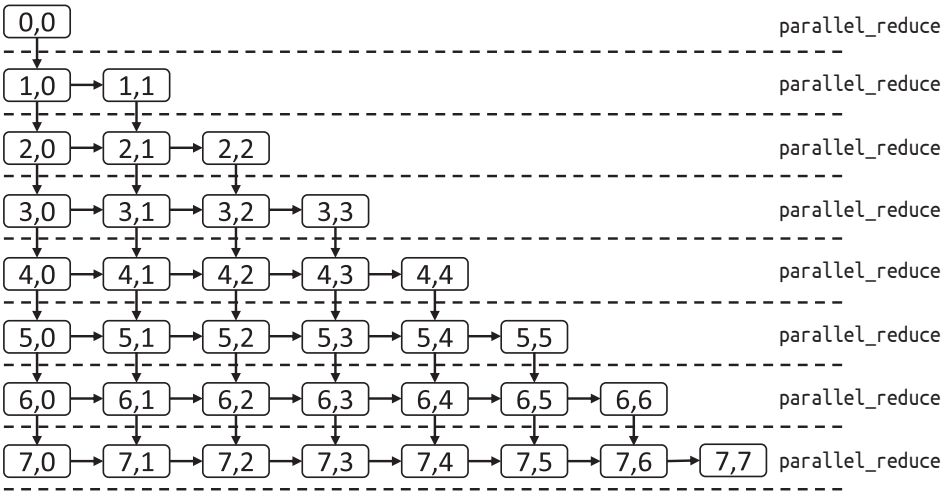


Рис. 3.15 ❖ Главный поток должен ждать завершения каждого `parallel_reduce`, прежде чем перейти к следующему `parallel_reduce`, что вводит лишние точки синхронизации

Используя граф зависимостей, мы можем выразить эти зависимости непосредственно и дать библиотеке TBB возможность выявить и задействовать имеющийся в графе параллелизм. Нам ни к чему поддерживать счетчики или явно отслеживать завершения, как в версии на основе `parallel_do` из главы 2, и мы не вводим ненужных точек синхронизации, как на рис. 3.14.

На рис. 3.16 показан граф зависимостей для этого примера. В векторе `std::vector nodes` хранятся объекты `continue_node`, каждый из которых представляет один блок итераций. При создании графа мы следуем общей схеме: (1) создать объект графа, (2) создать узлы, (3) добавить ребра, (4) подавать в граф сообщения и (5) ждать завершения графа. Но теперь мы создаем структуру графа во вложенном цикле, как показано на рис. 3.16. Функция `createNode` создает объект типа `continue_node` для каждого блока, а функция `addEdges` соединяет узел с соседями, которые должны дожидаться его завершения.

```
#include <iostream>
#include <memory>
#include <vector>
#include <tbb/tbb.h>

using Node = tbb::flow::continue_node<tbb::flow::continue_msg>;
using NodePtr = std::shared_ptr<Node>;

NodePtr createNode(tbb::flow::graph &g, int r, int c, int block_size,
                   std::vector<double> &x,
                   const std::vector<double> &a,
                   std::vector<double> &b);

void addEdges(std::vector<NodePtr> &nodes, int r, int c,
              int block_size, int num_blocks);

void fig_3_16(std::vector<double> &x, const std::vector<double> &a,
              std::vector<double> &b) {
  const int N = x.size();
  const int block_size = 1024;
  const int num_blocks = N / block_size;

  std::vector<NodePtr> nodes(num_blocks*num_blocks);
  tbb::flow::graph g;
  for (int r = num_blocks - 1; r >= 0; --r) {
    for (int c = r; c >= 0; --c) {
      nodes[r*num_blocks + c] =
        createNode(g, r, c, block_size, x, a, b);

      addEdges(nodes, r, c, block_size, num_blocks);
    }
  }
  nodes[0]->try_put(tbb::flow::continue_msg());

  g.wait_for_all();
}
```

Рис. 3.16 ❖ Реализация графа зависимостей для метода прямой подстановки

На рис. 3.17 приведена реализация функции `createNode`, а на рис. 3.18 – функции `addEdges`.

```
NodePtr createNode(tbb::flow::graph &g, int r, int c, int block_size,
                   std::vector<double> &x,
                   const std::vector<double> &a,
                   std::vector<double> &b) {
  const int N = x.size();
  return std::make_shared<Node>(g,
    [r, c, block_size, N, &x, &a, &b]
    (const tbb::flow::continue_msg &msg) {
      int i_start = r*block_size, i_end = i_start + block_size;
      int j_start = c*block_size, j_max = j_start + block_size - 1;
      for (int i = i_start; i < i_end; ++i) {
        int j_end = (i <= j_max) ? i : j_max + 1;
        for (int j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
      return msg;
    }
  );
}
```

Рис. 3.17 ❖ Реализация функции `createNode`

Объектам `continue_node`, создаваемым в `createNode`, передается лямбда-выражение, которое инкапсулирует вложенный цикл из версии прямой подстановки с блоками итераций, показанной на рис. 3.12. Поскольку данные по ребрам графа зависимостей не передаются, каждый узел должен получать данные из разделяемой памяти с помощью указателей, захваченных лямбда-выражением. На рис. 3.17 узел захватывает по значению целые числа `r`, `c`, `N` и `block_size`, а также ссылки на векторы `x`, `a` и `b`.

Функция `addEdges` на рис. 3.18 вызывает `make_edge`, чтобы соединить каждый узел с правым нижним соседом, поскольку они должны ждать завершения работы этого узла. Когда вложенный цикл на рис. 3.16 закончится, будет полностью построен граф зависимостей, похожий на тот, что показан на рис. 3.13.

```
void addEdges(std::vector<NodePtr> &nodes, int r, int c,
              int block_size, int num_blocks) {
  NodePtr np = nodes[r*num_blocks + c];
  if (c + 1 < num_blocks && r != c)
    tbb::flow::make_edge(*np, *nodes[r*num_blocks + c + 1]);
  if (r + 1 < num_blocks)
    tbb::flow::make_edge(*np, *nodes[(r + 1)*num_blocks + c]);
}
```

Рис. 3.18 ❖ Реализация функции `addEdges`

Как видно по рис. 3.16, после построения графа мы запускаем его, посылая сообщение `continue_msg` левому верхнему узлу. Всякий узел `continue_node`, не имеющий предшественников, выполняется сразу по получении сообщения.

Отправка сообщения левому верхнему узлу запускает граф. После этого мы должны дождаться его завершения, вызвав `g.wait_for_all()`.

Оценка масштабируемости графа зависимостей

Ограничения производительности, действующие для графов потока данных, применимы и к графам зависимостей. Но, поскольку граф зависимостей должен быть ациклическим, установить верхнюю границу масштабируемости для него проще. В этом обсуждении мы пользуемся обозначениями, введенными в проекте Cilk, разрабатываемом в MTИ (см., например, Blumofe, Joerg, Kuszmaul, Leiserson, Randall, and Zhou «Cilk: An Efficient Multithreaded Runtime System» в издании Proceedings of the Principles and Practice of Parallel Programming, 1995).

Обозначим T_1 суммарное время выполнения всех узлов графа; 1 означает, что это время, необходимое для выполнения графа, если имеется всего один поток. И обозначим T_∞ время выполнения узлов на критическом (самом длинном) пути, поскольку это минимально возможное время выполнения даже при наличии бесконечного числа потоков. Тогда максимальное теоретически возможное ускорение в результате распараллеливания графа зависимостей равно T_1/T_∞. При выполнении на машине с P процессорами время выполнения никогда не будет меньше максимума из T_1/P и T_∞.

Например, предположим для простоты, что выполнение каждого узла на рис. 3.13(а) занимает одинаковое время, и обозначим его t_n. Всего в графе 36 узлов (количество строк * количество столбцов), поэтому $T_1 = 36t_n$. Самый длинный путь от узла 0,0 к узлу 7,7 содержит 15 узлов (количество строк + количество столбцов – 1), так что в этом случае $T_\infty = 15t_n$. Даже при наличии бесконечного числа процессоров узлы на критическом пути должны выполняться строго по порядку, без перекрытия. Поэтому максимальное ускорение, достижимое для этого небольшого графа 8×8, равно $36t_n/15t_n = 2.4$. Но если бы количество уравнений было больше и матрица имела бы размер, скажем, 512×512, то было бы $512 \times 512 = 131\,328$ узлов, на критическом пути находилось бы $512 + 512 - 1 = 1023$ узла, и максимальное ускорение было бы равно $131\,328/1023 \approx 128$.

Рассматривая распараллеливание последовательного приложения с помощью графа зависимостей, постарайтесь по возможности профилировать последовательный код, оценить время работы каждого участка, который станет узлом, и длину критического пути. Тогда с помощью простого описанного выше вычисления вы сможете оценить верхнюю границу ускорения.

Дополнительные сведения о потоковых графах в TBB

У потоковых графов в TBB богатый набор узлов и интерфейсов, и в этой главе мы видели только вершину айсберга. В главе 17 мы рассмотрим API подробнее и ответим на некоторые важные вопросы.

- Как управлять потреблением ресурсов в потоковом графе?
- Когда следует использовать буферизацию?

- Существуют ли антипаттерны, которых нужно избегать?
- Существуют ли эффективные паттерны, которые нужно взять на вооружение?

Кроме того, потоковые графы допускают асинхронное выполнение и обладают гетерогенными свойствами, все это мы рассмотрим в главах 18 и 19.

Резюме

В этой главе мы изучали классы и функции в пространстве имен `tbb::flow`, которые позволяют создавать графы потока данных и графы зависимостей. Сначала мы обсудили, почему полезно выражать параллелизм в виде графов. Затем были рассмотрены основы интерфейса потоковых графов в TBB, в т. ч. приведен краткий обзор различных категорий узлов. Далее мы шаг за шагом построили небольшой граф потока данных для применения стереоскопического эффекта к паре изображений. Затем обсудили, как узлы графа отображаются на задачи TBB и каковы ограничения производительности потоковых графов. Мы рассмотрели также графы зависимостей, частный случай графов потока данных, в котором ребра служат для информирования о зависимостях, а не для передачи сообщений с данными. Мы также реализовали метод прямой подстановки в виде графа зависимостей и обсудили максимально достижимое ускорение. Наконец, обозначили ряд важных дополнительных тем, которые будут раскрыты в последующих главах.

Глава 4

TBB и параллельные алгоритмы стандартной библиотеки шаблонов C++

Чтобы эффективно использовать библиотеку TBB, важно понимать, как она поддерживает и дополняет стандарт C++. В этой главе мы обсудим три аспекта связей TBB со стандартом C++.

1. На протяжении истории библиотеки TBB она часто включала относящиеся к параллелизму средства, только-только появившиеся в стандарте C++. Включение их в TBB позволяет разработчикам начать пользоваться ими еще до поддержки всеми компиляторами. Поэтому все готовые дистрибутивы TBB сейчас включают Intel'евскую реализацию параллельных алгоритмов из стандартной библиотеки шаблонов C++ (STL). В ней для реализации многопоточности применяются задачи TBB, а для векторизации – команды SIMD. Обсуждению библиотеки Parallel STL в основном и посвящена эта глава.
2. Библиотека TBB предлагает также средства, которые не включены в стандарт C++, но упрощают выражение параллелизма разработчикам. Примерами могут служить обобщенные параллельные алгоритмы и потоковые графы. В этой главе мы обсудим нестандартные итераторы, включенные в TBB с целью расширить применимость алгоритмов из Parallel STL.
3. Наконец, по ходу этой главы мы будем отмечать, что некоторые добавления в стандарт C++ могут заменить собой определенные средства TBB. Но при этом подчеркнем, что в обозримом будущем TBB будет представлять функциональность, которая в стандарт не войдет. К возможностям TBB, обладающим непреходящей ценностью, можно отнести планировщик задач с заимствованием работ, потокобезопасные контейнеры, API потокового графа и масштабируемые распределители памяти.

Какое отношение библиотека STL имеет к этой книге?

Так ли надо было включать главу о добавлениях в стандартную библиотеку шаблонов C++ в книгу, посвященную TBB? Да, очень надо! TBB – написанная

на C++ библиотека для организации параллелизма, но она существует не в вакууме. Мы должны понимать, как она соотносится со стандартом C++.

Политики выполнения, обсуждаемые в этой главе, чем-то похожи на параллельные алгоритмы TBB, рассмотренные в главе 2, поскольку позволяют выразить тот факт, что алгоритм безопасно выполнять параллельно, но ***не*** предписывают никаких деталей реализации. Если мы хотим сочетать алгоритмы TBB и Parallel STL в одном приложении, сохранив эффективный компонуемый параллелизм (см. главу 9), то к нашим услугам реализация Parallel STL, в которой TBB служит движком параллельного выполнения! Поэтому, говоря о политиках параллельного выполнения в этой главе, мы акцентируем внимание на реализациях, основанных на TBB. В этом случае Parallel STL становится просто еще одним способом использования TBB в своем коде.

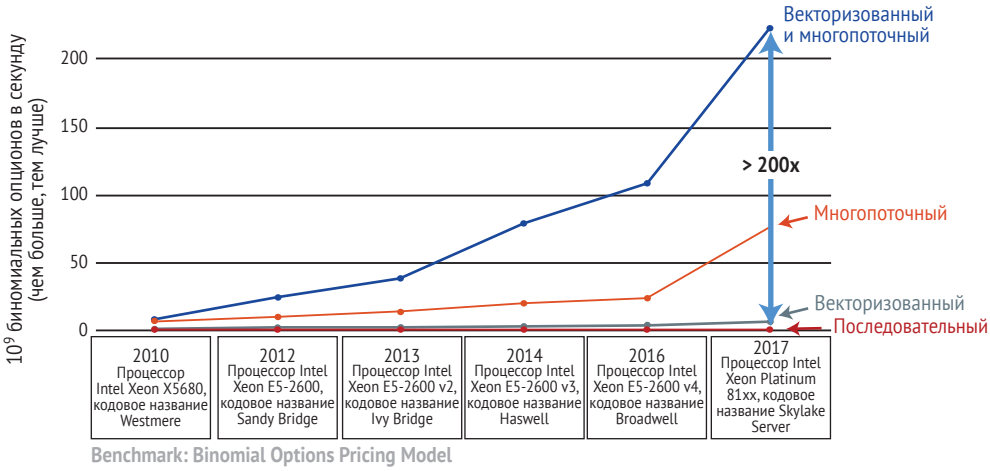
Обсуждая рис. 1.3 в главе 1, мы отметили, что во многих приложениях доступно несколько уровней параллелизма, в т. ч. уровень SIMD, который лучше всего подходит для векторных процессоров. Использование этого уровня параллелизма может оказаться критически важным, что наглядно демонстрируют результаты измерения производительности приложения для расчета биномиальных опционов на рис. 4.1. Но если векторный параллелизм используется сам по себе, то ускорение будет незначительным, т. к. оно ограничено шириной вектора. Однако рис. 4.1 напоминает, что не следует забывать о мультипликативном эффекте одновременного использования параллелизма задач и векторного параллелизма.

В главе 1 мы реализовали пример, где на верхнем уровне использовался уровень потоковых графов для организации многопоточности, в узлах графа – обобщенные параллельные алгоритмы TBB для повышения степени многопоточности, а в телах параллельных алгоритмов – векторные политики для достижения векторизации. Сочетая TBB с Parallel STL и ее политиками выполнения, мы получаем не только допускающие композицию уровни обмена сообщениями и разветвления–соединения, но и доступ к уровню SIMD.

Именно по этим причинам политики выполнения в библиотеке STL являются важной частью нашего знакомства с TBB!

TBB и стандарт C++

Команда, разрабатывающая TBB, – активный сторонник поддержки многопоточности в самом языке C++. На самом деле TBB часто включала средства параллелизма, спроектированные в соответствии с предложениями, направленными в комитет по стандартизации C++, чтобы разработчики могли перейти на эти интерфейсы еще до их широкой поддержки основными компиляторами. Примером является класс `std::thread`. Разработчики TBB признали важность `std::thread` и предложили другим разработчикам путь миграции до включения этого класса во все стандартные библиотеки C++, поместив это средство прямо в пространство имен `std`. В настоящее время реализация `std::thread` совпадает с платформенной реализацией, если таковая имеется, и обращается к собственной, лишь если на данной платформе в стандартной библиотеке нужная реализация отсутствует. Похожие истории можно рассказать о таких вошедших в стандарт средствах C++, как атомарные переменные, мьютексы и условные переменные.



Результаты эталонных тестов получены до реализации недавних исправлений программного обеспечения и обновлений прошивки, призванных закрыть уязвимости, получившие название Spectre и Meltdown. Реализация этих обновлений может сделать результаты неприменимыми к вашему устройству или системе.
Возможно, что оптимизация программного обеспечения и рабочих нагрузок с целью повышения производительности распространяется только на микропроцессоры Intel. Тесты производительности, такие как SYSmark и MobileMark, проводились с использованием конкретных компьютерных систем, компонентов, программного обеспечения, операций и функций. Изменение любого из этих факторов может привести к изменению результатов. Принимая решение о покупке, обратитесь также к другим источникам информации и результатов тестов производительности для более объективной оценки, в том числе к сравнению производительности данного продукта с другими. Для получения более полной информации посетите страницу Performance Benchmark Test Disclosure

Рис. 4.1 ❖ Производительность приложения биномиальной модели ценообразования опционов в четырех режимах выполнения: последовательный, векторизованный, многопоточный, векторизованный и многопоточный

Аналогия для осмысления политик выполнения в Parallel STL

Чтобы было проще рассуждать о различных политиках выполнения в библиотеке Parallel STL, мы предлагаем представить себе многополосную автостраду, показанную на рис. 4.2. Как и большинство аналогий, эта не совершенна, но все-таки поможет оценить преимущества различных политик.

Мы можем рассматривать каждую полосу как поток выполнения, человека – как подлежащую выполнению операцию (т. е. человеку нужно добраться из точки A в точку B), автомобиль – как процессорное ядро, а каждое место в автомобиле – как элемент в (векторном) регистре. При *последовательном выполнении* имеется всего одна полоса движения (один поток), и у каждого человека своя машина (векторные устройства не используются). Даже если несколько людей едут по одному маршруту, у каждого из них отдельная машина, и все движутся по одной полосе.

При *многопоточном выполнении* используется несколько полос движения (более одного потока выполнения). Теперь в единицу времени можно выполнить больше задач, но карпулинг (совместное использование автомобиля) по-прежнему запрещен. Даже если несколько человек выезжают из одного места и направляются в одно место, все равно каждый путешествует на своей ма-

шине. Мы стали более эффективно использовать автостраду, но автомобили (ядра) все еще используются неэффективно.

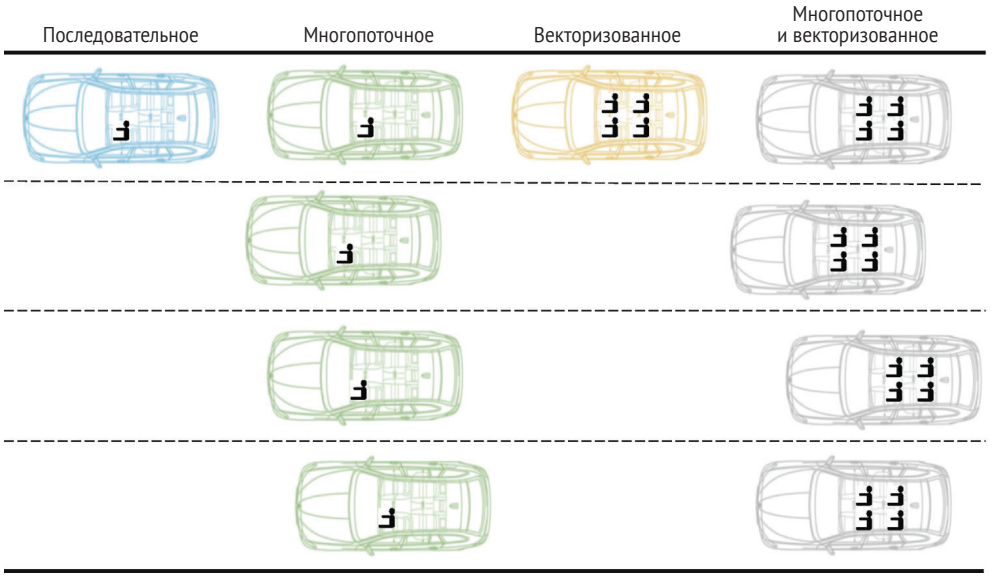


Рис. 4.2 ❖ Аналогия политик выполнения в Parallel STL с многополосной автострадой

Векторизованное выполнение можно уподобить карпулингу. Если несколько человек едут по одному маршруту, то они могут совместно использовать один автомобиль. Многие современные процессоры поддерживают векторные команды, например расширения SSE и AVX в процессорах Intel. Если мы не используем векторные команды, то процессор остается недозагруженным. Векторное устройство в ядре может применять одну операцию к нескольким элементам данных одновременно. Данные в векторных регистрах – все равно что люди в автомобиле, они движутся по одному маршруту.

Наконец, многопоточное и векторизованное выполнение можно сравнить с использованием всех полос автострады (всех ядер) в сочетании с карпулингом (использованием векторных устройств в каждом ядре).

Простой пример – алгоритм std::for_each

Итак, теперь у нас есть общее представление о политиках выполнения, но прежде чем с головой окунуться в технические детали, применим функцию `void f(float &e)` ко всем элементам вектора `v`, как показано на рис. 4.3(a). Воспользовавшись алгоритмом `std::for_each`, входящим в библиотеку STL, мы можем сделать то же самое, как показано на рис. 4.3(b). Как и цикл for, основанный на диапазоне, алгоритм `for_each` перебирает элементы вектора от `v.begin()`

до v.end() и для каждого вызывает лямбда-выражение. Это последовательное поведение for_each, подразумеваемое по умолчанию.

```
for (auto &i : v ) {
  f(i);
}
(a) a for loop

std::for_each(v.begin(), v.end(),
 [](float &i) {
   f(i);
 }
);
(b) std::for_each and no execution policy

std::for_each(pstl::execution::seq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(c) std::for_each and seq execution policy

std::for_each(pstl::execution::unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(d) std::for_each and unseq execution policy

std::for_each(pstl::execution::par, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(e) std::for_each and par execution policy

std::for_each(pstl::execution::par_unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
(f) std::for_each and par_unseq execution policy
```

Рис. 4.3 ❖ Простой цикл, реализованный с помощью std::for_each с различными политиками выполнения, определенный в библиотеке Parallel STL

Parallel STL позволяет информировать библиотеку о том, что эту семантику можно ослабить, чтобы задействовать параллелизм, или, как показано на рис. 4.3(c), явно указать, что нам нужна последовательная семантика. При работе с реализацией Parallel STL от компании Intel необходимо включать в код заголовочные файлы как для алгоритмов, так и для политик выполнения:

```
#include <pstl/execution>
#include <pstl/algorithm>
```

В C++17, если вообще не указывать политику выполнения или передать объект последовательной политики `seq`, то поведение будет таким же, как по умолчанию: как будто лямбда-выражение вызывается для каждого элемента вектора по порядку. Мы говорим «как будто», потому что оборудованию и компилятору разрешено распараллеливать алгоритм, но только если это будет незаметно соблюдающей стандарт программе.

Мощь Parallel STL проистекает из политик, ослабляющих это ограничение последовательного выполнения. Если мы хотим, чтобы операции можно было перекрывать (совмещать во времени), или векторизовать в одном потоке выполнения, то следует указать объект политики `unseq`, как показано на рис. 4.3(d). Библиотека может совмещать операции, например воспользовавшись такими SIMD-расширениями, как SSE или AVX. На рис. 4.4 это поведение изображено парными прямоугольниками, показывающими, что операции могут выполняться одновременно векторным устройством. Политика `unseq` допускает «карпулинг».

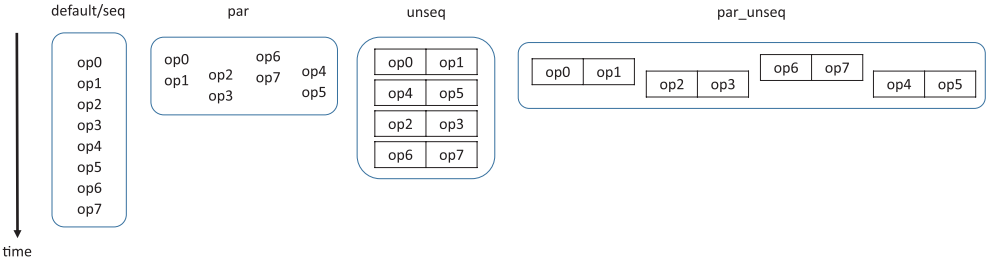


Рис. 4.4 ❖ Применение операций с различными политиками выполнения

На рис. 4.3(e) мы говорим библиотеке, что эту функцию безопасно выполнять для всех элементов вектора, применяя несколько потоков выполнения, – для этого задается параллельная политика `par`. Как показано на рис. 4.4, политика `par` позволяет распределять операции между несколькими потоками, но в каждом потоке операции не перекрываются (т. е. не векторизованы). В нашей аналогии с автострадой это означает, что используются все полосы, но без карпулинга.

Наконец, на рис. 4.3(f) использование объекта `par_unseq` сообщает приложению, что переданное лямбда-выражение можно распараллелить и векторизовать. На рис. 4.4 видно, что в таком режиме работает несколько потоков, и операции в каждом потоке перекрываются. Вот теперь мы в полной мере задействовали все имеющиеся ядра и эффективно используем каждое ядро, загрузив его векторные устройства.

На практике использовать политики выполнения следует с осторожностью. Как и при работе с обобщенными параллельными алгоритмами, используя политики для ослаблений ограничений на порядок выполнения STL-алгоритма,

мы заверяем библиотеку в том, что такое ослабление допустимо и выгодно. Библиотека верит нам на слово. Но не гарантирует, что в результате выбора некоторой политики выполнения производительность не упадет.

На рис. 4.3 есть еще один момент, на который следует обратить внимание: сами STL-алгоритмы находятся в пространстве имен `std`, но политики выполнения, определенные в библиотеке Intel Parallel STL, находятся в пространстве имен `pstl::execution`. Если вы работаете с компилятором, полностью совместимым со стандартом C++17, и указываете политики выполнения в пространстве имен `std::execution`, то может быть выбрана их реализация, не использующая TBB.

Какие алгоритмы предоставляет реализация Parallel STL?

В библиотеку STL включены преимущественно операции, применяемые к последовательностям. Есть несколько «белых ворон», например `std::min` и `std::max`, которые могут применяться и к значениям, но большинство алгоритмов, например `std::for_each`, `std::find`, `std::transform`, `std::copy` и `std::sort`, ориентированы на последовательности элементов. Этот упор на последовательности удобен, когда нам нужно работать с контейнерами, которые поддерживают итераторы, но только обременяет, когда нужно выразить что-то, не ориентированное на контейнеры. Ниже в этой главе мы увидим, что иногда можно применить нестандартный подход и использовать специальные итераторы, чтобы заставить некоторые алгоритмы работать как обычные циклы.

Объяснять, что делает каждый алгоритм STL, не входит в задачу этой главы, да и книги в целом. Существуют книги, посвященные исключительно библиотеке STL, например Nicolai Josuttis «C++ Standard Library: A Tutorial and Reference» (Addison-Wesley Professional). В этой главе мы будем заниматься только тем, что политики выполнения, появившиеся в стандарте C++17, означают в этих алгоритмах и как их можно использовать совместно с TBB.

У большинства STL-алгоритмов, описанных в стандарте C++, имеются перегруженные варианты (добавленные в версии C++17), принимающие политики выполнения. Кроме того, было добавлено несколько новых алгоритмов либо потому, что они особенно полезны в параллельных программах, либо потому, что комитет хотел избежать изменения семантики. Какие именно алгоритмы поддерживают политики выполнения, можно узнать, заглянув в стандарт или на сайты, например http://en.cppreference.com/w/cpp/algorithm.

Как получить и использовать копию библиотеки STL, в которой применяется TBB

Подробные инструкции по скачиванию и установке библиотеки Parallel STL от Intel приведены в главе 1. Если вы скачали и установили уже откомпилированную версию TBB 2018 обновление 5 или более позднюю, не важно – по ком-

мерческой лицензии, купленной у Intel, или в виде двоичного дистрибутива с GitHub, – то в ней есть и библиотека Parallel STL. Эта библиотека входит во все готовые пакеты TBB.

Но если вы хотите собрать библиотеку TBB из исходных файлов, скачанных с GitHub, то скачивать исходный код Parallel STL придется отдельно, поскольку исходный код обеих библиотек хранится в разных репозиториях: https://github.com/intel/tbb и https://github.com/intel/parallelstl.

Как мы уже видели, Parallel STL поддерживает несколько политик выполнения: одни предназначены для параллельного выполнения, другие – для векторизованного, третьи – для сочетания того и другого. В версии Parallel STL от Intel параллелизм реализован с применением TBB, а векторизация – с применением конструкций из OpenMP 4.0 SIMD. Чтобы получить максимальную отдачу от библиотеки Intel Parallel STL, необходим компилятор C++ с поддержкой C++11 и конструкций OpenMP 4.0 SIMD – и, разумеется, понадобится еще TBB. Мы настоятельно рекомендуем использовать компилятор Intel, входящий в состав любой версии Intel Parallel Studio XE 2018 или более поздней. Эти компиляторы не только включают TBB и поддерживают OpenMP 4.0 SIMD, но и содержат оптимизации, специально разработанные для повышения производительности некоторых STL-алгоритмов с политиками выполнения `unseq` или `par_unseq`.

Для сборки приложения, использующего Parallel STL, из командной строки необходимо задать переменные среды для компиляции и компоновки. Если вы уже установили Intel Parallel Studio XE, то это можно сделать, выполнив скрипт с именем вида `compilervars.{sh|csh|bat}`. Если установлена только Parallel STL, то для задания переменных среды нужно выполнить один из скриптов `pstlvars.{sh|csh|bat}` в каталоге `<pstl_install_dir>/{linux|mac|windows}/pstl/bin`. Дополнительные инструкции приведены в главе 1.

Алгоритмы в библиотеке Intel Parallel STL

Intel Parallel STL еще не поддерживает все политики выполнения для каждого STL-алгоритма. Актуальный список предоставляемых библиотекой алгоритмов с указанием поддерживаемых политик см. по адресу https://software.intel.com/en-us/get-started-with-pstl.

На рис. 4.5 показаны алгоритмы и политики выполнения, поддерживаемые на момент работы над этой книгой.

На рис. 4.6 показаны политики, поддерживаемые библиотекой Intel Parallel STL, включая как те, что уже вошли в стандарт C++17, так и те, что предложены для включения в будущий стандарт. Политики C++17 позволяют выбирать последовательное выполнение (`seq`), параллельное выполнение с использованием TBB (`par`) или параллельное выполнение с применением TBB и векторизацией (`par_unseq`). Политика непоследовательного выполнения (`unseq`) выбирает реализацию с одной только векторизацией.

Категории алгоритмов	
Немодифицирующие операции с последовательностями	**заголовок: <algorithm>**
`adjacent_find, all_of, any_of, count, count_if, find, find_end, find_first_of, find_if, find_if_not, for_each, for_each_n, mismatch, none_of, search, search_n`	
Модифицирующие операции с последовательностями	**заголовок: <algorithm>**
`copy, copy_if, copy_n, fill, fill_n, generate, generate_n, move, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, shuffle, swap_ranges, transform, unique*, unique_copy`	
Операция разбиения	**заголовок: <algorithm>**
`is_partitioned, partition*, partition_copy, stable_partition*`	
Операции сортировки	**заголовок: <algorithm>**
`is_sorted, is_sorted_until, nth_element*, partial_sort*, partial_sort_copy*, sort*, stable_sort*`	
Операции над отсортированными диапазонами	**заголовок: <algorithm>**
`includes*, inplace_merge*, merge*, set_difference*, set_intersection*, set_symmetric_difference*, set_union*`	
Операции с пирамидой	**заголовок: <algorithm>**
`is_heap, is_heap_until`	
Операции минимума и максимума	**заголовок: <algorithm>**
`max_element, min_element, minmax_element`	
Операции сравнения	**заголовок: <algorithm>**
`equal, lexicographical_compare`	
Числовые операции	**заголовок: <numeric>**
`adjacent_difference, exclusive_scan, inclusive_scan, reduce, transform_exclusive_scan, transform_inclusive_scan, transform_reduce`	
Операции с неинициализированной памятью	**заголовок: <memory>**
`uninitialized_copy, uninitialized_copy_n, uninitialized_default_costruct, uninitialized_default_costruct_n, uninitialized_fill, uninitialized_fill_n, uninitialized_move, uninitialized_move_n, uninitialized_value_construct, uninitialized_value_construct_n`	

* Поддерживает потоковую обработку, но не выполнение SIMD.

Рис. 4.5 ❖ Алгоритмы, поддерживающие политики выполнения в библиотеке Intel Parallel STL, по состоянию на 1 января 2019 года. Впоследствии могут быть добавлены новые алгоритмы и политики. См. обновления по адресу https://software.intel.com/en-us/get-started-with-pstl

Политика	Стандарт	Класс	Глобальный объект
Последовательная	C++17	`sequenced_policy`	`seq`
Параллельная	C++17	`parallel_policy`	`par`
Параллельная + непоследовательная	C++17	`parallel_unsequenced_policy`	`par_unseq`
Непоследовательная	Предложение	`unsequenced_policy`	`unseq`

Рис. 4.6 ❖ Политики выполнения, поддерживаемые библиотекой Intel Parallel STL

Нестандартные итераторы открывают дополнительные способы использования

Ранее в этой главе мы рассмотрели простое применение алгоритма `std::for_each` и показали, как в нем можно использовать различные политики выполнения. Пример для политики `par_unseq` на рис. 4.3(f) выглядел так:

```
std::for_each(pstl::execution::par_unseq, v.begin(), v.end(),
  [](float &i) {
    f(i);
  }
);
```

На первый взгляд кажется, что алгоритм довольно сильно ограничен, поскольку посещает элементы по порядку и к каждому применяет унарную функцию. И действительно, если применять его к контейнеру ожидаемым образом, то ограничения налицо. Например, он не принимает диапазон, как алгоритм `parallel_for` из TBB.

Однако C++ – мощный язык, и мы можем раздвинуть пределы применимости STL-алгоритмов, проявив творческую смекалку. Как обсуждалось в главе 2, итератор – это объект, указывающий на некоторый элемент в диапазоне элементов, причем для него определены операторы, позволяющие этот диапазон обходить. Существуют различные категории итераторов: *однонаправленные*, *двунаправленные* и *произвольного доступа*. Многие стандартные контейнеры C++ предоставляют функции `begin` и `end`, которые возвращают итераторы, позволяющие обойти элементы контейнера. Один из распространенных способов расширить сферу применения STL – воспользоваться *нестандартными* итераторами. Это классы, реализующие интерфейс итератора, но не включенные в библиотеку STL.

В библиотеку TBB включено три нестандартных итератора, чтобы было удобнее использовать STL-алгоритмы. Их типы описаны на рис. 4.7 и становятся доступны, если включить заголовок `iterators.h` или всеобъемлющий заголовок `tbb.h`.

Тип	Описание
`template<typename IntType>` `class counting_iterator;`	Итератор произвольного доступа, хранящий внутренний счетчик, который изменяется в зависимости от арифметических операций, выполняемых над итератором. Функция `operator*()` возвращает текущее значение счетчика. Обычно используется для получения индекса в контейнере или группе контейнеров
`template<typename… Types>` `class zip_iterator;`	Итератор произвольного доступа, который обходит сразу несколько итераторов. Функция `operator*()` возвращает кортеж разыменованных значений отдельных итераторов

`template<typename UnaryFunc,`
`typename Iter>`
`class transform_iterator;`
	Итератор произвольного доступа, который применяет преобразование к последовательности. Функция `operator*()` применяет унарную функцию преобразования к значению, возвращенному в результате разыменования базового итератора

Рис. 4.7 ❖ Классы нестандартных итераторов в TBB

В качестве примера мы можем передать нестандартные итераторы алгоритму `std::for_each`, так чтобы его работа больше напоминала обычный цикл for. Рассмотрим простой цикл, показанный на рис. 4.8(a). Он записывает выражение `a[i]+b[i]*b[i]` в `a[i]` для каждого `i` в диапазоне `[0,n)`.

```
std::vector<float> a(n, 1.0), b(n, 3.0);

for (int i = 0; i < n; ++i) {
  a[i] = a[i] + b[i]*b[i];
}
```

(a) Исходный последовательный цикл for

```
std::for_each(tbb::counting_iterator<int>(0),
              tbb::counting_iterator<int>(n),
  [&a, &b](int i) {
    a[i] = a[i] + b[i]*b[i];
  });
```

(b) Использование tbb::counting_iterator

```
auto begin = tbb::make_zip_iterator(a.begin(), b.begin());
auto end = tbb::make_zip_iterator(a.end(), b.end());

std::for_each(begin, end,
  [](const std::tuple<float&, float&>& v) {
    float &a = std::get<0>(v);
    float b = std::get<1>(v);
    a = a + b*b;
  });
```

(c) Использование tbb::zip_iterator

```
auto zbegin = tbb::make_zip_iterator(a.begin(), b.begin());
auto zend = tbb::make_zip_iterator(a.end(), b.end());
auto square_b = [](const std::tuple<float &, float &>& v) {
  float b = std::get<1>(v);
  return std::tuple<float &, float>(std::get<0>(v), b*b);
};
auto begin = tbb::make_transform_iterator(zbegin, square_b);
auto end = tbb::make_transform_iterator(zend, square_b);

std::for_each(begin, end,
  [](const std::tuple<float&, float>& v) {
    float &a = std::get<0>(v);
    a = a + std::get<1>(v);
  });
```

(d) Использование tbb::transform_iterator

Рис. 4.8 ❖ Использование `std::for_each` с нестандартными итераторами

На рис. 4.8(b) класс `counting_iterator` использован для создания некоего аналога диапазона. Лямбда-выражению в алгоритме `for_each` передаются целые числа от 0 до `n-1`. Хотя `for_each` по-прежнему обходит только одну последовательность, эти значения применяются как индексы сразу двух векторов, `a` и `b`.

На рис. 4.8(c) класс `zip_iterator` используется для одновременного обхода векторов `a` и `b`. Библиотека TBB предоставляет функцию `make_zip_iterator`, упрощающую создание итераторов:

```
template<typename... T>
zip_iterator<T...> make_zip_iterator(T&&... args);
```

На рис. 4.8(c) мы по-прежнему указываем только одну последовательность при вызове `for_each`. Но теперь лямбда-выражению в качестве аргумента передается кортеж `std::tuple`, содержащий ссылки на значения типа `float`, по одному из каждого вектора.

Наконец, на рис. 4.8(d) добавлено использование класса `transform_iterator`. Сначала мы объединяем обе последовательности – векторы `a` и `b` – в одну, воспользовавшись классом `zip_iterator`, как на рис. 4.8(c). Но дополнительно создаем еще одно лямбда-выражение, которое присваиваем переменной `square_b`. Это лямбда-выражение будет использовано для преобразования кортежа `std::tuple` ссылок на `float`, полученных в результате разыменования `zip_iterator`. Мы передаем его при вызовах функции `make_tranform_iterator`:

```
template<typename UnaryFunc, typename Iter>
transform_iterator<UnaryFunc, Iter>
make_transform_iterator(Iter it, UnaryFunc unary_func);
```

В момент разыменования объектов `transform_iterator` на рис. 4.8(d) они получают элемент от итератора `zip_iterator`, возводят в квадрат второй элемент кортежа и создают новый кортеж `std::tuple`, который содержит ссылку на `float` из `a` и квадрат значения из `b`. Аргумент, переданный лямбда-выражению в алгоритме `for_each`, включает уже возведенное в квадрат значение, поэтому функции не нужно вычислять `b[i]*b[i]`.

Поскольку нестандартные итераторы наподобие приведенных на рис. 4.7 очень полезны, они встречаются не только в библиотеке TBB, но и в других библиотеках, например Boost C++ (www.boost.org) и `Thrust` (https://thrust.github.io/doc/group__fancyiterator.html). Но в самой библиотеке STL их пока нет.

Некоторые наиболее полезные алгоритмы

Покончив с предисловием, перейдем к подробному обсуждению наиболее полезных и общих алгоритмов в библиотеке Parallel STL: `for_each`, `transform`, `reduce` и `transform_reduce`. При обсуждении каждого алгоритма мы отмечаем аналоги из числа обобщенных алгоритмов TBB. Преимущество интерфейсов Parallel STL над интерфейсами TBB состоит в том, что Parallel STL – часть стандарта C++. А их недостаток – в меньшей выразительности и более бедных средствах настройки. Мы будем отмечать недостатки в описании каждого алгоритма.

std::for_each, std::for_each_n

В этой главе мы уже много говорили об алгоритме `for_each`. Помимо `for_each`, Parallel STL предоставляет алгоритм `for_each_n`, который посещает только первые `n` элементов. У обоих алгоритмов несколько интерфейсов, ниже показаны те, что принимают политику выполнения:

```
template<class ExecutionPolicy, class ForwardIt,
         class UnaryFunction>
void for_each( ExecutionPolicy&& policy,
               ForwardIt first, ForwardIt last,
               UnaryFunction f);

template<class ExecutionPolicy, class ForwardIt, class Size,
         class UnaryFunction>
ForwardIt for_each_n(ExecutionPolicy&& policy,
                     ForwardIt first, Size n, UnaryFunction f);
```

В сочетании с нестандартными итераторами `for_each` может быть очень выразительным, что и было продемонстрировано на рис. 4.8. Например, мы можем взять простой пример умножения матриц из главы 2 и переписать его, как показано на рис. 4.9, с помощью класса `counting_iterator`.

```
std::for_each(
  /* policy */ pstl::execution::par,
  /* first */  tbb::counting_iterator<int>(0),
  /* last */   tbb::counting_iterator<int>(M),
  [&a, &b, &c, M](int i) {
    for (int j = 0; j < M; ++j) {
      int c_index = i*M + j;
      for (int k = 0; k < M; ++k) {
        c[c_index] += a[i*M + k] * b[k*M + j];
      }
    }
  }
);
```

Рис. 4.9 ❖ Использование `std::for_each` в сочетании с `tbb::counting_iterator` для создания параллельной версии умножения матриц

В версии STL, основанной на TBB, например Intel Parallel STL, политика `par` реализована с помощью `tbb::parallel_for`, поэтому производительности `std::for_each` и `tbb::parallel_for` на таком простом примере будут примерно одинаковы.

Разумеется, сразу возникает вопрос. Если `std::for_each` пользуется `tbb::parallel_for` для реализации политики `par`, но является стандартным интерфейсом и дает также доступ к другим политикам, то почему бы не использовать *только* `std::for_each`, а о `tbb::parallel_for` забыть?

К сожалению, так не получится. Не всегда код такой простой. Если нас интересует эффективная многопоточная реализация, то лучше использовать `tbb::parallel_for` напрямую. В главе 2 мы говорили, что даже в этом примере умножения матриц наша простая реализация не оптимальна. Во второй части книги мы обсудим важные средства оптимизации, имеющиеся в TBB и позволяющие настроить код. В главе 16 мы увидим, что эти средства дают значительный выигрыш в производительности. Увы, ни одним из них не удастся воспользоваться, если мы работаем с алгоритмом из Parallel STL. В стандартных интерфейсах для них просто нет места.

Если применить алгоритм из Parallel STL и указать стандартную политику, например `par`, `unseq` или `par_unseq`, то мы получим то, что выберет за нас реализация. В комитет по стандартизации C++ поданы предложения, в частности о включении исполнителей, которые когда-нибудь в будущем, возможно, снимут это ограничение. Но пока степень контроля над STL-алгоритмами невелика. При использовании обобщенных алгоритмов TBB, например `parallel_for`, у нас есть доступ к богатому набору средств оптимизации, описанному во второй части книги, в т. ч. разбивателям, различным типам блочных диапазонов, степени детализации, указаниям о привязке к процессорам, возможностям изоляции и т. д.

В простых случаях алгоритм из стандартной библиотеки Parallel STL может быть не хуже его аналога из TBB, но в более реалистичных сценариях TBB предоставляет гибкость и контроль, необходимые для получения вожделенной производительности.

std::transform

Еще один полезный алгоритм в Parallel STL – `transform`. Он применяет унарную операцию к элементам одной последовательности или бинарную операцию к элементам двух последовательностей и записывает результаты в одну выходную последовательность. Два интерфейса с поддержкой политик выполнения выглядят так:

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class UnaryOperation >
ForwardIt2 transform(ExecutionPolicy&& policy, ForwardIt1 first1,
                     ForwardIt1 last1, ForwardIt2 d_first,
                     UnaryOperation unary_op);

template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2,
          class ForwardIt3, class BinaryOperation >
ForwardIt3 transform(ExecutionPolicy&& policy, ForwardIt1 first1,
                     ForwardIt1 last1, ForwardIt2 first2,
                     ForwardIt3 d_first,
                     BinaryOperation binary_op);
```

На рис. 4.8 мы пользовались алгоритмом `for_each` и нестандартными итераторами для чтения двух векторов и записи результата `a[i] = a[i] + b[i]*b[i]` в выходной вектор. Это идеальная ситуация для применения `std::transform`, поскольку у `transform` имеется интерфейс, поддерживающий две входные последовательности и одну выходную, – см. рис. 4.10.

Как и в случае `std::for_each`, применимость этого алгоритма ограничена, поскольку имеется всего две входные последовательности и одна выходная. Цикл, который пишет в несколько выходных массивов или контейнеров, естественно выразить с помощью одного обращения к `transform` не получится. Конечно, это возможно – в C++ возможно почти все, – но потребует применения нестандартных итераторов, например `zip_iterator`, и довольно уродливого кода для доступа к нескольким контейнерам.

```
std::vector<float> a(n, 1.0), b(n, 3.0);

std::transform(pstl::execution::par_unseq,
  /* in1 range */ a.begin(), a.end(),
  /* in2 first */ b.begin(),
  /* out first */ a.begin(),
  [](float ae, float be) -> float {
    return ae + be*be;
  }
);
```

Рис. 4.10 ❖ Применение `std::transform` для сложения двух векторов

std::reduce

Мы обсуждали редукцию при рассмотрении алгоритма `tbb::parallel_reduce` в главе 2. Алгоритм `reduce` из Parallel STL производит редукцию элементов последовательности. Но, в отличие от `tbb::parallel_reduce`, он предоставляет только операцию редукции. В следующем разделе мы обсудим алгоритм `transform_reduce`, который больше напоминает `tbb::parallel_reduce`, т. к. предоставляет обе операции: преобразование и редукцию. Ниже показаны два интерфейса `std::reduce`, поддерживающих политики выполнения:

```
template<class ExecutionPolicy, class ForwardIt, class T>
T reduce(ExecutionPolicy&& policy,
         ForwardIt first, ForwardIt last, T init);

template<class ExecutionPolicy, class ForwardIt, class T,
         class BinaryOp>
T reduce(ExecutionPolicy&& policy, ForwardIt first,
         ForwardIt last, T init, BinaryOp binary_op);
```

Алгоритм `reduce` вычисляет обобщенную сумму элементов последовательности, зная бинарную операцию `binary_op` и нейтральный элемент `init`. В первом интерфейсе в роли `binary_op` выступает не входной параметр, а операция `std::plus<>`. Обобщенное суммирование – это редукция, при которой элементы можно группировать и переставлять в любом порядке, поскольку предполагается, что операция ассоциативна и коммутативна. Поэтому возникают все те проблемы с округлением чисел с плавающей точкой, которые обсуждались на врезке в главе 2.

Чтобы просуммировать элементы вектора, можно воспользоваться `std::reduce` с любой политикой выполнения, как показано на рис. 4.11.

```
std::vector<float> a(n, 1.0);

float sum = std::reduce(pstl::execution::seq, a.begin(), a.end());
sum += std::reduce(pstl::execution::unseq, a.begin(), a.end());
sum += std::reduce(pstl::execution::par, a.begin(), a.end());
sum += std::reduce(pstl::execution::par_unseq, a.begin(), a.end());
```

Рис. 4.11 ❖ Применение `std::reduce` для вычисления суммы элементов вектора четырьмя способами

std::transform_reduce

Как было сказано выше, `transform_reduce` похож на `tbb::parallel_reduce`, поскольку предоставляет две операции: преобразование и редукцию. Но, как и большинство STL-алгоритмов, он применим только к одной или двум входным последовательностям:

```
template<class ExecutionPolicy, class ForwardIt,
class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);

template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2,
         class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1,
                   ForwardIt2 first2, T init,
                   BinaryOp1 binary_op1, BinaryOp2 binary_op2);
```

С помощью `std::transform_reduce` можно реализовать важное и широко распространенное линейно-алгебраическое ядро – скалярное произведение. Для этой цели он применяется настолько часто, что существует специальный интерфейс, в котором по умолчанию используются операции `std::plus<>` (редукция) и `std::multiplies<>` (преобразование):

```
template<class ExecutionPolicy,
         class ForwardIt1, class ForwardIt2, class T>
T transform_reduce(ExecutionPolicy&& policy,
                   ForwardIt1 first1, ForwardIt1 last1,
                   ForwardIt2 first2, T init);
```

На рис. 4.12(a) приведен последовательный код для вычисления скалярного произведения двух векторов `a` и `b`. Мы можем использовать `std::transform_reduce` и передать свои собственные лямбда-выражения для обеих операций, как показано на рис. 4.12(b). Или, как на рис. 4.12(c), положиться на операции по умолчанию.

```
float v = 0.0;
for (int i = 0; i < n; ++i) {
  v += a[i]*b[i];
}
```

(a) последовательный код для вычисления скалярного произведения двух векторов

```
std::transform_reduce(pstl::execution::par,
  /* in1 range */ a.begin(), a.end(),
  /* in2 range */ b.begin(),
  /* init */ 0.0,
  /* op1, the reduce */
  [](float ae, float be) -> float {
    return ae + be;
  },
  /* op2, the transform */
  [](float ae, float be) -> float {
    return ae * be;
  }
);
```

(b) std::transform_reduce с указанием обеих операций

```
std::transform_reduce(pstl::execution::par,
  /* in1 range */ a.begin(), a.end(),
  /* in2 range */ b.begin(),
  /* init */ 0.0
);
```

(c) std::transform_reduce с операциями по умолчанию

Рис. 4.12 ❖ Применение `std::transform_reduce` для вычисления скалярного произведения

И снова, как и для других алгоритмов из Parallel STL, мы можем проявить смекалку и, воспользовавшись нестандартными итераторами типа `counting_iterator`, применить этот алгоритм не только для обработки элементов в контейнерах. Например, на основе `std::transform_reduce` переписать программу вычисления числа π, реализованную в главе 2 с помощью `tbb::parallel_reduce` (см. рис. 4.13).

У использования алгоритма из Parallel STL, например `std::transform_reduce`, вместо `tbb::parallel_reduce` имеются те же плюсы и минусы, что и у других описанных выше алгоритмов. Стандартизованный интерфейс делает программу более переносимой. Но не позволяет воспользоваться средствами оптимизации, описанными во второй части книги.

```
float fig_4_13() {
  constexpr float dx = 1.0 / num_intervals;
  float sum = std::transform_reduce(
    /* policy */ std::execution::par_unseq,
    /* first */ tbb::counting_iterator<int>(0),
    /* last */  tbb::counting_iterator<int>(num_intervals),
    /* init = */ 0.0,
    /* reduce */
    [](float x, float y) -> float {
      return x + y;
    },
    /* transform */
    [=](int i) -> float {
      float x = (i + 0.5)*dx;
      float h = sqrt(1 - x*x);
      return h*dx;
    }
  );
  return 4 * sum;
}
```

Рис. 4.13. Применение `std::transform_reduce` для вычисления числа π

Политики выполнения в деталях

Политики выполнения в Parallel STL позволяют выразить ограничения, которые мы хотим применить к упорядочению операций во время выполнения STL-алгоритма. Стандартный набор политик возник не на пустом месте – он отражает ослабленные ограничения, необходимые для выполнения эффективного параллельного (многопоточного) или векторизованного (SIMD) кода.

Если вам достаточно интерпретировать `sequenced_policy` как последовательное выполнение, `parallel_policy` – как параллельное выполнение, `unsequenced_policy` – как векторизованное выполнение, а `parallel_unsequenced_policy` – как параллельное и векторизованное выполнение, то можете пропустить этот раздел. Но если вы хотите разобраться в тонкостях политик, то читайте дальше.

sequenced_policy

Политика `sequenced_policy` означает, что выполнение алгоритма *выглядит так, будто* (1) все используемые в нем *функции доступа к элементам* вызываются в том же потоке, в котором вызван алгоритм, и (2) вызовы функций доступа к элементам не перемежаются (т. е. строго упорядочены по времени в данном потоке). Функцией доступа к элементу называется любая вызываемая алгоритмом функция, которая обращается к элементам, например функции в итераторах, функции сравнения и обмена и любые другие предоставленные пользователем функции, применяемые к элементам. Как уже было сказано, мы говорим «так, будто», потому что оборудованию и компилятору разрешено нарушать эти правила, при условии что это не видно отвечающей стандарту программе.

Отметим, что во многих STL-алгоритмах не оговорено, что операции применяются в каком-то конкретном порядке даже в последовательном случае. Например, `std::for_each` действительно требует, чтобы доступ к элементам последовательности осуществлялся по порядку (для последовательной политики), но для `std::transform` это не так. Алгоритм `std::transform` посещает все элементы последовательности, но в каком порядке, не оговаривается. Если явно не указано противное, то последовательное выполнение означает, что вызовы функций доступа к элементам *неопределенно упорядочены* в вызывающем потоке. Говоря, что два вызова функции «неопределенно упорядочены», мы имеем в виду, что одна функция завершается до начала другой, но какая именно функция вызывается первой, не имеет значения. Таким образом, библиотека не вправе перемежать выполнение операций из разных функций, что, например, может исключить использование SIMD-операций.

Правило «так, будто» иногда приводит к неожиданным результатам в плане производительности. Например, политика `sequenced_policy` может показывать такие же результаты, как `unsequenced_policy`, потому что компилятор векторизует обе. Если полученные результаты вас смущают, то взгляните на отчет компилятора по оптимизации, откуда станет ясно, какие оптимизации были применены.

parallel_policy

Политика `parallel_policy` позволяет вызывать функции доступа к элементам из того же потока, что и сам алгоритм, или из других потоков, созданных библиотекой для распараллеливания. Однако все вызовы из одного и того же потока упорядочены друг относительно друга, т. е. выполнение функций доступа в одном потоке перемежать запрещено.

При использовании библиотеки Intel Parallel STL политика `parallel_policy` реализована с помощью обобщенных алгоритмов и задач TBB. Операции выполняются главным потоком и рабочими потоками TBB.

unsequenced_policy

Политика `unsequenced_policy` гарантирует, что все функции доступа к элементам вызываются из того же потока, что сам алгоритм. Но внутри вызывающего потока их выполнение может перемежаться. Это ослабление ограничения политики `sequenced` важно, потому что позволяет библиотеке агрегировать операции из разных вызовов функции в одну SIMD-команду или совмещать операции во времени.

SIMD-параллелизм можно реализовать с помощью векторных команд в ассемблерном коде, внутренних функций компилятора или прагм компилятора. В библиотеке Intel Parallel STL используются прагмы OpenMP SIMD.

Поскольку выполнение функций доступа к элементам может перемежаться в одном потоке, использовать в них мьютексы небезопасно (мьютексы будут описаны в главе 5). Представьте, к примеру, что произойдет, если несколько операций блокировки из разных функций перемежаются до того, как выполнена соответствующая операция разблокировки.

parallel_unsequenced_policy

Как нетрудно догадаться, политика `parallel_unsequenced_policy` ослабляет ограничения в двух отношениях: (1) функции доступа к элементам могут вызываться как из того же потока, что сам алгоритм, так из других потоков, созданных для распараллеливания, и (2) выполнение функций внутри одного потока может перемежаться.

Какую политику выполнения использовать?

При выборе политики выполнения прежде всего нужно убедиться, что она не ослабляет ограничения до такой степени, что значения, вычисленные алгоритмом, оказываются неверными.

Например, можно использовать `std::for_each` для вычисления выражения `a[i] = a[i] + a[i-1]` для вектора `a`, но этот код зависит от последовательного порядка выполнения `for_each` (который, в отличие от неопределенно упорядоченных алгоритмов, применяет заданную функцию к элементам по порядку):

```
float previous_value = 0.0;
std::for_each(
  pstl::execution::par, a.begin(), a.end(),
    [&](float &in)  {
      float p = previous_value;
      previous_value = in;
      in += p;
  }
);
```

В этом примере последнее значение хранится в переменной `previous_value`, захваченной по ссылке лямбда-выражением. Это работает, только если операции внутри одного потока управления выполняются по порядку. Любая политика, кроме `seq`, даст неправильные результаты.

Но предположим, что мы отнеслись к делу ответственно и знаем, какие политики допустимы для наших операций и какой STL-алгоритм использовать. Как сделать выбор между `sequenced_policy`, `unsequenced_policy`, `parallel_policy` и `parallel_unsequenced_policy`?

К сожалению, простого ответа не существует. Но есть рекомендации.

- Многопоточное выполнение следует использовать, только если у алгоритма достаточно работы, чтобы распараллеливание принесло выигрыш. Эвристические правила для принятия решения о том, использовать задачи или нет, обсуждаются в главе 16. Эти правила применимы и здесь. С параллельным выполнением сопряжены накладные расходы, и если работы недостаточно, то мы лишь увеличим издержки, не получив ничего взамен.
- Накладные расходы векторизации ниже, поэтому ее можно использовать в небольших внутренних циклах. Простые алгоритмы могут получить выигрыш от векторизации, даже если многопоточность его не дает.
- Но и у векторизации накладные расходы имеются. Для использования векторных регистров данные необходимо собрать вместе. Если данные

расположены в памяти не рядом или если к ним невозможен доступ с единичным шагом, то компилятору придется генерировать дополнительные команды для помещения данных в векторные регистры. В таких случаях векторизованный цикл может оказаться медленнее последовательного. Выведите отчеты компилятора по векторизации и изучите профили времени выполнения – нужно быть уверенным, что векторизация не сделает только хуже.

- Поскольку Parallel STL позволяет легко переключать политики, самый хороший образ действий – профилировать код и посмотреть, какая политика лучше работает на конкретной платформе.

Другие способы ввести SIMD-параллелизм

Помимо использования параллельных алгоритмов в STL, есть еще несколько способов включить в приложение SIMD-параллелизм. Самый простой и наиболее предпочтительный – при любой возможности использовать оптимизированные предметно-ориентированные или математические библиотеки. Например, Intel Math Kernel Library (Intel MKL) предлагает хорошо оптимизированные реализации многих математических функций, в т. ч. используемых в пакетах BLAS, LAPACK и FFTW. Эти функции могут применять многопоточность и векторизацию, если это выгодно, так что, выбрав их, мы получим то и другое в качестве бесплатного бонуса. А халява – это хорошо! Intel MLK поддерживает выполнение на основе TBB для многих функций, поэтому их композиция с другими видами основанного на TBB параллелизма обеспечивается автоматически.

Разумеется, может возникнуть необходимость в реализации алгоритмов, отсутствующих в готовых библиотеках. Тогда есть три подхода к добавлению векторных команд: (1) встроенный ассемблерный код, (2) внутренние функции компилятора, реализующие SIMD, и (3) векторизация при помощи компилятора.

Встроенный ассемблерный код позволяет включить конкретные команды, в т. ч. векторные, прямо в код приложения. Это низкоуровневый подход, зависящий от компилятора и процессора и, стало быть, наименее переносимый и сильнее других подверженный ошибкам. Но зато он дает полный контроль над командами (к добру или к худу). Мы прибегаем к этому подходу только в самом крайнем случае!

Лишь немногим лучше внутренние функции, реализующие SIMD. Большинство компиляторов предоставляет набор внутренних функций, позволяющих вставлять платформенно-зависимые команды, не прибегая к ассемблерному коду. Но хотя команды вставлять действительно проще, результат все равно зависит от компилятора и платформы, а сам подход чреват ошибками. Мы стремимся избегать его.

Последний вариант – положиться на векторизацию компилятором. На одном полюсе мы имеем полностью автоматизированную векторизацию – нужно только включить подходящие флаги компиляции, а компилятор сделает все остальное – хочется надеяться, к лучшему. Если заработает, то замечательно! Мы получаем все выгоды от векторизации задаром. А халява, как мы пом-

ним, – это хорошо. Но иногда компилятору необходимо подсказывать, чтобы он смог (или захотел) векторизовать наши циклы. Как именно подсказывать, зависит от компилятора, например в компиляторе Intel есть прагмы `#pragma ivdep` и `#pragma vector always`. Но есть и стандартизованные подходы, например прагмы `simd`, определенные в OpenMP. Как полностью автоматическая, так и управляемая пользователем векторизация на уровне компилятора гораздо лучше переносимы, чем вставка платформенно-зависимых команд непосредственно в код. На самом деле даже в библиотеке Intel Parallel STL используются прагмы OpenMP `simd` для переносимой поддержки векторизации в политиках `unseq` и `parallel_unseq`.

Резюме

В этой главе мы дали обзор библиотеки Parallel STL – какие алгоритмы и политики выполнения она поддерживает и как получить библиотеку, в которой движком является TBB. Затем обсудили классы нестандартных итераторов, предоставляемые TBB с целью расширить применимость STL-алгоритмов. Далее мы рассмотрели наиболее полезные и общие алгоритмы параллельного программирования: `std::for_each`, `std::transform`, `std::reduce` и `std::transform_reduce`. Мы показали, что некоторые примеры из главы 2 можно переписать с помощью этих алгоритмов. Но предупредили, что STL-алгоритмы не так выразительны, как алгоритмы из TBB, и что, пользуясь ими, мы теряем важные средства настройки производительности, описанные во второй части книги. Хотя библиотека Parallel STL полезна в некоторых простых случаях, из-за существующих на данный момент ограничений мы не рискуем рекомендовать ее для широкого применения в многопоточном программировании. При всем при том задачи TBB – не лучший путь к SIMD-параллелизму. Политики `unseq` и `parallel_unseq`, предоставляемые библиотекой Intel Parallel STL, включенной во все недавние дистрибутивы TBB, дополняют средства многопоточности TBB поддержкой векторизации.

Дополнительная информация

- *Vladimir Polin, and Mikhail Dvorskiy.* Parallel STL: Boosting Performance of C++ STL Code: C++ and the Evolution Toward Parallelism // The Parallel Universe Magazine, Intel Corporation. 2017. Issue 28. P. 5–18.
- *Alexey Moskalev, and Andrey Fedorov.* Getting Started with Parallel STL // https://software.intel.com/en-us/get-started-with-pstl. March 29, 2018.
- *Pablo Halpern, Arch D. Robison, Robert Geva, Clark Nelson, and Jen Maurer.* Vector and Wavefront Policies // Programming Language C++ (WG21), P0076r3. http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0076r3.pdf. July 7, 2016.
- Руководства по архитектурам Intel 64 и IA-32 для разработчиков программного обеспечения: https://software.intel.com/en-us/articles/intel-sdm.
- Руководство по внутренним функциям компилятора Intel: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

Глава 5

Синхронизация – почему ее нужно избегать и как это сделать

Подчеркнем с самого начала: если вам не нужны средства синхронизации, описанные в этой главе, тем лучше. Мы рассмотрим механизмы синхронизации и альтернативы, помогающие обеспечить взаимное исключение. Слова «синхронизация» и «исключение» должны вызывать крайне отрицательные эмоции у параллельных программистов, пекущихся о производительности. Мы стремимся избегать их, потому что они крадут драгоценное время, а в некоторых случаях также ресурсы процессора и энергию. Если мы можем переделать структуры данных и алгоритм, так чтобы необходимости в синхронизации и взаимном исключении не возникало, это замечательно! К сожалению, во многих случаях избежать операций синхронизации невозможно, и если вы столкнулись с этим, читайте дальше. Из этой главы следует вынести главный урок: тщательное обдумывание алгоритма обычно приводит к более чистой реализации, в которой нет чрезмерного использования синхронизации. Мы проиллюстрируем этот процесс, распараллелив простой код сначала в лоб, прибегая к мьютексам, затем воспользуемся атомарными переменными и напоследок уменьшим синхронизацию потоков, применив приватизацию и редукцию. И в завершение покажем, как поточно-локальная память (thread local storage – TLS) позволяет избежать накладных расходов на взаимное исключение. В этой главе предполагается, что вы хотя бы в общих чертах знакомы с понятиями «блокировки», «разделяемого изменяемого состояния», «взаимного исключения», «потокобезопасности», «гонки за данные» и другими относящимися к синхронизации материями. Если нет, то познакомьтесь с кратким введением в предмет в предисловии.

Сквозной пример: гистограмма изображения

Начнем с простого примера, который можно реализовать с помощью разных видов объектов взаимного исключения (мьютексов), атомарных переменных и даже вообще отказавшись от большинства операций синхронизации. Мы

опишем все возможные реализации, отметим их плюсы и минусы и воспользуемся ими для иллюстрации мьютексов, блокировок, атомарных переменных и поточно-локальной памяти.

Есть много видов гистограмм, но гистограммы изображений, наверное, используются чаще всего, особенно в фото- и видеокамерах и в программах обработки изображений. Например, почти во всех фоторедакторах имеется палитра, на которой показана гистограмма редактируемой фотографии (рис. 5.1).

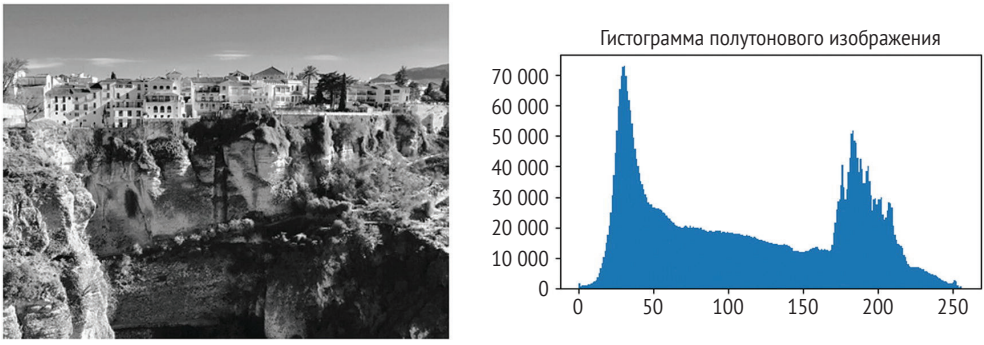


Рис. 5.1 ❖ Полутоновое изображение (город Ронда, провинция Малага) и его гистограмма

Для простоты будем рассматривать полутоновые изображения. В таком случае гистограмма представляет количество пикселей (по оси *y*) для каждого возможного значения яркости (по оси *x*). Если пиксели представлены байтами, то всего существует 256 значений яркости, или тонов, где нулю соответствует самый темный тон, а 255 – самый светлый. На рис. 5.1 видно, что в этом изображении чаще всего встречается темный тон: из 5 мегапикселей более 70 тысяч имеют тон 30 – на него приходится острый пик в окрестности точки $x = 30$. Фотографы и специалисты по обработке изображений видят в гистограммах способ быстро оценить распределение тонов и понять, скрыта ли какая-то информация в затемненных или высветленных участках изображения.

На рис. 5.2 вычисление гистограммы иллюстрируется для изображения 4×4, в котором тона пикселей принимают только восемь значений от 0 до 7. Двумерное изображение обычно представляется одномерным вектором, в котором 16 пикселей хранятся по строкам. Поскольку всего имеется восемь разных тонов, для гистограммы нужно только восемь элементов с индексами от 0 до 7. Элементы вектора гистограммы иногда называют «интервалами» (bin), которые мы «классифицируем», а затем подсчитываем количество пикселей каждого тона. На рис. 5.2 показана гистограмма `hist`, соответствующая данному изображению. Число «4», хранящееся в интервале с индексом 1, – результат подсчета четырех пикселей с тоном 1. Таким образом, основная операция обновления значений интервалов при обходе изображения – `hist[<tone>]++`.

С алгоритмической точки зрения, гистограмма представляется массивом целых чисел, в котором достаточно элементов для всех возможных тонов. В предположении, что изображение представлено массивом байтов, возмож-

ных тонов 256, поэтому в гистограмме должно быть 256 элементов, или интервалов. На рис. 5.3 приведен последовательный код вычисления гистограммы такого изображения.

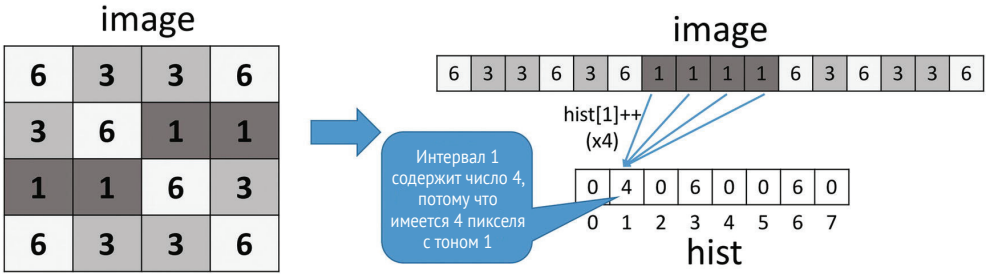


Рис. 5.2 ❖ Вычисление гистограммы hist изображения с 16 пикселями (каждое значение в гистограмме соответствует одному тону)

```
int main() {
  constexpr long int n = 1000000;
  constexpr int num_bins = 256;

  // Инициализировать генератор случайных чисел
  std::random_device seed;    // Случайное начальное значение
  std::mt19937 mte{seed()};   // mersenne_twister_engine
  std::uniform_int_distribution<> uniform{0,num_bins};
  // Инициализировать изображение
  std::vector<uint8_t> image; // пустой вектор
  image.reserve(n);           // заранее выделить память для вектора
  std::generate_n(std::back_inserter(image), n,
                  [&] { return uniform(mte); }
                );

  // Инициализировать гистограмму
  std::vector<int> hist(num_bins);

  // Последовательное выполнение
  tbb::tick_count t0 = tbb::tick_count::now();
  std::for_each(image.begin(), image.end(),
      [&hist] (uint8_t i) { hist[i]++; }
  );
  tbb::tick_count t1 = tbb::tick_count::now();
  double t_serial = (t1 - t0).seconds();

  std::cout << "Serial time: " << t_serial << std::endl;
  return 0;
}
```

Рис. 5.3 ❖ Последовательная реализация вычисления гистограммы. Наиболее интересные фрагменты заключены в рамку

Если вы понимаете весь код в этом листинге, то, пожалуй, можете пропустить данный раздел. Сначала объявляется вектор image размера n (скажем, один миллион для мегапиксельного изображения), а затем, после инициализации генератора случайных чисел, вектор заполняется случайными числами типа uint8_t в диапазоне [0, 255]. В качестве генератора используется вихрь Мерсенна (Mersenne twister engine), mte, который порождает случайные числа, равномерно распределенные в диапазоне [0, num_bins). Эти числа помещаются в вектор изображения. Затем строится вектор hist размера num_bins (по умолчанию все элементы инициализируются нулями). Отметим, что мы объявили пустой вектор изображения и впоследствии зарезервировали для него n целых чисел, вместо того чтобы конструировать объект image(n). Тем самым мы избежали обхода вектора для инициализации его нулями, которые впоследствии все равно были бы заменены случайными числами.

Собственно вычисление гистограммы можно было бы написать на C в более традиционной манере:

```
for (int i = 0; i < N; ++i) hist[image[i]]++;
```

Здесь мы в каждом элементе вектора гистограммы подсчитываем количество пикселей с данным тоном. Но в примере 5.3 мы продемонстрировали, как это сделать на C++ с применением STL-алгоритма for_each; этот способ, наверное, покажется более естественным программистам на C++. В подходе на основе for_each каждый элемент вектора image (тон типа uint8_t) передается лямбда-выражению, которое увеличивает на 1 значение в интервале, ассоциированном с тоном элемента. Для хронометража мы воспользовались классом tbb::tick_count, который измеряет время вычисления гистограммы. Функции-члены now и seconds вряд ли нуждаются в пояснении.

Небезопасная параллельная реализация

Первая наивная попытка распараллелить вычисление гистограммы, основанная на использовании алгоритма tbb::parallel_for, показана на рис. 5.4.

Чтобы можно было сравнить гистограммы, полученные в результате последовательной и параллельной реализаций, мы объявили новый вектор гистограммы hist_p. Далее возникает сумасшедшая идея обойти все пиксели параллельно. А почему бы и нет? Разве пиксели не являются независимыми? Для этой цели мы можем воспользоваться шаблоном parallel_for, описанным в главе 2, и поручить разным потокам чтение разных участков изображения. Однако эта идея не работает – сравнение векторов hist и hist_p (да, оператор hist!=hist_p в C++ делает именно то, что надо) в конце листинга на рис. 5.4 показывает, что векторы различны:

```
c++ -std=c++11 -O2 -o fig_5_4 fig_5_4.cpp -ltbb
./fig_5_4
Serial: 0.606273, Parallel: 6.71982, Speed-up: 0.0902216
Parallel computation failed!!
```

```
// Параллельное выполнение
std::vector<int> hist_p(num_bins);

t0 = tbb::tick_count::now();
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               for (size_t i = r.begin(); i < r.end(); ++i)
                 hist_p[image[i]]++;
             });
t1 = tbb::tick_count::now();
double t_parallel = (t1 - t0).seconds();

std::cout << "Serial: "   << t_serial   << ", ";
std::cout << "Parallel: " << t_parallel << ", ";
std::cout << "Speed-up: " << t_serial/t_parallel << std::endl;

if (hist != hist_p)
    std::cerr << "Parallel computation failed!!" << std::endl;
return 0;
```

Рис. 5.4 ❖ Небезопасная параллельная реализация вычисления гистограммы изображения

Проблема в том, что в параллельной реализации разные потоки, скорее всего, обновляют один и тот же разделяемый интервал одновременно. Иными словами, наш код не является потокобезопасным. Если говорить формально, наш параллельный небезопасный код демонстрирует «неопределенное поведение», или, иными словами, некорректен. На рис. 5.5 эта проблема иллюстрируется на примере двух потоков A и B, исполняемых ядрами 0 и 1, так что каждый поток обрабатывает половину пикселей. Поскольку в участке изображения, порученном потоку A, есть пиксель с тоном 1, то поток выполнит предложение `hist_p[1]++`. Поток B также читает пиксель такой же яркости и тоже выполняет предложение `hist_p[1]++`. Если оба инкремента происходят в одно и то же время, один на ядре 0, другой на ядре 1, то с большой вероятностью один инкремент будет пропущен.

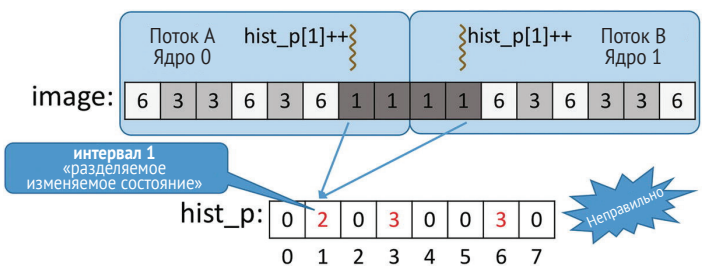


Рис. 5.5 ❖ Небезопасное параллельное обновление разделяемого вектора гистограммы

Это происходит, потому что операция инкремента не атомарна, а обычно состоит из трех ассемблерных команд: прочитать переменную из памяти в регистр, инкрементировать регистр и записать переменную назад в память[1]. Формально такого рода операция называется «чтение–изменение–запись». Возможность конкурентной записи в разделяемую переменную формально называется «разделяемым изменяемым состоянием». На рис. 5.6 показана возможная последовательность машинных команд, соответствующая предложению `hist_p[1]++` на C++.

`hist_p[1]++` →	**R1 в ядре 0**	**R1 в ядре 1**	**Время, или такт**
`load R1, @(hist_p+1)`	1	1	X
`add R1, R1 #1`	2	2	X+1
`store R1, @(hist_p+1)`	2	2	X+2

Рис. 5.6 ❖ Небезопасное обновление разделяемой переменной или разделяемого изменяемого состояния

Если к моменту выполнения этих двух операций инкремента уже был найден предыдущий пиксель яркости 1, то `hist_p[1]` содержит значение 1. Это значение будет прочитано в частные регистры обоими потоками, в результате чего в этот интервал будет записано число 2, а не 3, как положено. Пример, конечно, упрощенный, в нем не принимаются в расчет кеши и когерентность кешей, но феномен гонки за данные он иллюстрирует. Развернутый пример приведен в приложении (рис. P.15 и P.16).

Можно было бы понадеяться, что такая несчастливая последовательность событий на практике вряд ли случится, а если и случится, то результат параллельной версии изменится в приемлемых пределах. Конечно же, ради более быстрого выполнения стоит пожертвовать этой малостью, разве не так? Не совсем. Как показано выше, наша небезопасная параллельная реализация примерно в 10 раз медленнее последовательной (при работе в четыре потока на 4-ядерном процессоре и `n`, равном миллиону пикселей). Виной тому – протокол когерентности кешей, описанный в предисловии (см. раздел «Локальность и месть кешей»). При последовательном выполнении вектор гистограммы, скорее всего, будет целиком находиться в кеше L1 ядра, исполняющего код. Поскольку пикселей миллион, нужно будет выполнить миллион инкрементов в векторе, и большая их часть будет выполняться со скоростью работы кеша.

Примечание. В большинстве процессоров Intel строка кеша содержит 16 целых чисел (64 байта). Для вектора гистограммы с 256 целыми нужно всего 16 строк кеша, если вектор правильно выровнен. Поэтому после 16 непопаданий в кеш (или гораздо меньше, если включен режим предвыборки) все интервалы гистограммы будут кешированы, и для доступа к каждому в последовательной реализации понадобится всего три такта (это очень быстро!), конечно, если L1-кеш достаточно большой и строки, занятые гистограммой, не вытесняются другими данными.

[1] Главный принцип архитектуры фон Неймана состоит в том, что логика вычислений отделена от хранения данных, т. е. данные должны быть перемещены туда, где над ними можно производить вычисления, обработаны и снова перемещены в хранилище.

С другой стороны, в параллельной реализации все потоки стремятся кешировать интервалы в своих частных кешах, но когда один поток производит запись в интервал на одном ядре, протокол когерентности кешей делает недействительными все 16 интервалов в той же строке кеша на всех остальных ядрах. В результате последующие операции доступа к ставшим недействительными строкам кеша занимают на порядок больше времени, чем столь желанный доступ к L1-кешу. В результате такой взаимной инвалидации получается, что потоки в параллельной реализации инкрементируют некешированные интервалы, тогда как последовательная работает только с кешированными интервалами. Напомним еще раз, что для мегапиксельного изображения количество операций инкремента вектора гистограммы равно одному миллиону, поэтому мы хотим, чтобы инкремент выполнялся как можно быстрее. В параллельной реализации вычисления мы сталкиваемся как с ложным разделением (например, когда поток A увеличивает `hist_p[0]`, а поток B – `hist_p[15]`, поскольку оба интервала оказываются в одной строке кеша), так и с истинным разделением (когда оба потока A и B увеличивают `hist_p[i]`). Как бороться с тем и другим, мы рассмотрим в последующих разделах.

Первая безопасная параллельная реализация: крупнозернистая блокировка

Сначала решим проблему параллельного доступа к разделяемой структуре данных. Нам нужен механизм, не дающий другим потокам читать и изменять разделяемую переменную, когда какой-то поток уже находится в процессе изменения этой переменной. Проще говоря, нам нужна примерочная, в которую может войти один человек, примерить одежду, а затем предоставить помещение следующему в очереди. На рис. 5.7 показано, что запертая дверь примерочной не дает войти никому другому. В параллельном программировании дверь примерочной называется *мьютексом*. Когда человек входит в примерочную, он захватывает мьютекс и удерживает его, заперев дверь на замок, а когда выходит из примерочной – освобождает мьютекс, открыв дверь и оставив ее незапертой. Формально, мьютекс – это объект, обеспечивающий взаимное исключение при выполнении защищенного участка кода. Участок кода, нуждающийся в защите, обычно называют «критической секцией». Пример с примерочной иллюстрирует также идею состязания, т. е. состояния, в котором ресурс (примерочная) нужен сразу нескольким людям (рис. 5.7(c)). Поскольку в каждый момент времени в примерочной может находиться только один человек, ее использование «сериализовано». Так и любой ресурс, защищенный мьютексом, может снижать производительность программы, во-первых, из-за накладных расходов на управление мьютексом, а во-вторых, – и это важнее – из-за состязания за ресурс и сериализации операций доступа. Основная причина, по которой мы хотим свести синхронизацию к минимуму, – предотвращение состязания и сериализации, ограничивающих масштабируемость параллельных программ.

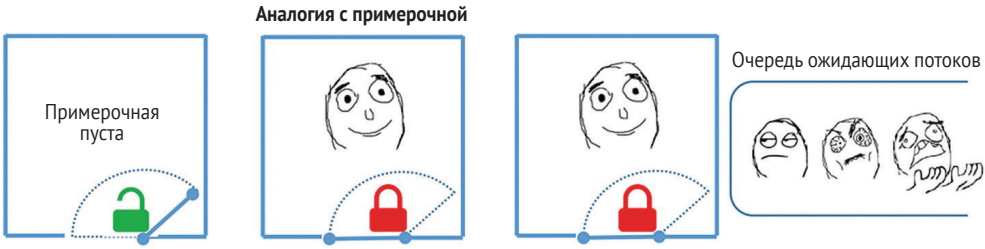


Рис. 5.7 ❖ Запирание двери в примерочную исключает других желающих

В этом разделе мы сосредоточимся на классах мьютексов в TBB и смежных с ними механизмах синхронизации. Хотя TBB появилась раньше C++11, стоит отметить, что C++11 все-таки стандартизовал класс мьютекса, хотя он и не так хорошо настраивается, как классы в библиотеке TBB. В TBB простейшим является класс `spin_mutex`, который становится доступен после включения заголовка `tbb/spin_mutex.h` или всеобъемлющего заголовка `tbb.h`. Располагая этим средством, мы можем реализовать безопасную параллельную версию вычисления гистограммы изображения, как показано на рис. 5.8.

```
#include <tbb/spin_mutex.h>

  // Параллельное выполнение
  using my_mutex_t = tbb::spin_mutex;
  my_mutex_t my_mutex;

  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
               [&](const tbb::blocked_range<size_t>& r)
               {
                 my_mutex_t::scoped_lock my_lock{my_mutex};
                 for (size_t i = r.begin(); i < r.end(); ++i)
                   hist_p[image[i]]++;
               });
```

Рис. 5.8 ❖ Первая безопасная параллельная реализация вычисления гистограммы изображения, в которой используется крупнозернистая блокировка

Объект `my_lock` захватывает мьютекс `my_mutex` в момент создания и автоматически освобождает его в деструкторе, вызываемом при выходе из области видимости. Поэтому рекомендуется заключать защищенные участки кода в дополнительные фигурные скобки, чтобы максимально сократить время блокировки и поскорее открыть доступ к ресурсу другим ожидающим потокам.

Примечание. Если бы в коде на рис. 5.8 мы забыли указать имя объекта блокировки, например:

```
// было my_lock{my_mutex}
my_mutex_t::scoped_lock {my_mutex};
```

то код откомпилировался бы без предупреждений, но область видимости `scoped_lock` простиралась бы только до точки с запятой. Без указания имени объекта (`my_lock`) мы конструируем анонимный объект класса `scoped_lock`, время жизни которого ограничено точкой с запятой, потому что нет никакого именованного объекта, который мог бы существовать за пределами определения. Это бесполезно и *не* защищает критическую секцию с помощью взаимного исключения.

На рис. 5.9 представлена более явная, но **нерекомендуемая** альтернатива коду рис. 5.8.

```
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               my_mutex_t::scoped_lock my_lock;
               my_lock.acquire(my_mutex);
               for (size_t i = r.begin(); i < r.end(); ++i)
                 hist_p[image[i]]++;
               my_lock.release();
             });
```

Рис. 5.9 ❖ Нерекомендуемая альтернатива захвату мьютекса

Адепты C++ предпочитают вариант рис. 5.8, называемый идиомой RAII (захват ресурса есть инициализация), поскольку он освобождает нас от необходимости помнить об освобождении блокировки. Но еще важнее тот факт, что в RAII-версии деструктор объекта блокировки, в котором мьютекс освобождается, вызывается даже в случае исключения, поэтому мьютекс не останется захваченным в результате исключения. Если бы в варианте на рис. 5.9 произошло исключение перед вызовом функции-члена `my_lock.release()`, то мьютекс все равно был бы освобожден, потому что вызывается деструктор. Если объект блокировки покидает область видимости, но перед этим был освобожден функцией-членом `release()`, то деструктор не делает ничего.

Но вернемся к коду на рис. 5.8. Вероятно, у вас возник вопрос: «Минуточку, но разве мы не сериализовали параллельный код, добавив крупнозернистую блокировку?» Да, вы правы! Как показано на рис. 5.10, каждый поток, желающий обработать свою часть изображения, сначала пытается захватить мьютекс, но удастся это сделать только одному, а остальные будут нетерпеливо ждать освобождения мьютекса. И лишь когда поток, удерживающий блокировку, снимет ее, другой поток сможет приступить к выполнению защищенного кода. В итоге `parallel_for` на деле выполняется последовательно! Но есть и хорошая новость – теперь конкурентных операций инкремента интервалов гистограммы нет, и код наконец-то выполняется правильно. Ура!

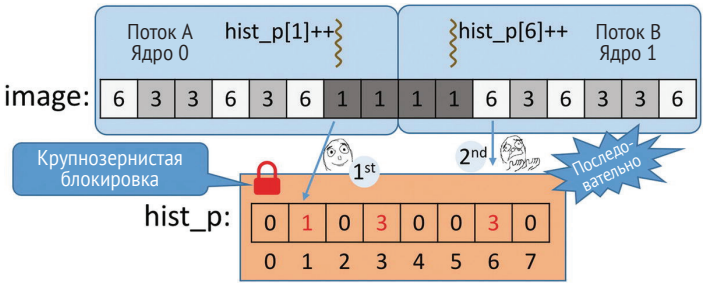


Рис. 5.10 ❖ Поток A удерживает крупнозернистую блокировку, пока обновляет интервал 1, а поток B ждет, потому что весь вектор гистограммы заблокирован

Если теперь откомпилировать и выполнить новую версию, то окажется, что параллельное выполнение стало чуть медленнее последовательного:

```
c++ -std=c++11 -O2 -o fig_5_8 fig_5_8.cpp -ltbb
./fig_5_8
Serial: 0.61068, Parallel: 0.611667, Speed-up: 0.99838
```

Этот подход называется крупнозернистой блокировкой, потому что мы защищаем крупнозернистую структуру данных (в данном случае всю структуру – вектор гистограммы – целиком). Можно было бы разбить вектор на несколько частей и защищать каждую часть своей блокировкой. Так мы смогли бы увеличить степень конкурентности (потоки, обращающиеся к разным частям, могут работать параллельно), но заодно увеличили бы сложность кода и объем памяти, занятой мьютексами.

Здесь уместно предостережение! На рис. 5.11 показана типичная ошибка, допускаемая начинающими параллельными программистами.

```
//my_mutex_t my_mutex     **теперь объявлен в теле**
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               my_mutex_t my_mutex;
               my_mutex_t::scoped_lock my_lock{my_mutex};
               for (size_t i = r.begin(); i < r.end(); ++i)
                 hist_p[image[i]]++;
             });
```

Рис. 5.11 ❖ Типичная ошибка, допускаемая начинающими параллельными программистами

Компилятор не выдает ни ошибок, ни предупреждений, так что же с этим кодом не так? Вернемся к аналогии с примерочной – мы намеревались воспрепятствовать одновременному нахождению нескольких человек в помещении. В показанном выше коде объект `my_mutex` определен внутри выполняемого параллельно участка, так что существует по одному мьютексу на каждую задачу,

и каждая задача захватывает свой мьютекс. Конечно, это не предотвращает одновременный доступ к критической секции. Как видно по рис. 5.12, этот код по существу означает, что у каждого человека имеется своя дверь в примерочную! Это совсем не то, чего мы хотели. Решение заключается в том, чтобы объявить `my_mutex` один раз (как на рис. 5.8), чтобы все должны были входить в примерочную через одну дверь.

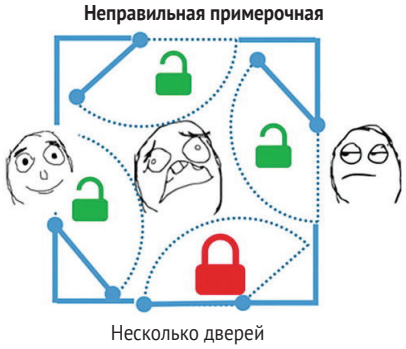


Рис. 5.12 ❖ Примерочная с несколькими дверьми

Прежде чем переходить к мелкозернистой блокировке, обсудим два момента. Во-первых, время выполнения «распараллеленного, а затем сериализованного» кода на рис. 5.8 больше времени работы последовательной реализации. Это связано как с накладными расходами на такое «сериализованное распараллеливание», так и с неудачным использованием кешей. Конечно, ни ложного, ни истинного разделения здесь нет, потому что и «разделять» нечего! Или все-таки есть что? В последовательной реализации только один поток обращается к кешированному вектору гистограммы. В крупнозернистой реализации, когда один поток обрабатывает свой участок изображения, он кеширует гистограмму в кеше того ядра, на котором работает. Когда следующий поток в очереди наконец сможет обработать свой участок, ему, возможно, потребуется кешировать гистограмму в другом кеше (если этот поток работает на другом ядре). Потоки по-прежнему разделяют вектор гистограммы, и количество непопаданий в кеш, вероятно, будет больше, чем при последовательном выполнении.

Второй момент – возможность настройки поведения мьютекса путем выбора одного из нескольких вариантов мьютексов, показанных на рис. 5.13. Поэтому рекомендуется включить предложение

```
using my_mutex_t = <вариант_мьютекса>
```

или его эквивалент на C

```
typedef <mutex_flavor> my_mutex_t;
```

и далее использовать тип `my_mutex_t`. Таким образом, мы легко можем изменить вариант мьютекса в программе и экспериментально определить, какой вариант лучше. Возможно, придется также включить другой заголовочный файл, как показано на рис. 5.13, или использовать всеобъемлющий файл `tbb.h`.

Вариант мьютекса	Масштабируемый	Справедливый	Рекурсивный	При долгом ожидании	Размер
`mutex` (`mutex.h`)	зависит от ОС	зависит от ОС	нет	блокирует	⩾ 3 слов
`recursive_mutex` (`recursive_mutex.h`)	зависит от ОС	зависит от ОС	да	блокирует	⩾ 3 слов
`spin_mutex` (`spin_mutex.h`)	нет	нет	нет	уступает	1 байт
`speculative_spin_mutex` (`spin_mutex.h`)	зависит от оборудования	нет	нет	уступает	2 строки кеша
`queuing_mutex` (`queuing_mutex.h`)	да	да	нет	уступает	1 слово
`spin_rw_mutex` (`spin_rw_mutex.h`)	нет	нет	нет	уступает	1 слово
`speculative_spin_rw_mutex` (`spin_rw_mutex.h`)	зависит от оборудования	нет	нет	уступает	3 строки кеша
`queuing_rw_mutex` (`queuing__rw_mutex.h`)	да	да	нет	уступает	1 слово
`null_mutex` (`queuing_mutex.h`)	теоретически	да	да	никогда	пустой
`null_rw_mutex` (`queuing_mutex.h`)	теоретически	да	да	никогда	пустой

Рис. 5.13 ❖ Различные варианты мьютексов и их свойства

Варианты мьютексов

Чтобы разобраться в различных вариантах мьютексов, нужно сначала описать свойства, по которым они классифицируются.

- **Масштабируемые** мьютексы не потребляют ни дополнительных тактов процессора, ни полосы пропускания к памяти, ожидая своей очереди. Идея в том, что ожидающий поток не должен потреблять аппаратных ресурсов, нужных работающим потокам.
- **Справедливые** мьютексы упорядочивают ожидающие их потоки по очередности в порядке FIFO.
- **Рекурсивные** мьютексы разрешают потоку, удерживающему мьютекс, захватить тот же самый мьютекс еще раз. Переосмыслить свой код, чтобы избежать мьютексов, – дело хорошее, но избегать рекурсивных мьютексов – почти необходимость! Тогда зачем же TBB их предоставляет? Потому что бывают редкие ситуации, когда без рекурсивных мьютексов не обойтись. Они могут пригодиться и тогда, когда нет времени или желания придумывать более эффективное решение.

В таблице на рис. 5.13 мы включили также размер объекта мьютекса и поведение потока в случае, когда он вынужден долго ждать возможности захватить мьютекс. Что касается последнего пункта, то у потока в состоянии ожидания есть три варианта действий: активное ожидание, блокировка или уступка. Блокирующийся поток переходит в заблокированное состояние, в котором единственный потребляемый ресурс – память, занятая в спящем состоянии. Когда поток наконец захватывает мьютекс, он просыпается и возвращается в состояние готовности, в котором все потоки ждут доступа к процессору. Планировщик ОС назначает временные кванты готовым потокам, стоящим в очереди

к процессору. Уступающий поток, который ждет получения мьютекса, остается в состоянии готовности. Дойдя до начала очереди, он получает возможность поработать, но если мьютекс все еще заблокирован другим потоком, то снова отказывается от своего кванта (больше ему ничего не остается!) и возвращается в очередь готовых.

Примечание. Отметим, что в этом процессе присутствуют две очереди: (i) очередь готовых к выполнению потоков, которую обслуживает планировщик ОС необязательно в порядке FIFO, – стоящие в ней потоки получают в свое распоряжение простаивающее ядро и становятся исполняемыми; (ii) очередь к мьютексу, обслуживаемая ОС или библиотекой мьютексов в пространстве пользователя, – находящиеся в ней потоки ждут возможности захватить мьютекс, к которому стоит очередь.

Если ядро не перегружено (очередь к нему пуста), а поток уступает процессор, потому что нужный ему мьютекс все еще занят, то он окажется единственным в очереди готовых и сразу же будет диспетчеризован, т. е. получит доступ к процессорному ядру. Таким образом, в этом случае механизм уступки эквивалентен активному ожиданию.

Теперь, когда мы понимаем свойства, характеризующие реализацию мьютекса, можно перейти к конкретным вариантам мьютексов в библиотеке TBB.

`mutex` и **`recursive_mutex`** – это предоставляемые TBB обертки вокруг реализованного в ОС механизма мьютексов. Мы используем обертки вместо «родного» мьютекса, потому что TBB добавляет безопасность относительно исключений и такие же интерфейсы, как у других мьютексов. Эти мьютексы блокируют поток при долгом ожидании, поэтому потребляют меньше процессорных тактов, но занимают место в памяти и характеризуются более длительным временем отклика, когда мьютекс становится доступным.

Напротив, **`spin_mutex`** никогда не блокирует поток. Он активно ожидает в пространстве пользователя, удерживая блокировку. Ожидающий поток уступит процессор после нескольких попыток захватить мьютекс, но если на ядро нет других претендентов, то этот поток так и будет потреблять такты и энергию. С другой стороны, после освобождения мьютекса время, необходимое для его захвата, наименьшее из возможных (нет нужды будить поток и ждать очереди на выполнение). Этот мьютекс несправедлив, т. е. вне зависимости от того, сколько времени поток ждал своей очереди, более расторопный поток может вклиниться и захватить мьютекс, если первым обнаружит, что тот свободен. Это подход типа «налетай – подешевело», и в крайних случаях слабейший поток будет прозябать, так и не получая доступа к мьютексу. Тем не менее именно этот вариант мьютекса рекомендуется в ситуациях ненапряженного состязания, поскольку может оказаться самым быстрым.

`queueing_mutex` – масштабируемая и справедливая версия `spin_mutex`. Он также активно ожидает в пространстве пользователя, но потоки, ожидающие мьютекса, захватывают его в порядке очередности FIFO, так что бесконечное прозябание невозможно.

`speculative_spin_mutex` построен поверх аппаратной транзакционной памяти (Hardware Transactional Memory – HTM), имеющейся в некоторых процессорах. Философия HTM – будь оптимистом! HTM разрешает всем потокам одновре-

менный вход в критическую секцию в надежде, что конфликтов в борьбе за разделяемую память не будет! А если все-таки будут? Тогда оборудование обнаружит конфликт и откатит выполнение одного из конфликтующих потоков, которому придется повторить выполнение кода в критической секции. В крупнозернистой реализации на рис. 5.8 мы можем добавить такую строку:

```
using my_mutex_t = speculative_spin_mutex;
```

и тогда алгоритм `parallel_for`, обходящий изображение, снова станет параллельным. Теперь всем потокам разрешено входить в критическую секцию (чтобы обновить интервалы гистограммы для своего участка), и лишь если действительно возникнет конфликт при обновлении интервалов, то выполнение придется повторить. Чтобы этот подход был эффективен, защищенная критическая секция должна быть настолько мала, чтобы конфликты и повторы случались редко. В коде на рис. 5.8 дело обстоит не так.

`spin_rw_mutex`, **`queueing_rw_mutex`** и **`speculative_spin_rw_mutex`** – варианты уже рассмотренных мьютексов типа читатель–писатель. Они позволяют нескольким читателям одновременно обращаться к одной и той же разделяемой переменной. У конструктора объекта `lock` имеется второй булев аргумент, равный `false`, если мы собираемся только читать (но не писать) внутри критической секции:

```
using rwmutex_t = spin_rw_mutex;
rwmutex_t my_mutex;
{
  rwmutex_t::scoped_lock my_lock{my_mutex,/*is_writer=*/false};
// Захвачена читательская блокировка, поэтому разрешено
// несколько конкурентных операций чтения

}
```

На случай, когда по какой-то причине читательская блокировка должна быть преобразована в писательскую, TBB предлагает функцию-член `upgrade_to_writer()`, используемую следующим образом:

```
bool success = my_lock.upgrade_to_writer()
```

которая возвращает `true`, если `my_lock` успешно преобразован без освобождения мьютекса, и `false` в противном случае.

Наконец, **`null_mutex`** и **`null_rw_mutex`** – просто фиктивные объекты, не делающие ничего. Зачем же они нужны? Они могут быть полезны для передачи объекта мьютекса шаблонной функции, которой иногда необходим настоящий мьютекс, а иногда – нет. Если функции не нужен мьютекс, достаточно передать фиктивную заглушку.

Вторая безопасная параллельная реализация: мелкозернистая блокировка

Итак, мы уже много знаем о разных вариантах мьютексов, и пора подумать об альтернативе крупнозернистой блокировки на рис. 5.8. Одна такая альтернатива – объявить по одному мьютексу для каждого интервала гистограммы,

тогда вместо блокировки всей структуры данных мы будем защищать только тот элемент, который увеличиваем. Для этого понадобится вектор мьютексов `fine_m`, показанный на рис. 5.14.

```
using my_mutex_t=tbb::spin_mutex;
std::vector<my_mutex_t> fine_m(num_bins);
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               for (size_t i = r.begin(); i < r.end(); ++i){
                 int tone=image[i];
                 my_mutex_t::scoped_lock my_lock{fine_m[tone]};
                 hist_p[tone]++;
               }
             });
```

Рис. 5.14 ❖ Вторая безопасная параллельная реализация вычисления гистограммы изображения с применением мелкозернистой блокировки

Как видно из лямбда-выражения внутри `parallel_for`, когда потоку нужно увеличить значение в интервале `hist_p[tone]`, он захватывает мьютекс `fine_m[tone]`, не давая другим потокам обращаться к тому же интервалу. То есть говорит: «можете обновлять другие интервалы, но только не этот». Это иллюстрируется на рис. 5.15, где потоки A и B параллельно обновляют значения в разных интервалах вектора гистограммы.

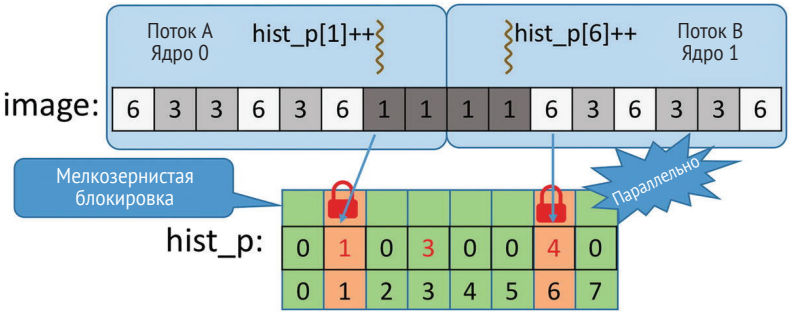


Рис. 5.15 ❖ Благодаря мелкозернистой блокировке удается лучше задействовать параллелизм

Однако, с точки зрения производительности, эта альтернатива далеко не оптимальна (на самом деле она худшая из всех рассмотренных до сих пор):

```
c++ -std=c++11 -O2 -o fig_5_14 fig_5_14.cpp -ltbb
./fig_5_14
Serial: 0.59297, Parallel: 26.9251, Speed-up: 0.0220229
```

Теперь нам необходим не только массив гистограммы, но и массив мьютексов той же длины. Это увеличивает требования к памяти, и, кроме того, нужно

будет кешировать больше данных, страдая как от истинного, так и от ложного разделения. Паршиво!

Помимо присущих любой блокировке накладных расходов, блокировки создают еще две проблемы: караван и взаимоблокировка. Сначала рассмотрим «караван» (convoying). Название навеяно метафорой, согласно которой все потоки следуют друг за другом в строю, скорость которого определяется низкой скоростью переднего. На рис. 5.16 показан пример, иллюстрирующий эту ситуацию. Предположим, что потоки 1, 2, 3, 4 исполняют один и тот же код на одном и том же ядре, и в этом коде имеется критическая секция, защищенная спин-мьютексом A. Если потоки удерживают мьютекс в разное время, то они спокойно работают без состязания (ситуация 1). Но может случиться, что квант потока 1 закончится до того, как он успеет освободить свой мьютекс, и тогда он окажется в конце очереди готовых (ситуация 2).

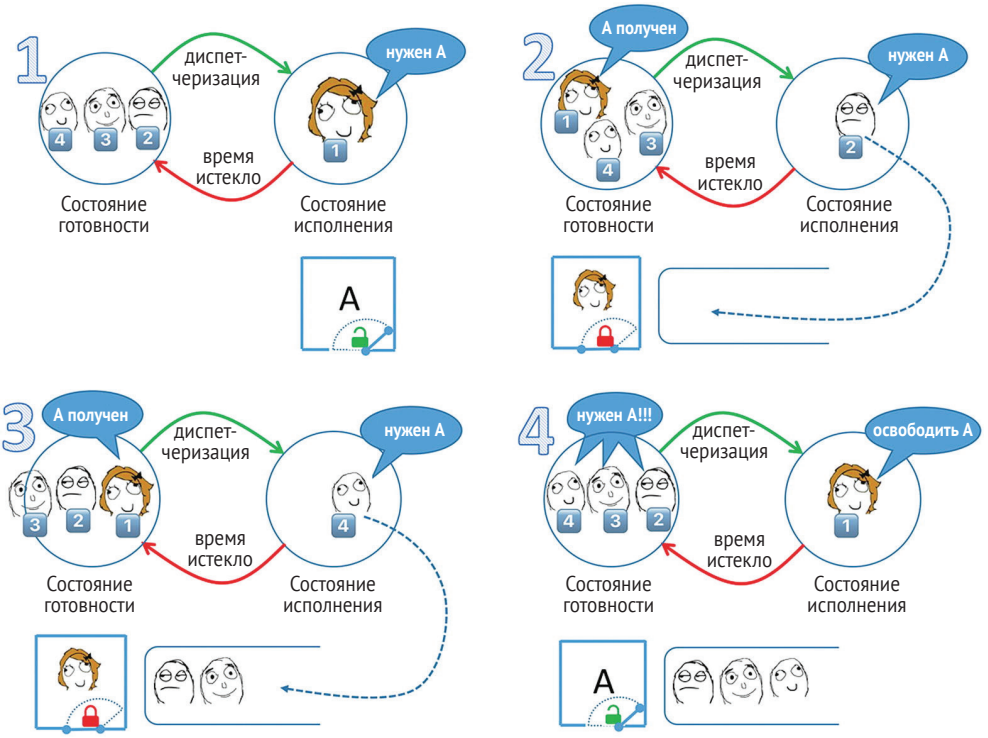
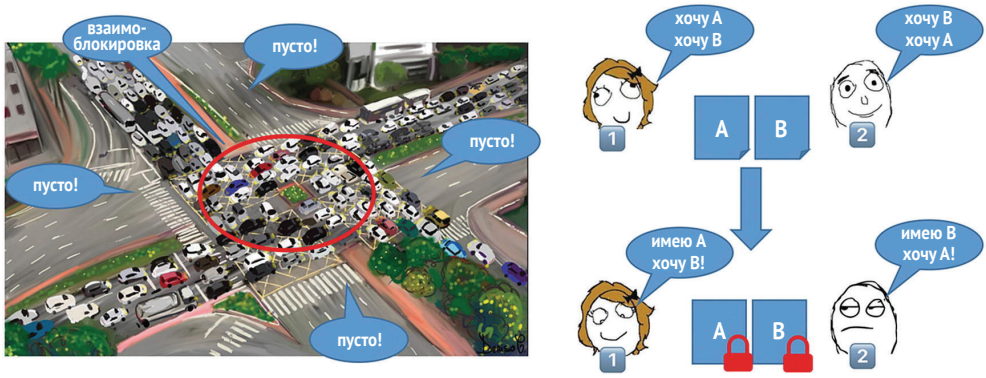


Рис. 5.16 ❖ Караван в случае перегруженного ядра
(одно ядро исполняет четыре потока, и все они нуждаются в одном и том же мьютексе A)

Потоки 2, 3, 4 получат свои кванты времени, но не смогут захватить мьютекс, поскольку поток 1 все еще владеет им (ситуация 3). Это означает, что потоки 2, 3 и 4 могут либо уступить процессор, либо активно ждать, но в любом случае они вынуждены тащиться за грузовиком, едущим на первой передаче. Когда поток 1 снова получит квант времени, он освободит мьютекс A (ситуация 4). Теперь потоки 2, 3 и 4 изготовились к битве за мьютекс, в которой победит

только один, а остальные снова будут ждать. Эта ситуация повторяется, особенно если потокам 2, 3 и 4 нужно больше одного кванта времени для работы в своих критических секциях. Теперь потоки 2, 3 и 4 оказались непреднамеренно скоординированными, все они синхронно проходят один и тот же участок кода, что только повышает вероятность состязания за мьютекс! Заметим, что караван особенно неприятен, когда ядра перегружены (как в данном примере, где на одном ядре исполняется четыре потока), и это лишний аргумент в пользу нашей рекомендации избегать перегруженности.

С блокировками связана также хорошо известная проблема взаимоблокировки. На рис. 5.17(a) показана кошмарная ситуация, в которой никто не может двинуться дальше, хотя свободные ресурсы имеются (пустые полосы, по которым никто не может ехать). Это реальный затор, но выкиньте эту метафору из головы (если сможете!) и вернитесь в виртуальный мир параллельного программирования. Если имеется N потоков, которые удерживают блокировку и в то же время хотят захватить блокировку, которую уже удерживает какой-то другой поток из того же множества, то все эти N потоков блокируют друг друга. Пример показан на рис. 5.17(b) для случая двух потоков: поток 1 удерживает мьютекс A и ждет возможности захватить мьютекс B, а поток 2 удерживает мьютекс B и ждет возможности захватить мьютекс A. Очевидно, что ни один поток не может продвинуться, и оба они навечно застыли в смертельном объятии! Такой печальной участи можно избежать, если поток не будет пытаться захватить другой мьютекс, когда один уже захвачен. Или, по крайней мере, все потоки всегда будут захватывать мьютексы в одном и том же порядке.



(a) Взаимоблокировка в реальной жизни

(b) Взаимоблокировка в виртуальной жизни

Рис. 5.17 ❖ Взаимоблокировки

Взаимоблокировку можно спровоцировать ненамеренно, если поток, удерживающий блокировку, вызывает функцию, которая захватывает другую блокировку. Рекомендуется воздерживаться от вызова функций, когда удерживается блокировка, если неизвестно, что функция делает (обычно этот совет формулируется иначе – не вызывать чужих функций, удерживая блокировку). Альтернатива – тщательно проверять, что цепочка вызовов функций не может привести к взаимоблокировке. Да и вот еще – избегать блокировок всюду, где это возможно!

Хотя ни караван, ни взаимоблокировка не возникают в нашей реализации гистограммы, они должны убедить вас, что блокировки зачастую приносят больше проблем, чем решают, и что это вовсе не лучший способ добиться высокой производительности параллельной программы. Лишь когда вероятность состязания мала и время нахождения в критической секции минимально, можно смириться с блокировками. В таких случаях простой `spin_lock` или `speculative_spin_lock` может дать некоторое ускорение. Но в остальных случаях масштабируемость алгоритма на основе блокировок сильно страдает, и лучший совет – подойти творчески и придумать новую реализацию, вообще не требующую мьютексов. Но возможна ли мелкозернистая синхронизация, не опирающаяся на несколько мьютексов и позволяющая избежать сопряженных с ними накладных расходов и потенциальных проблем?

Третья потокобезопасная параллельная реализация: атомарные переменные

По счастью, имеется не такой дорогостоящий алгоритм, к которому во многих случаях можно прибегнуть, чтобы избавиться от мьютексов и блокировок. Для выполнения атомарных операций можно воспользоваться атомарными переменными. На рис. 5.6 было показано, что операция инкремента не атомарна, а делится на три подоперации (чтение, инкремент и сохранение). Но если объявить атомарную переменную и поступить так:

```
#include <tbb/atomic.h>
tbb:atomic<int> counter;
counter++;
```

то инкремент *станет* атомарной операцией. Это означает, что любому другому потоку, обращающемуся к значению счетчика, ситуация будет представляться так, будто инкремент выполняется за один шаг (не с помощью трех меньших операций, а с помощью одной). То есть любой другой поток, сколь угодно «остроглазый», будет считать, что операция либо не началась, либо уже завершилась, но никогда не увидит ее завершенной наполовину.

Атомарные операции не подвержены ни каравану, ни взаимоблокировке[1] и выполняются быстрее взаимного исключения. Но не все операции можно выполнить атомарно, а те, которые можно, применимы не ко всем данным. Точнее, тип `atomic<T>` поддерживает атомарные операции, когда тип `T` – целочисленный, перечисление или указательный. Атомарные операции, поддерживаемые для переменной `x` такого типа `atomic<T>`, перечислены на рис. 5.18.

Имея эти пять операций, можно реализовать много производных. Например, `x++`, `x--`, `x+=...` и `x-=...` – операции, производные от `x.fetch_and_add()`.

---

[1] Атомарные операции не могут быть вложенными, поэтому взаимоблокировке неоткуда взяться.

`=x`	прочитать значение `x`
`x=`	записать значение `x` и вернуть его
`x.fetch_and_store(y)`	выполнить `x = y` и вернуть старое значение `x`
`x.fetch_and_add(y)`	выполнить `x += y` и вернуть старое значение `x`
`x.compare_and_swap(y)`	если `x` равно `z`, то выполнить `x = y`. В любом случае вернуть старое значение `x`

Рис. 5.18 ❖ Фундаментальные операции над атомарными переменными

Примечание. В предыдущих главах мы уже упоминали, что начиная с версии стандарта C++11 в язык C++ включены средства синхронизации и многопоточности. В библиотеке TBB они существовали до включения в стандарт. Хотя среди прочих доступны и типы `std::mutex` и `std::atomic`, TBB по-прежнему предоставляет частично перекрывающуюся функциональность с помощью классов `tbb::mutex` и `tbb::atomic`, в основном для совместимости с ранее написанными приложениями. В одной программе можно без опаски использовать оба варианта, и только вам решать, какому отдать предпочтение в конкретной ситуации. Если говорить о `std::atomic`, то из нее можно выжать немного дополнительной производительности по сравнению с `tbb::atomic`, если использовать для разработки безблокировочных алгоритмов и структур данных в «слабо упорядоченных архитектурах» (например, ARM или PowerPC, но не при работе с процессорами Intel, в которых модель памяти строго упорядоченная). В последнем разделе этой главы мы приводим ссылки на дополнительные материалы по согласованности памяти и модели конкурентности в C++, где эта тема рассмотрена всесторонне. Нам же достаточно знать, что функции `fetch_and_store`, `fetch_and_add` и `compare_and_swap` по умолчанию обладают последовательной согласованностью (`memory_order_seq_cst` в терминологии C++), которая предотвращает некоторые виды выполнения не по порядку и потому занимает чуть больше времени. Чтобы учесть этот момент, TBB предлагает также семантику захвата и освобождения: захват по умолчанию в атомарном чтении (`...=x`) и освобождение по умолчанию в атомарной записи (`x=...`). Желательную семантику можно определить также с помощью аргумента шаблона, например `x.fetch_and_add<release>` гарантирует только порядок доступа к памяти при освобождении. В C++11 разрешены и другие, более слабые, порядки доступа к памяти (`memory_order_relaxed` и `memory_order_consume`), которые в конкретных случаях и архитектурах могут дать несколько большую свободу в выборе порядка чтения и записи и чуть повысить производительность. Если ради наивысшей производительности мы готовы работать на уровне, более близком к «железу», понимая, что это обернется дополнительными усилиями по кодированию и отладке, то в нашем распоряжении низкоуровневые средства C++11, которые, впрочем, можно сочетать с высокоуровневыми абстракциями TBB.

Еще одна полезная идиома, основанная на атомарных переменных, уже использовалась в примере с прямой подстановкой на рис. 2.23. Если имеется атомарная переменная `refCount`, инициализированная значением `y`, и несколько потоков выполняют такой код:

```
if (--refCount==0) {… /* тело */… };
```

то только `y`-й поток при выполнении этой строки войдет в «body».

Из пяти фундаментальных операций `compare_and_swap` (CAS) можно считать матерью всех атомарных операций чтения–изменения–записи. Дело в том, что все остальные операции могут быть реализованы в терминах CAS.

Примечание. На случай, если вы захотите защитить небольшую критическую секцию и прониклись убежденностью во вредности блокировок, расскажем чуть больше о деталях операции CAS. Предположим, что требуется атомарно умножить разделяемую целую переменную v на 3 (не спрашивайте, зачем, – есть причины!). Мы стремимся найти безблокировочное решение, хотя знаем, что умножение не входит в число атомарных операций. И тут на выручку приходит CAS. Первым делом объявим v как атомарную переменную:

```
tbb::atomic<uint_32_t> v;
```

так что теперь мы можем вызывать `v.compare_and_swap(new_v, old_v)`, которая атомарно выполняет следующую последовательность операций:

```
ov=v; if (ov==old_v) v=new_v; return ov;
```

Это означает, что тогда и только тогда, когда v равно `old_v`, мы можем записать в v новое значение. В любом случае возвращается ov (разделяемая переменная v, упоминаемая в сравнении ==). Теперь реализация нашего атомарного «умножения на 3» сводится к так называемому CAS-циклу:

```
void fetch_and_triple(tbb::atomic<uint32_t>& v)
{
   uint32_t old_v;
   do {
     old_v=v; // сделать мгновенный снимок
   } while (v.compare_and_swap(old_v * 3, old_v)!=old_v);
}
```

Новая функция `fetch_and_triple` потокобезопасна (ее могут безопасно вызывать несколько потоков одновременно), даже когда при вызове ей передают одну и ту же разделяемую атомарную переменную. По существу, это цикл do-while, в котором сначала делается снимок разделяемой переменной (это ключ к последующему сравнению на случай, если другому потоку удалось изменить ее). Затем – атомарно – **если никакой другой поток не изменил v** (`v==old_v`), то изменение выполняем мы (`v=old_v*3`) и возвращаем v. Поскольку в этом случае `v == old_v` (повторим: никакой другой поток не изменил v), мы покидаем цикл do-while и возвращаем из функции успешно обновленную переменную v.

Но возможно, что после снимка какой-то другой поток обновил v. Тогда `v!=old_v`, откуда следует, что (i) мы не обновляем v и (ii) мы остаемся в цикле do-while в надежде, что госпожа удача улыбнется нам в следующий раз (когда никакой другой алчный поток не осмелится тронуть нашу v в промежутке между снимком и ее последующим обновлением). На рис. 5.19 показано, что v всегда обновляется – либо потоком 1, либо потоком 2. Не исключено, что одному из потоков придется повторить попытку (как потоку 2, который записывает 81, хотя изначально собирался записать 27) один или несколько раз, но при обычных обстоятельствах это не должно стать проблемой.

В этой стратегии есть два недостатка: (i) она плохо масштабируется и (ii) она может быть подвержена «проблеме ABA» (в главе 6 приведены сведения о классической проблеме ABA). Что касается первого, то рассмотрим P потоков, состязающихся за одну и ту же атомарную переменную, и предположим, что только один выигрывает после P–1 попыток, затем второй после P–2 попыток, потом третий после P–3 попыток и т. д. В результате объем работы растет квадратично. Эту проблему можно смягчить, прибегнув к стратегии «экспоненциальной задержки», т. е. кратно уменьшать частоту последовательных попыток снизить конкуренцию. Проблема же ABA возникает, когда в промежутке между снимком и последующим обновлением v какой-то другой поток изменяет значение v с A

на B и обратно на A. Наш CAS-цикл радостно продолжит работу, не заметив действий вмешавшегося потока, что может привести к проблеме. Решившись прибегнуть к CAS-циклу в своем приложении, убедитесь, что вы понимаете эту проблему и ее возможные последствия.

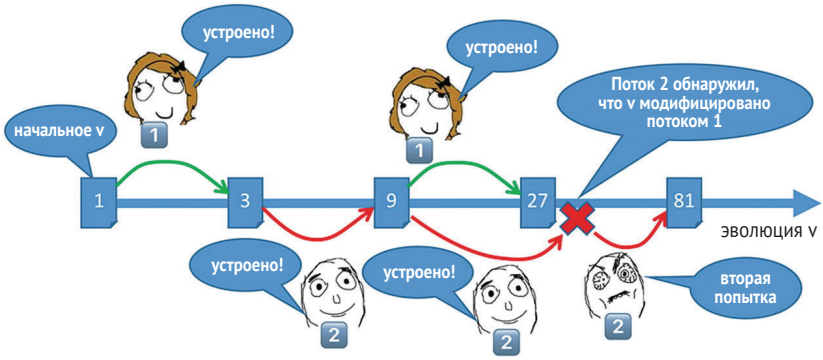


Рис. 5.19 ❖ Два потока одновременно обращаются к нашей атомарной функции `fetch_and_triple`, реализованной на основе CAS-цикла

Но вернемся к нашему сквозному примеру. Новую реализацию вычисления гистограммы можно выразить с помощью атомарных переменных, как показано на рис. 5.20.

```
#include <tbb/atomic.h>

  // Параллельное выполнение
  std::vector<tbb::atomic<int>> hist_p(num_bins);
  t0 = tbb::tick_count::now();
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
               [&](const tbb::blocked_range<size_t>& r)
               {
                 for (size_t i = r.begin(); i < r.end(); ++i)
                   hist_p[image[i]]++;
               });
```

Рис. 5.20 ❖ Третья безопасная параллельная реализация вычисления гистограммы изображения с помощью атомарных переменных

В этой реализации мы отказываемся от мьютексов и блокировок и объявляем вектор интервалов, все элементы которого имеют тип `tbb::atomic<int>` (по умолчанию инициализированы нулем). Тогда параллельный инкремент интервалов в лямбда-выражении безопасен. В итоге мы добились такого же параллельного инкремента элементов вектора гистограммы, как в мелкозернистой стратегии блокировки, но не несем расходов на управление мьютексами и их хранение.

Впрочем, с точки зрения производительности эта реализация все еще слишком медленная:

```
c++ -std=c++11 -O2 -o fig_5_20 fig_5_20.cpp -ltbb
./fig_5_20
Serial: 0.614786, Parallel: 7.90455, Speed-up: 0.0710006
```

Помимо накладных расходов на атомарный инкремент, есть еще проблемы ложного и истинного разделения, которыми мы пока не занимались. С ложным разделением мы разберемся в главе 7, когда будем говорить о распределителях с выравниванием и технике дополнения. Ложное разделение часто портит всю картину и тормозит распараллеленную программу, поэтому мы настоятельно советуем ознакомиться с рекомендуемыми приемами борьбы с ней, описанными в главе 7.

Ну хорошо, в предположении, что проблема ложного разделения решена, как быть с истинным? Два потока рано или поздно попробуют увеличить один и тот же интервал, что приведет к поочередной перезагрузке кешей. Нужна какая-то оригинальная идея!

Улучшенная параллельная реализация: приватизация и редукция

Реальная проблема в примере вычисления гистограммы связана с тем, что существует единственный разделяемый вектор, содержащий 256 интервалов, и все потоки жаждут его обновить. Пока что мы видели несколько функционально эквивалентных реализаций: крупнозернистую, мелкозернистую и на основе атомарных переменных, но ни одна из не может считаться удовлетворительной, если учитывать и такие нефункциональные требования, как производительность и энергопотребление.

Для предотвращения разделения чего-то есть общее решение – приватизировать это что-то. Параллельное программирование в этом смысле не исключение. Если у каждого потока будет частная копия гистограммы, то все они будут радостно работать с ней, кешировать в кеше ядра, на котором работают, и инкрементировать интервалы со скоростью кеша (идеальный случай). Ни тебе ложного разделения, ни истинного – вообще никакого, потому что вектор гистограммы больше не разделяется.

Замечательно, но тогда … в конечном итоге у каждого потока будет свое видение гистограммы, поскольку он посетил только часть пикселей изображения. Ничего страшного, ибо тут на сцену выходит редукция. После вычисления частных неполных версий гистограммы необходимо редуцировать вклады всех потоков и получить полную гистограмму. В этой части синхронизация все-таки присутствует, потому что некоторым потокам, возможно, придется подождать, пока другие закончат свои локальные вычисления, но, вообще говоря, такое решение оказывается намного дешевле ранее описанных. На рис. 5.21 показана техника приватизации и редукции в нашем примере.

TBB предлагает несколько способов приватизации и редукции, один основан на поточно-локальной памяти (TLS), другой – более простой в примене-

нии – на шаблоне редукции. Сначала рассмотрим вычисление гистограммы с применением TLS.

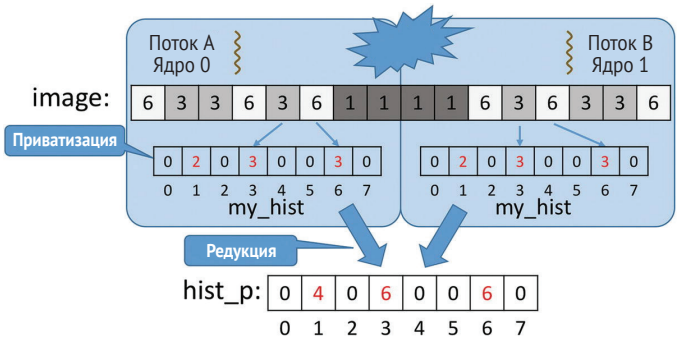


Рис. 5.21 ❖ Каждый поток вычисляет свою локальную гистограмму `my_hist`, а на втором шаге все они сводятся в одну

Поточно-локальная память

Для наших целей под поточно-локальной памятью понимается частная копия данных, хранимая в каждом потоке. TLS позволяет уменьшить количество обращений к разделяемому изменяемому состоянию со стороны нескольких потоков и задействовать локальность, т. к. каждая частная копия может (иногда частично) размещаться в локальном кеше ядра, на котором работает поток. Конечно, копии занимают место, так что следует проявлять умеренность.

Важный аспект TBB заключается в том, что нам неизвестно, сколько потоков реально используется в данный момент. Даже если мы работаем на 32-ядерной системе и используем алгоритм `parallel_for` для выполнения 32 итераций, нет никакой гарантии, что активно будет 32 потока. Это критически важно для обеспечения компонуемости кода, поскольку такой код будет работать, даже если вызывается из параллельной программы или вызывает распараллеленную функцию библиотеки (детали см. в главе 9). Поэтому мы не знаем, сколько поточно-локальных копий данных потребуется в нашем примере `parallel_for` с 32 итерациями. Шаблонные классы для поточно-локальной памяти в TBB позволяют абстрагировать запрос на выделение нужного числа копий, манипулирования ими и объединения их, не вынуждая нас беспокоиться о точном количестве. Тем самым мы получаем возможность создавать масштабируемые, компонуемые и переносимые приложения.

TBB содержит два шаблонных класса для поточно-локальной памяти. Оба предоставляют доступ к локальному элементу из потока и создают элементы по требованию (лениво). Различаются они предполагаемым способом использования:

- класс `enumerable_thread_specific` (ETS) предоставляет поточно-локальную память, которая ведет себя как STL-контейнер, содержащий по одному элементу на поток. Контейнер позволяет обходить элементы, применяя обычные для STL идиомы итерирования. Любой поток может обойти все локальные копии, видя локальные данные других потоков;

- класс `combinable` предоставляет поточно-локальную память для выполнения в каждом потоке частичных вычислений, результаты которых впоследствии будут объединены. Каждый поток видит только свои локальные данные, а также объединенные данные после вызова `combine`.

Класс enumerable_thread_specific

Сначала посмотрим, как параллельное вычисление гистограммы можно реализовать с помощью шаблонного класса `enumerable_thread_specific`. На рис. 5.22 приведен код параллельной обработки различных участков входного изображения, когда каждый поток пишет в свою частную копию вектора гистограммы.

```
#include <tbb/enumerable_thread_specific.h>

  // Параллельное выполнение
  using vector_t = std::vector<int>;
  using priv_h_t = tbb::enumerable_thread_specific<vector_t>;
  priv_h_t priv_h{num_bins};
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
               [&](const tbb::blocked_range<size_t>& r)
               {
                 priv_h_t::reference my_hist = priv_h.local();
                 for (size_t i = r.begin(); i < r.end(); ++i)

                   my_hist[image[i]]++;
               });
  // Последовательная редукция частных гистограмм
  vector_t hist_p(num_bins);

  for(auto i=priv_h.begin(); i!=priv_h.end(); ++i){
    for (int j=0; j<num_bins; ++j) hist_p[j]+=(*i)[j];
  }
```

Рис. 5.22 ❖ Параллельное вычисление частных копий гистограммы с помощью класса `enumerable_thread_specific`

Сначала мы объявляем объект `priv_h` типа `enumerable_thread_specific`, содержащий тип `vector<int>`. Конструктору передается размер вектора – `num_bins` целых чисел. Затем внутри `parallel_for` заранее неизвестное число потоков обрабатывает участки пространства итераций, и для каждого участка вычисляется тело (в нашем примере – лямбда-выражение) `parallel_for`. Поток, отвечающий за данный участок, вызывает функцию-член `my_hist = priv_h.local()`, которая работает следующим образом. Если это первый раз, когда поток вызывает `local()`, то для этого потока создается частный вектор. В противном случае вектор уже был создан, и мы используем его. В обоих случаях возвращается ссылка на частный вектор, она присваивается переменной `my_hist`, которая внутри `parallel_for` служит для обновления интервалов гистограммы данного участка.

Таким образом, поток, обрабатывающий разные участки изображения, создаст частную гистограмму для первого участка и будет использовать ее повторно для последующих. Изящно, правда?

В конце `parallel_for` мы имеем неизвестное количество частных гистограмм, которые нужно объединить для получения полной гистограммы `hist_p`. Но как выполнить редукцию, если мы даже не знаем, сколько у нас частных гистограмм? По счастью, класс `enumerable_thread_specific` не только предоставляет поточно-локальную память для элементов типа `T`, но и допускает итерирование от начала до конца, как в случае STL-контейнера. Это делается в конце кода на рис. 5.22, где переменная `i` (типа `priv_h_t::const_iterator`) последовательно обходит все частные гистограммы, а во вложенном цикле по `j` складываются значения счетчиков в отдельных гистограммах `hist_p`.

Если бы мы хотели продемонстрировать свои выдающиеся навыки программирования на C++, то могли бы воспользоваться тем фактом, что `priv_h` – тоже STL-контейнер, и записать редукцию, как показано на рис. 5.23.

```
 for (auto& i:priv_h) { // i перебирает все частные векторы
  std::transform(hist_p.begin(),     // начало источника 1
                 hist_p.end(),       // конец  источника 1
                 i.begin(),          // начало источника 2
                 hist_p.begin(),     // начало приемника
                 std::plus<int>() );// бинарная операция
}
```

Рис. 5.23 ❖ Более элегантный способ реализации редукции

Поскольку операция редукции выполняется часто, в классе `enumerable_thread_specific` есть еще две функции-члена для ее реализации: `combine_each()` и `combine()`. На рис. 5.24 показано, как использовать `combine_each`, – этот код полностью эквивалентен коду на рис. 5.23.

```
priv_h.combine_each([&](vector_t a)
 { // для каждой частной гистограммы a
   std::transform(hist_p.begin(),     // начало источника 1
                  hist_p.end(),       // конец  источника 1
                  a.begin(),          // начало источника 2
                  hist_p.begin(),     // начало приемника
                  std::plus<int>() );// бинарная операция
 });
```

Рис. 5.24 ❖ Использование `combine_each()` для реализации редукции

Прототип функции-члена `combine_each()` имеет вид:

```
template<template Func> void combine_each(Func f)
```

и, как видно на рис. 5.24, в качестве `Func f` передано лямбда-выражение, в котором STL-алгоритм `transform` отвечает за объединение частных гистограмм в `hist_p`. В общем случае функция-член `combine_each` вызывает унарный функтор

для каждого элемента в объекте enumerate_thread_specific. Эта функция объединения с сигнатурой void(T) или void(const T&) обычно редуцирует частные копии в общую переменную.

Альтернативная функция-член combine() возвращает значение типа T и имеет такой прототип:

```
template<template Func> T combine(Func f)
```

где бинарный функтор f должен иметь сигнатуру T(T, T) или T(const T&, const T&). На рис. 5.25 показана реализация редукции с сигнатурой T(T, T), в которой для каждой пары частных векторов вычисляется их сумма в векторе a, и этот вектор возвращается для возможных последующих редукций. Функция-член combine() сама позаботится о посещении всех локальных копий гистограммы, необходимых для возврата указателя на окончательную гистограмму hist_p.

```
vector_t hist_p = priv_h.combine(
  [](vector_t a, vector_t b) -> vector_t
    {
      std::transform(a.begin(),          // начало источника 1
                     a.end(),            // конец  источника 1
                     b.begin(),          // начало источника 2
                     a.begin(),          // начало приемника
                     std::plus<int>() );// бинарная операция
       return a;
    });
```

Рис. 5.25 ❖ Использование combine() для реализации той же самой редукции

И как теперь обстоит дело с производительностью?

```
c++ -std=c++11 -O2 -o fig_5_22 fig_5_22.cpp -ltbb
./fig_5_22
Serial: 0.668987, Parallel: 0.164948, Speed-up: 4.05574
```

Вот это дело! Напомним, что эксперименты проводились на 4-ядерной машине, поэтому ускорение 4.05 на самом деле суперлинейное (достигнутое благодаря агрегированию L1-кешей всех четырех ядер). Все три эквивалентные редукции на рис. 5.23, 5.24 и 5.25 выполняются последовательно, поэтому еще остается простор для повышения производительности, если количество редуцируемых частных копий велико (скажем, гистограмму вычисляют 64 потока) или если операция редукции вычислительно сложна (например, частная гистограмма содержит 1024 интервала). Мы уделим внимание этому вопросу, но сначала рассмотрим альтернативную реализацию поточно-локальной памяти.

Тип combinable

Объект типа combinable<T> предоставляет каждому потоку его собственный экземпляр типа T для хранения поточно-локальных значений во время параллельных вычислений. В отличие от описанного выше класса ETS, объект combinable не допускает итерирования, как на рис. 5.22 и 5.23 в применении к priv_h.

Но функции-члены `combine_each()` и `combine()` присутствуют, поскольку этот класс включен в TBB с единственной целью – осуществить редукцию поточно-локальной памяти.

На рис. 5.26 приведена еще одна реализация параллельного вычисления гистограммы, теперь на основе класса `combinable`.

```
#include <tbb/combinable.h>

using vector_t = std::vector<int>;
tbb::combinable<vector_t>priv_h{[](){return vector_t(num_bins);}};

parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               vector_t& my_hist = priv_h.local();
               for (size_t i = r.begin(); i < r.end(); ++i)
                 my_hist[image[i]]++;
             });


// последовательная редукция частных гистограмм
vector_t hist_p(num_bins);
priv_h.combine_each([&](vector_t a)
  { // для каждой частной гистограммы a
    std::transform(hist_p.begin(),     // начало источника 1
                   hist_p.end(),       // конец  источника 1
                   a.begin(),          // начало источника 2
                   hist_p.begin(),     // начало приемника
                   std::plus<int>() ); // бинарная операция
  });
```

Рис. 5.26 ❖ Реализация вычисления гистограммы с помощью объекта `combinable`

В данном случае `priv_h` – объект типа `combinable`, а его конструктору передается лямбда-выражение, которое будет вызываться при каждом вызове `priv_h.local()`. В нашем примере это лямбда-выражение просто создает пустой вектор, рассчитанный на `num_bins` целых чисел. Алгоритм `parallel_for`, обновляющий частные гистограммы потоков, очень похож на реализацию, показанную на рис. 5.22, с тем отличием, что `my_hist` – просто ссылка на вектор целых чисел. Как уже было сказано, теперь мы не можем обойти частные гистограммы вручную, как на рис. 5.22, но функции-члены `combine_each()` и `combine()` работают почти так же, как соответствующие функции-члены класса ETS, продемонстрированные на рис. 5.24 и 5.25. Отметим, что редукция по-прежнему выполняется последовательно, поэтому такое решение годится, только когда количество редуцируемых объектов мало и (или) мало время редукции двух объектов.

В классах ETS и `combinable` имеются и другие функции-члены, рассчитанные на более сложные применения. Все это документировано в приложении B.

Самая простая параллельная реализация: шаблон редукции

Как было сказано в главе 2, в библиотеку TBB уже включен высокоуровневый параллельный алгоритм parallel_reduce, упрощающий параллельную реализацию редукции. Так почему бы для параллельной редукции частных гистограмм просто не воспользоваться этим шаблоном? На рис. 5.27 показано, как использовать его для эффективного вычисления гистограммы.

```
#include <tbb/parallel_reduce.h>

// Параллельное выполнение
  using vector_t = std::vector<int>;
  using image_iterator = std::vector<uint8_t>::iterator;
  t0 = tbb::tick_count::now();
  vector_t hist_p = parallel_reduce (
    /*диапазон*/
    tbb::blocked_range<image_iterator>{image.begin(), image.end()},

    /*нейтральный элемент*/
    vector_t(num_bins),

    // 1-е лямбда-выражение: параллельное вычисление частных гистограмм
    [](const tbb::blocked_range<image_iterator>& r, vector_t v) {
        std::for_each(r.begin(), r.end(),
            [&v](uint8_t i) {v[i]++;});
        return v;
    },

    // 2-е лямбда-выражение: параллельная редукция частных гистограмм
    [](vector_t a, const vector_t & b) -> vector_t {
        std::transform(a.begin(),          // начало источника 1
                       a.end(),            // конец  источника 1
                       b.begin(),          // начало источника 2
                       a.begin(),          // начало приемника
                       std::plus<int>() );// бинарная операция
        return a;
    });
```

Рис. 5.27 ❖ Улучшенная параллельная реализация вычисления гистограммы изображения с использованием приватизации и редукции

Первым аргументом parallel_reduce является диапазон итерирования, который будет автоматически разбит на участки и распределен между потоками. Несколько упрощая то, что в действительности происходит под капотом, скажем, что потоки получают частные гистограммы, инициализированные нейтральным элементом операции редукции – в данном случае вектором интер-

валов, инициализированным нулями. Первое лямбда-выражение нужно для локального вычисления частичных гистограмм в результате посещения участков изображения. Второе лямбда-выражение реализует операцию редукции, которую в данном случае можно было бы выразить так:

```
[](vector_t a, const vector_t & b) -> vector_t {
    for (int i=0; i<num_bins; ++i) a[i] += b[i];
    return a;
  }
```

а это в точности то же самое, что делает STL-алгоритм `std::transform`. Время выполнения примерно такое же, как для классов ETS и `combinable`:

```
c++ -std=c++11 -O2 -o fig_5_27 fig_5_27.cpp -ltbb
./fig_5_27
Serial: 0.594347, Parallel: 0.148108, Speed-up: 4.01293
```

Чтобы пролить более яркий свет на практические последствия различных рассмотренных выше реализаций гистограммы, на рис. 5.28 мы свели воедино коэффициенты ускорения, полученные на нашем 4-ядерном процессоре. Точнее, речь идет о процессоре Core i7-6700HQ (архитектура Skylake, шестое поколение) с частотой 2.6 ГГц, L3-кешем 6 МБ и оперативной памятью 16 ГБ.

Очевидны три разных вида поведения. Небезопасное, мелкозернистое и атомарное решения при четырех ядрах значительно медленнее последовательного (значительно – это на один порядок медленнее!). Как уже было сказано, проблему составляет частая синхронизация – из-за истинного и ложного разделения и блокировок, а постоянное перебрасывание интервалов из одного кеша в другой приводит к сильно разочаровывающему ускорению. Мелкозернистое решение самое плохое, поскольку налицо истинное и ложное разделение как самого вектора гистограммы, так и вектора мьютексов. Особняком стоит крупнозернистое решение, которое лишь немногим хуже последовательного. Напомним, что это просто «распараллеленная, а затем сериализованная» версия, в которой крупнозернистая блокировка пускает потоки в критическую секцию строго по одному. Небольшое снижение производительности в этой версии обусловлено накладными расходами на распараллеливание и управление мьютексами, но зато мы избавлены от ложного и истинного разделения. Наконец, безусловными лидерами являются решения на основе приватизации + редукции (TLS и `parallel_reduce`). Они довольно хорошо масштабируются, даже лучше, чем линейно, но `parallel_reduce`, будучи чуть медленнее из-за древовидной редукции, в этой задаче не окупается. Количество ядер невелико, а время редукции (сложение векторов, содержащих 256 целых) пренебрежимо мало. Для этой тривиальной задачи вполне достаточно последовательной редукции, реализованной с помощью классов TLS.

Реализация	Небезопасная	Крупная	Мелкая	Атомарная	TLS	Редукция
Speedup:	0.09	0.99	0.02	0.07	4.05	4.01

Рис. 5.28 ❖ Ускорение, полученное для различных реализаций гистограммы на процессоре Intel Core i7-6700HQ (Skylake)

Подведем итоги

Чтобы подытожить все предложенные варианты реализации совсем простого алгоритма вычисления гистограммы, перечислим их и отметим плюсы и минусы каждого. На рис. 5.29 показаны некоторые варианты для еще более простого примера – сложения элементов вектора, содержащего 800 чисел, с применением восьми потоков. Последовательный код мог бы выглядеть так:

```
sum = 0;
for (int i = 0; i < N; ++i) sum += vec[i];
```

Как в фильме «Хороший, плохой, злой», «сценарий» этой главы можно свести к перечислению: «Заблуждающийся, опрометчивый, венценосный, ядерный, локальный и мудрый».

- Заблуждающийся: мы можем завести восемь потоков, параллельно увеличивающих глобальный счетчик `sum_g`, вообще ни о чем не думая и не испытывая никаких угрызений совести! Скорее всего, `sum_g` окажется неправильным, а протокол когерентности кешей сведет на нет все усилия повысить производительность. Мы вас предупредили.

```
long long sum_g = 0;
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const tbb::blocked_range<size_t>& r)
  {
    for (int i=r.begin(); i<r.end(); ++i) sum_g+=vec[i];
  });
```

- Опрометчивый: если использовать крупнозернистую блокировку, то результат получится правильным, но обычно ценой сериализации кода, если только мьютекс не реализует аппаратную транзакционную память (примером могут служить варианты `speculative`). Это самый простой способ защитить критическую секцию, но не самый эффективный. В нашем примере суммирования элементов вектора крупнозернистой блокировкой защищено суммирование в каждом участке вектора.

```
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const tbb::blocked_range<size_t>& r){
    my_mutex_t::scoped_lock mylock{my_mutex};
    for (int i=r.begin(); i<r.end(); ++i) sum_g+=vec[i];
  });
```

- Венценосный: мелкозернистую блокировку обычно труднее реализовать, и, как правило, она требует дополнительную память для хранения мьютексов, защищающих мелкозернистые участки структуры данных. Но есть и положительная сторона: степень конкурентности потоков возрастает. Имеет смысл попробовать различные варианты мьютексов и выбрать тот, который лучше всего показывает себя в производственном коде. При суммировании элементов вектора нет никакой структуры данных, которую можно было бы разбить на части, чтобы независимо

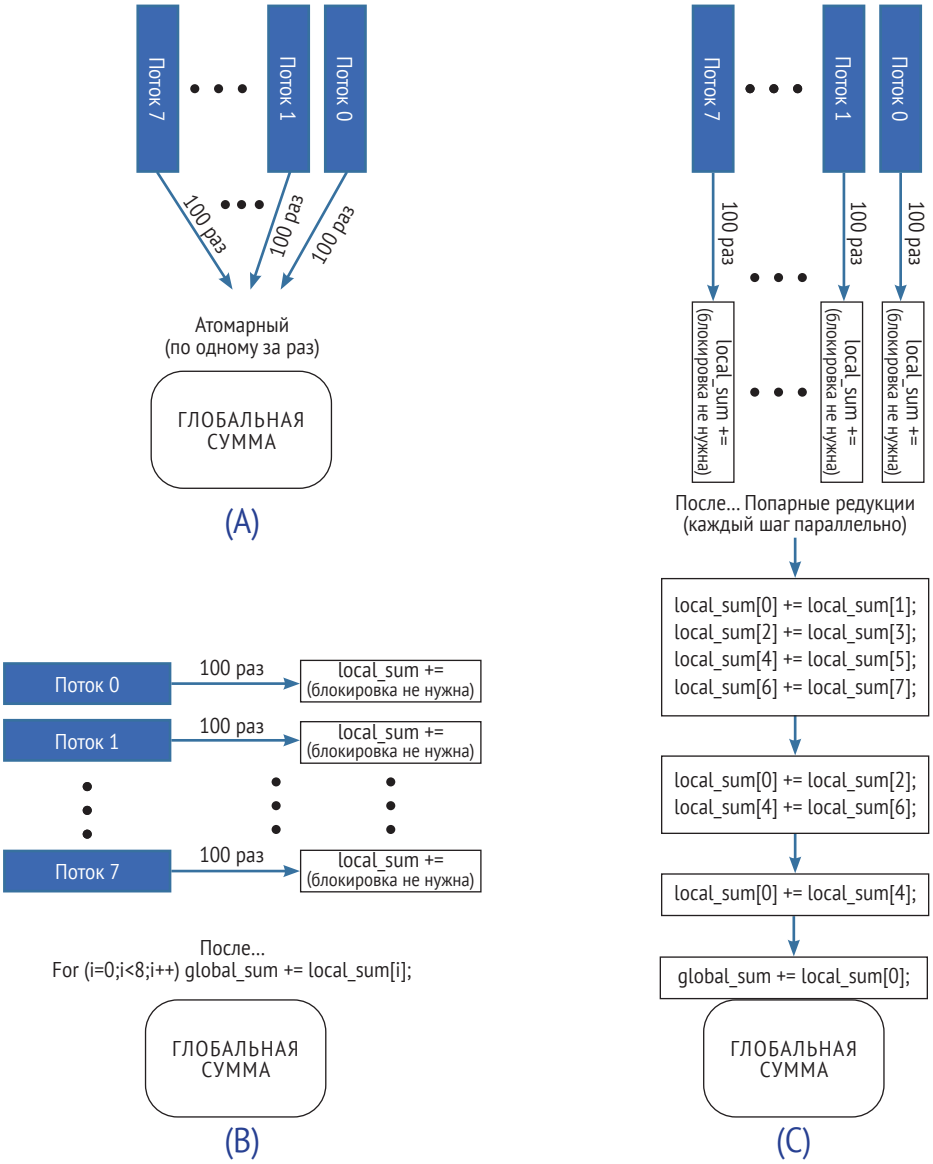


Рис. 5.29 ❖ Как избежать состязания при суммировании 800 чисел в восемь потоков:
(A) атомарный: защита глобальной суммы с помощью атомарных операций;
(B) локальный: использование `enumerable_thread_specific`;
(C) мудрый: использование `parallel_reduce`

защитить каждую часть. Назовем мелкозернистой следующую реализацию, в которой имеется облегченная критическая секция (в этом случае она такая же последовательная, как крупнозернистая, но потоки конкурируют за блокировки меньшей зернистости).

```
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const tbb::blocked_range<size_t>& r){
      for (int i=r.begin(); i<r.end(); ++i){
        my_mutex_t::scoped_lock mylock{my_mutex};
        sum_g+=vec[i];
      }
   });
```

- Ядерный: в некоторых случаях на помощь приходят атомарные переменные. Например, если разделяемое изменяемое состояние можно сохранить в целочисленном типе, а необходимая операция достаточно проста. Это дешевле мелкозернистых блокировок, а уровень конкурентности ничем не хуже. Ниже показан код для примера суммирования элементов вектора (рис. 5.29(A)) – такой же последовательный, как в двух предыдущих вариантах, и с таким же уровнем состязания за глобальную переменную, как в мелкозернистом случае.

```
tbb::atomic<long long> sum_a{0};
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const tbb::blocked_range<size_t>& r)
  {
    for (int i=r.begin(); i<r.end(); ++i) sum_a+=vec[i];
  });
```

- Локальный: не всегда удается придумать реализацию, в которой приватизация локальных копий разделяемого изменяемого состояния спасает положение. Но если удается, то реализовать поточно-локальную память (TLS) помогают классы `enumerate_thread_specific` (ETS) или `combinable`. Они работают даже тогда, когда количество взаимодействующих потоков неизвестно, и предоставляют удобные методы редукции. Эти классы достаточно гибкие, что позволяет использовать их в различных ситуациях, даже когда редукции по одному пространству итераций не хватает. На рис. 5.29(B) приведен код суммирования элементов вектора, в котором частные частичные суммы `priv_s` складываются последовательно.

```
using priv_s_t = tbb::enumerable_thread_specific<long long>;
priv_s_t priv_s{0};
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const tbb::blocked_range<size_t>& r)
  { priv_s_t::reference my_s = priv_s.local();
    for (int i=r.begin(); i<r.end(); ++i) my_s+=vec[i];
  });
long long sum_p = 0;
for (auto& i:priv_s) {sum_p+=i;}
```

- Мудрый: когда вычисление отвечает паттерну редукции, настоятельно рекомендуется воспользоваться шаблоном `parallel_reduce`, а не кодировать самостоятельно приватизацию и редукцию с помощью механизмов поточно-локальной памяти в TBB. Следующий код может показаться более запутанным, чем предыдущий, но мудрые программные архи-

текторы придумали хитроумные трюки для оптимизации этой типичной операции редукции. Например, в данном случае порядок редукции древовидный и имеет сложность $O(\log n)$, а не $O(n)$, что хорошо видно на рис. 5.29(C). Пользуйтесь тем, что дает библиотека, и не изобретайте велосипед. Этот метод, безусловно, лучше других масштабируется на большое число ядер и дорогостоящие операции редукции.

```
sum_p = parallel_reduce(tbb::blocked_range<size_t>{0, N}, 0,
 [&](const tbb::blocked_range<size_t>&r,const long long &mysum)
    {
      long long res = mysum;
      for (int i=r.begin(); i<r.end(); ++i) res+=vec[i];
      return res;
    },
 [&](const long long& a, const long long& b)
    {
      return a+b;
    });
```

Как и для вычисления гистограммы, на рис. 5.30 приведены оценки производительности различных реализаций сложения элементов вектора размера 10^9 на нашем 4-ядерном процессоре Core i7. Вычисление стало еще более мелкозернистым (всего лишь увеличение одной переменной), поэтому относительное влияние 10^9 операций блокировки-разблокировки еще сильнее, что наглядно показывает коэффициент ускорения (точнее сказать, замедления!) атомарной (Ядерный) и мелкозернистой (Венценосный) реализаций. Крупнозернистая (Опрометчивый) реализация понесла чуть больший урон, чем в случае гистограммы. Подход на основе TLS (Локальный) всего в 1,86 раза быстрее последовательного кода. Небезопасное решение (Заблуждающийся) теперь в 3,37 раза быстрее последовательного, а победителем оказалась реализация `parallel_reduction` (Мудрый), которая достигает ускорения в 3,53 раза при четырех ядрах.

Сценарий	Заблуждающийся	Опрометчивый	Венценосный	Ядерный	Локальный	Мудрый
Speedup:	3.37	0.92	0.0008	0.01	1.86	3.53

Рис. 5.30 ❖ Ускорение при различных реализациях сложения элементов вектора для $N = 10^9$ на машине с процессором Intel Core i7-6700HQ (Skylake)

Вы, наверное, не понимаете, зачем мы так подробно рассматривали все эти реализации, раз рекомендуем только последнюю. Почему сразу не перейти к решению на базе `parallel_reduce`, если оно лучшее? Увы, жизнь в параллельном мире трудна, и не все задачи распараллеливания удается решить с помощью простой редукции. В этой главе мы показали, как воспользоваться механизмами синхронизации, если они действительно необходимы, но также продемонстрировали преимущества переосмысления алгоритма и структуры данных, если такая возможность имеется.

Резюме

Библиотека TBB предлагает различные варианты мьютексов, а также атомарные переменные для синхронизации потоков в случаях, когда нужно обеспечить безопасный доступ к разделяемым данным. Библиотека также предоставляет поточно-локальную память (TLS), классы (ETS и `combinable`) и алгоритмы (`parallel_reduction`), помогающие избежать синхронизации. В этой главе мы предприняли эпическое путешествие в мир распараллеливания вычисления гистограммы изображения. На этом сквозном примере мы рассмотрели различные параллельные реализации, начиная с некорректной, осмотрев по пути несколько способов синхронизации – крупнозернистую блокировку, мелкозернистую блокировку – и заканчивая реализациями, в которых блокировки вообще не используются. По ходу дела мы останавливались для знакомства с достопримечательностями – описали свойства, по которым классифицируются мьютексы, различные виды мьютексов в TBB и типичные проблемы, возникающие при использовании мьютексов для реализации алгоритмов. И теперь, в конце путешествия, урок, вынесенный из этой главы, очевиден: пользоваться блокировками имеет смысл, только если производительность для вас не на первом месте!

Дополнительная информация

Ниже перечислены некоторые дополнительные материалы, относящиеся к теме этой главы.

- *Anthony Williams.* C++ Concurrency in action. 2-е изд. Manning Publications, 2018.
- *Daniel J. Sorin, Mark D. Hill, and David A. Wood.* A Primer on Memory Consistency and Cache Coherence. Morgan & Claypool Publishers, 2011.

Фотография городка Ронда в провинции Малага на рис. 5.1 сделана автором Рафаэлем Асенхо и используется с его разрешения.

Шаржи, встречающиеся на рисунках в главе 5, используются с разрешения сайта 365psd.com «33 Vector meme faces».

Транспортная пробка на рис. 5.17 была нарисована Денизой-Андреа Констатинеску во время аспирантуры в Малагском университете, используется с разрешения автора.

Глава 6

Структуры данных для конкурентного программирования

В предыдущей главе мы много говорили о том, как сильно не любим блокировки. Не любим, потому что они уменьшают эффективность наших параллельных программ, ограничивая их масштабируемость. Конечно, они могут оказаться «необходимым злом», нужным для корректности, однако мы предусмотрительно стремимся структурировать алгоритмы, так чтобы избегать блокировок. В этой главе описываются соответствующие средства. Главы 1–4 были посвящены масштабируемым алгоритмам. У них есть общая черта – все они избегают или сводят к минимуму блокировки. В главе 5 мы познакомились с методами явной синхронизации, в т. ч. блокировками, на случай, если без них не обойтись. В следующих двух главах мы опишем, как избежать явной синхронизации, полагаясь на средства TBB. А в этой главе обсудим структуры данных, предназначенные для отказа от блокировок, а именно конкурентные контейнеры. Смежная тема, использование поточно-локальной памяти (TLS), уже была рассмотрена в главе 5.

В этой и следующей главах рассматриваются ключевые части TBB, которые обеспечивают координацию доступа к данным из разных потоков, избегая явной синхронизации, описанной в главе 5. Тем самым мы хотим подтолкнуть вас к написанию кода способами, масштабируемость которых проверена на практике. Мы отдаем предпочтение решениям, тщательно продуманным разработчиками TBB (чтобы подчеркнуть важность этого для корректности программы, мы обсудим проблему A-B-A). Следует всегда помнить, что выбор алгоритма может оказать решающее влияние на производительность параллельной программы и простоту ее реализации.

Выбирайте алгоритмы с умом: конкурентные контейнеры – не панацея

Параллельный доступ к данным особенно хорошо работает, когда проистекает из ясной стратегии распараллеливания, главная часть которой – правильный выбор алгоритмов. У контролируемого доступа, предлагаемого, в частности, конкурентными контейнерами, есть цена: сделать контейнер «высококонкурентным» – удовольствие не бесплатное и даже не всегда возможное. TBB предлагает конкурентные контейнеры, когда такая поддержка хорошо работает на практике (очереди, хеш-таблицы и векторы). TBB не пытается поддержать конкурентность для контейнеров типа «список» или «дерево», для которых мелкозернистое разделение плохо масштабируется, – лучше искать возможность для распараллеливания на пути пересмотра алгоритмов и (или) структур данных.

Конкурентные контейнеры – это потокобезопасные варианты контейнеров, которые могут успешно работать в параллельных программах. Это высокопроизводительная альтернатива последовательному контейнеру, обернутому крупнозернистой блокировкой (см. обсуждение в главе 5). TBB-контейнеры обычно предоставляют мелкозернистую блокировку или безблокировочную реализацию, а иногда то и другое сразу.

Основы важнейших структур данных

Если вы знакомы с хеш-таблицами, неупорядоченными отображениями, неупорядоченными множествами, очередями и векторами, то можете пропустить этот раздел и перейти сразу к разделу «Конкурентные контейнеры». Чтобы освежить в памяти фундаментальные положения, мы предлагаем краткое введение в ключевые структуры данных, прежде чем рассказывать о том, как они приспособлены в TBB для параллельного программирования.

Неупорядоченные ассоциативные контейнеры

У *неупорядоченных ассоциативных контейнеров* есть более простое название – *коллекция*. Мы будем также называть их «множествами». Но для различных типов коллекций в техническом лексиконе имеются и отдельные наименования: отображение, множество и хеш-таблица.

Ассоциативные контейнеры – это структуры данных, которые позволяют по *ключу* найти *значение*, ассоциированное с этим ключом. Их можно рассматривать как своего рода массив, поэтому применяется название «ассоциативный массив». Только индексы в таком массиве сложнее, чем простая последовательность чисел: вместо `Cost[1]`, `Cost[2]`, `Cost[3]` речь может идти о `Cost[стакан сока]`, `Cost[буханка хлеба]`, `Cost[щенок на витрине]`.

Наши ассоциативные контейнеры можно классифицировать по двум признакам:

1. **Отображение** или **множество**: существует ли *значение*? Или только *ключ*?
2. **Несколько значений**: можно ли вставить в одну коллекцию два элемента с одинаковыми *ключами*?

Отображение или множество

«Отображением» мы называем «множество» с присоединенными значениями. Представьте себе корзину фруктов (яблоко, апельсин, банан, груша, лимон). Имея *множество*, содержащее фрукты, мы можем сказать, есть ли в корзине определенный вид фруктов. Только *да* или *нет*. Можно добавить новый вид фруктов в корзину или удалить его из корзины. *Отображение* добавляет к этому значение, которое часто само является структурой данных, содержащей информацию. Имея *отображение* типа фруктов на коллекцию (корзину фруктов), мы можем сохранить количество, цену и прочую информацию. Вместо простого *да* или *нет* можно задавать вопросы о стоимости `Cost[яблоко]` или зрелости `Ripeness[банан]`. Если значением является структура, содержащая несколько полей, то можно спрашивать сразу о нескольких атрибутах, например стоимости, зрелости и цвете.

Несколько значений

Вставлять в обычное отображение или множество элемент с таким же *ключом*, как у уже находящегося в контейнере, запрещено (гарантия уникальности). Однако это разрешено для «мультимножества» и «мультиотображения». В контейнерах с приставкой «мульти» дублирование допустимо, но при этом мы теряем возможность запросить что-то вроде `Cost[яблоко]`, потом что ключ «яблоко» уже не единственный.

Хеширование

Все вышеупомянутое (ассоциативные массивы, множества и отображения, обычные и мульти) обычное реализуется с помощью *хеш-функций*. Чтобы понять, что это такое, лучше всего начать с мотивации. Рассмотрим ассоциативный массив `LibraryCardNumber[имя читателя]`. Массив `LibraryCardNumber` возвращает номер библиотечной карточки по имени читателя (заданного в виде строки символов), которое играет роль индекса. Один из способов реализовать ассоциативный массив – воспользоваться связанным списком элементов. К сожалению, для поиска в нем пришлось бы просматривать все элементы по очереди. В худшем случае придется обойти весь список, что в параллельной программе в высшей степени неэффективно из-за состязания за доступ к разделяемой структуре списка. Но и без всякого распараллеливания для проверки отсутствия элемента с данным ключом при вставке пришлось бы просмотреть весь список. Если в списке тысячи или миллионы читателей, то на это понадобится уйма времени. Более экзотические структуры данных, например деревья, могут разрешить некоторые, но не все перечисленные проблемы.

Теперь представим себе очень большой массив, в который можно помещать данные. К этому массиву мы обращаемся как обычно – `массив[целое]`. Это очень быстро. Нужна только волшебная *хеш-функция*, которая принимает индекс ассоциативного массива (*имя читателя*) и преобразует его в нужное нам целое число.

Неупорядоченность

Характеристику обсуждаемых *ассоциативных контейнеров* мы начали словом *неупорядоченный*. Конечно, можно было бы отсортировать ключи и обращаться к контейнерам по порядку. Ничто не мешает. Например, в роли ключа может выступать имя человека, и мы хотим создать телефонный справочник в алфавитном порядке.

Слово *неупорядоченный* в этом контексте не означает, что мы не можем программировать с учетом порядка. Оно лишь означает, что сама структура данных (контейнер) не поддерживает порядок. Если существует способ обойти контейнер (в терминологии C++ – *итерировать*), то гарантируется лишь, что каждый элемент будет посещен ровно один раз, но никакой конкретный порядок обхода не гарантируется, он может меняться от выполнения к выполнению, от машины к машине и т. д.

Конкурентные контейнеры

TBB предоставляет высококонкурентные классы контейнеров, полезные во всех многопоточных приложениях на C++; эти классы можно использовать совместно с любым методом реализации многопоточности, включая, разумеется, саму TBB!

Стандартная библиотека шаблонов C++ изначально проектировалась без учета конкурентности. Как правило, STL-контейнеры не поддерживают конкурентные обновления, так что такая попытка может привести к повреждению контейнера. Конечно, STL-контейнер можно обернуть крупнозернистым мьютексом, сделав его безопасным для конкурентного доступа, тогда в каждый момент времени с контейнером сможет работать только один поток. Но этот подход исключает всякую конкурентность, а значит, ограничивает коэффициент ускорения при распараллеливании, если применяется в критическом для производительности участке кода. Примеры защиты с помощью мьютексов были приведены в главе 5, где мы использовали их для защиты инкремента интервалов гистограммы. Аналогичная методика годится и для защиты потоконебезопасных STL-алгоритмов с целью избежать проблем с корректностью. Если это делается не в критических для производительности участках, то влияние на производительность может оказаться минимальным. Это важный момент: преобразование обычных контейнеров в конкурентные TBB-контейнеры следует предпринимать, только если это продиктовано необходимостью. Структуры данных, используемые параллельно, следует проектировать с учетом конкурентности для обеспечения масштабируемости приложений.

Конкурентные контейнеры в TBB предоставляют функциональность, аналогичную STL-контейнерам, но потокобезопасным способом. Например, класс `tbb::concurrent_vector` аналогичен классу `std::vector`, но позволяет увеличивать размер вектора параллельно. Конкурентный контейнер не нужен, если мы собираемся только читать его параллельно. Он нужен тогда, когда параллельный код модифицирует контейнер.

TBB предлагает несколько контейнерных классов, призванных заменить соответствующие STL-контейнеры, сохранив совместимость, чтобы несколько потоков могли одновременно вызывать методы контейнера. У TBB-контейнеров уровень конкурентности гораздо выше, что достигается следующими способами – каким-то одним или совместно:

- мелкозернистая блокировка: несколько потоков работают с контейнером, блокируя только те его части, которые действительно нужно блокировать (как в примере с гистограммой из главы 5). Коль скоро разные потоки обращаются к разным частям, они могут работать конкурентно;
- безблокировочные техники: потоки учитывают и исправляют возможные повреждения со стороны параллельно работающих потоков.

Отметим, что конкурентные TBB-контейнеры все равно обходятся недаром, хотя и недорого. Как правило, у них более высокие накладные расходы, чем у обычных STL-контейнеров, поэтому операции могут занимать немного больше времени. Если существует вероятность одновременного доступа, то следует использовать конкурентные контейнеры. В противном случае лучше ограничиться традиционными STL-контейнерами. То есть использовать конкурентные контейнеры имеет смысл, если ускорение в результате дополнительной обеспечиваемой ими конкурентности перевешивает более низкую собственную производительность.

Интерфейсы контейнеров остались такими же, как в STL, за исключением тех случаев, когда изменения необходимы для поддержки конкурентности. Мы можем забежать вперед и, воспользовавшись случаем, рассмотреть классический пример, показывающий, почему некоторые интерфейсы не являются потокобезопасными – *и это очень важно понимать*! Классический пример (см. рис. 6.9) – необходимость в операции *извлечь-если-непуст* (она называется `try_pop`) для очередей взамен применяемой в STL последовательности двух операций: *проверить-что-непуст* и *извлечь*, если оказалось, что очередь не пуста. Опасность такого STL-кода заключается в том, что другой поток может вклиниться, опустошить контейнер (после проверки, но перед извлечением) и тем самым создать состояние гонки, при котором операция извлечения может быть заблокирована. Это означает, что STL-код потоконебезопасен. Можно было бы окружить блокировкой всю последовательность, чтобы предотвратить модификацию очереди между проверкой и извлечением, но известно, что такие блокировки вредят производительности при использовании в распараллеленных частях приложения. Разобравшись в этом простом примере, вы станете лучше понимать, что необходимо для правильной поддержки параллелизма.

Как и в STL, TBB-контейнеры – шаблоны, одним из аргументов которых является распределитель памяти. Контейнер использует этот распределитель для выделения памяти под видимые пользователю элементы. По умолчанию в TBB используется масштабируемый распределитель, входящий в состав библиотеки (обсуждается в главе 7). Но для выделения памяти под внутренние структуры может использоваться другой распределитель.

В настоящее время TBB предлагает следующие конкурентные контейнеры:

- неупорядоченные ассоциативные контейнеры:
  - неупорядоченное отображение (включая неупорядоченное мультиотображение);

  - неупорядоченное множество (включая неупорядоченное мультимножество);
  - хеш-таблица;
- очередь (включая ограниченную очередь и очередь с приоритетами);
- вектор.

Почему в TBB-контейнерах по умолчанию используется распределитель TBB

Аргумент-распределитель поддерживается всеми TBB-контейнерами, и по умолчанию используется масштабируемый распределитель TBB (см. главу 7).

По умолчанию в контейнерах используется либо `tbb::cache_aligned_allocator`, либо `tbb:tbb_allocator`. В этой главе мы будем уточнять, какой именно, но, вообще говоря, для справки лучше использовать заголовочные файлы TBB и приложение B к этой книге. Прикомпоновывать к программе библиотеку, содержащую масштабируемый распределитель TBB, нет необходимости, поскольку если эта библиотека отсутствует, то TBB-контейнеры по умолчанию используют `malloc`. Однако мы рекомендуем компоновать с масштабируемым распределителем TBB, потому что при этом производительность, вероятно, повысится; особенно просто сделать это, используя ее как прокси-библиотеку (см. главу 7).

Имя класса и замечания, касающиеся C++11	Конкурентный обход и вставка	С ключами ассоциированы значения	Поддержка конкурентного стирания	Встроенная блокировка	Отсутствие видимой блокировки (безблокировочный интерфейс)	Разрешена вставка одинаковых элементов	Акцессоры [] и at
`concurrent_hash_map` Предшествует C++11	✓	✓	✓	✓	×	×	×
`concurrent_unordered_map` *Очень похож на unordered_map в C++11*	✓	✓	×	×	✓	×	
`concurrent_unordered_multimap` *Очень похож на unordered_multimap в C++11*	✓	✓	×	×	✓	×	
`concurrent_unordered_set` *Очень похож на unordered_set в C++11*	✓	×	×	×	✓	×	×
`concurrent_unordered_multiset` *Очень похож на unordered_multiset в C++11*	✓	×	×	×	✓	✓	×

Рис. 6.1 ❖ Сравнение конкурентных неупорядоченных ассоциативных контейнеров

Конкурентные неупорядоченные ассоциативные контейнеры

Неупорядоченные ассоциативные контейнеры – это группа шаблонов классов, реализующая различные варианты хеш-таблиц. Эти контейнеры и их отличительные свойства перечислены на рис. 6.1. В конкурентных неупорядоченных ассоциативных контейнерах можно хранить произвольные элементы, например целые числа или объекты пользовательских классов, поскольку это шабло-

ны. TBB предлагает реализации неупорядоченных ассоциативных контейнеров, способные хорошо работать при конкурентном доступе.

Хеш-отображение (которое также часто называют хеш-таблицей) – это структура данных, отображающая ключи на значения с помощью хеш-функции. Хеш-функция вычисляет индекс по ключу, и этот индекс используется для доступа к ячейке (bucket), в которой хранится ассоциированное с ключом значение (или значения).

Выбор хорошей хеш-функции очень важен! Идеальная хеш-функция сопоставляла бы каждому ключу уникальную ячейку, поэтому *коллизий*, когда двум ключам соответствует одна ячейка, не было бы вообще. Но на практике хеш-функции не идеальны и иногда генерируют одинаковые индексы для разных ключей. Реализация хеш-таблицы должна быть готова к таким коллизиям, и это приводит к некоторым издержкам, поэтому хеш-функции следует проектировать, так чтобы минимизировать количество коллизий, т. е. распределять входные данные по ячейкам почти равномерно.

Преимущество хеш-отображений связано с тем, что в среднем время поиска и вставки составляет $O(1)$. А у хеш-отображения из библиотеки TBB имеется и дополнительное преимущество – поддержка конкурентного доступа в плане корректности и производительности. Это предполагает использование хорошей хеш-функции – не приводящей к большому числу коллизий на множестве возможных ключей. Если хеш-функция неидеальна или размер хеш-таблицы выбран неудачно, то теоретическая возможность времени поиска $O(n)$ в худшем случае может стать реальностью.

На практике хеш-отображения часто оказываются более эффективны, чем другие структуры данных для поиска, в т. ч. деревья поиска. Поэтому хеш-отображения применяются во многих ситуациях, в т. ч. в ассоциативных массивах, для индексирования баз данных, реализации кешей и множеств.

---

concurrent_hash_map

TBB включает класс `concurrent_hash_map`, который отображает ключи на значения таким образом, что несколько потоков могут одновременно обращаться к значениям с помощью методов `find`, `insert` и `erase`. Поскольку `tbb::concurrent_hash_map` проектировался в расчете на параллелизм, его интерфейсы потокобезопасны, в отличие от интерфейсов `map` и `set` в STL, которые мы рассмотрим ниже в этой главе.

Ключи не упорядочены. В `concurrent_hash_map` может быть не более одного элемента с данным ключом. Тип `HashCompare` определяет способ хеширования ключей *и* их сравнения на равенство. Для любой хеш-таблицы ожидается, что если ключи равны, то их хеш-коды одинаковы. Именно поэтому `HashCompare` соединяет концепции сравнения и хеширования в одном объекте, а не рассматривает их по отдельности. Еще одно следствие этого постулата – запрет на изменение хеш-кода ключа, если хеш-таблица не пуста.

Класс `concurrent_hash_map` работает как контейнер элементов типа `std::pair <const Key,T>`. Обычно при доступе к элементу контейнера нас интересует либо его обновление, либо чтение. Шаблонный класс `concurrent_hash_map` поддерживает обе операции с помощью классов `accessor` и `const_accessor` соответственно, которые ведут себя как интеллектуальные указатели. Объект `accessor` представляет доступ для обновления (записи). Пока он указывает на некоторый элемент, все остальные попытки найти тот же ключ в таблице блокируются. Объект `const_accessor` аналогичен, но представляет доступ только для чтения.

Несколько акцессоров могут одновременно указывать на один и тот же элемент. Эта возможность позволяет заметно улучшить производительность в ситуациях, когда элементы часто читаются, но редко обновляются.

На рис. 6.2 и 6.3 приведены фрагменты единой программы, в которых демонстрируется использование контейнера `concurrent_hash_map`. Производительность этого примера можно повысить, уменьшив промежуток времени, в течение которого происходит доступ к элементу. Методы `find` и `insert` принимают объект `accessor` или `const_accessor` в качестве аргумента. От выбора типа зависит, для чего мы обращаемся к `concurrent_hash_map`: для обновления или для чтения. После возврата из метода доступ продолжается, пока объект `accessor` или `const_accessor` не будет уничтожен. Поскольку наличие доступа к элементу может блокировать другие потоки, старайтесь сократить время жизни `accessor`

```
#include <tbb/concurrent_hash_map.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <string>

// Структура, определяющая операции хеширования и сравнения
для пользовательского типа.
struct MyHashCompare {
  static size_t hash( const std::string& x ) {
    size_t h = 0;
    for( const char* s = x.c_str(); *s; ++s )
      h = (h*17)^*s;
    return h;
  }
  // ! true, если строки равны.
  static bool equal( const std::string& x, const std::string& y
) {
    return x==y;
  }
};

// Конкурентная хеш-таблица для отображения string на int.
typedef tbb::concurrent_hash_map<std::string,int,MyHashCompare>
StringTable;

// Объект-функция для подсчета количества вхождений строк.
struct Tally {
  StringTable& table;
  Tally( StringTable& table_ ) : table(table_) {}
  void operator()(
    const tbb::blocked_range<std::string*> range ) const {
    for( std::string* p=range.begin(); p!=range.end(); ++p ) {
      StringTable::accessor a;
      table.insert( a, *p );
      a->second += 1;
    }
  }
};
```

Рис. 6.2 ❖ Пример хеш-таблицы, часть 1 из 2

или const_accessor. Для этого объявляйте его в самом внутреннем блоке. Если нужно прекратить доступ еще до выхода из блока, пользуйтесь методом release. На рис. 6.5 показано переработанное тело цикла на рис. 6.2, в котором метод release используется вместо того, чтобы ждать уничтожения по выходе из области видимости. Метод remove(key) также может работать конкурентно. Неявно он запрашивает доступ для записи. Поэтому перед тем как удалять ключ, он ждет завершения еще активных операций доступа.

```
const size_t N = 10;

std::string Data[N] = { "Hello", "World", "TBB", "Hello",
      "So Long", "Thanks for all the fish", "So Long",
      "Three", "Three", "Three" };

void main() {
  // Конструируем пустую таблицу.
  StringTable table;

  // Поместить вхождения в таблицу.
  tbb::parallel_for(
      tbb::blocked_range<std::string*>( Data, Data+N, 1000 ),
      Tally(table) );

  // Отображаем вхождения в процессе простого обхода
  // (примечание: concurrent_hash_map не предоставляет const_iterator)
  // Вы видите в этом коде проблему?
  // читайте раздел «Обход этих структур –
  // приглашение к неприятностям»
  for( StringTable::iterator i=table.begin();
       i!=table.end();
       ++i )
    printf("%s %d\n",i->first.c_str(),i->second);
}
```

Рис. 6.3 ❖ Пример хеш-таблицы, часть 2 из 2

```
Three 3
So Long 2
Hello 2
TBB 1
World 1
Thanks for all the fish 1
```

Рис. 6.4 ❖ Вывод программы, показанной на рис. 6.2 и 6.3

```
for( std::string* p=range.begin(); p!=range.end(); ++p ) {
  StringTable::accessor a;
  table.insert( a, *p );
  a->second += 1;
  a.release();
}
```

Рис. 6.5 ❖ Переработка кода рис. 6.2 для уменьшения времени жизни акцессора в надеже улучшить масштабируемость

Советы по повышению производительности хеш-отображений

- Всегда задавайте начальный размер хеш-таблицы. Подразумеваемый по умолчанию размер 1 – катастрофа для масштабируемости! Хороший размер начинается с нескольких сотен. Если вам кажется, что меньший размер – это правильно, то использование блокировки для небольшой таблицы будет предпочтительнее с точки зрения производительности благодаря локальности кеша.
- Проверяйте свою хеш-функцию – и удостоверьтесь, что псевдослучайность младших битов хеш-значения удовлетворительна. В частности, не следует использовать указатели в качестве ключей, потому что младшие биты указателя часто равны нулю из-за выравнивания объектов в памяти. Если это так, то настоятельно рекомендуется разделить указатель на размер типа объекта, на который он указывает, сдвинув тем самым нулевые биты за пределы видимости и оставив те, которые действительно изменяются. Стоит рассмотреть в качестве стратегии умножение на простое число и сдвиг вправо на несколько битов. Как в любой хеш-таблице, равные ключи должны иметь одинаковые хеш-коды, а идеальная хеш-функция распределяет ключи равномерно по пространству хеш-кодов. Подбор оптимальной хеш-функции, безусловно, зависит от приложения, но функции, предоставляемые TBB по умолчанию, обычно работают хорошо.
- Не пользуйтесь акцессорами, если этого можно избежать, а если нельзя, то старайтесь ограничить время их жизни (см. пример на рис. 6.5). По существу, это мелкозернистые блокировки, которые не дают работать другим потокам, пока существуют, а значит, потенциально ограничивают масштабируемость.
- Пользуйтесь распределителями памяти, предоставляемыми TBB (см. главу 7). Указывайте `scalable_allocator` в качестве аргумента шаблона контейнера, если хотите, чтобы использовался именно он (а не `malloc`).

Поддержка конкурентности в интерфейсах map/multimap и set/multiset

В стандарте C++ STL определены типы `unordered_set`, `unordered_map`, `unordered_multiset` и `unordered_multimap`. Они различаются только ограничениями на элементы. Рисунок 6.1 – удобный справочник для сравнения пяти вариантов поддержки конкурентных отображений и множеств, в т. ч. класса `tbb::concurrent_hash_map`, который использовался в примерах кода на рис. 6.2–6.5.

В STL не определено ничего, содержащего слова «hash», потому что в C++ изначально не определена хеш-таблица. Интерес к включению поддержки хеш-таблиц в STL был настолько велик, что во многих версиях STL была расширена в этом направлении; это относится к компиляторам SGI, gcc и Microsoft. Но в отсутствие стандарта расплодились вариации на тему того, что понимать под

«хеш-таблицами» и «хеш-отображениями» в плане возможностей и производительности. Начиная с C++11 хеш-таблица была добавлена в STL, и для класса выбрали имя `unordered_map`, чтобы предотвратить путаницу и конфликты имен с достандартными реализациями. Можно даже сказать, что имя `unordered_map` более информативное, потому что содержит подсказку об интерфейсе класса и неупорядоченной природе его элементов.

Оригинальная хеш-таблица в TBB предшествует C++11 и называется `tbb::concurrent_hash_map`. Она сохраняет ценность и не была изменена для приведения в соответствие со стандартом. Но теперь TBB поддерживает также `unordered_map` и `unordered_set`, чтобы отразить добавления в C++11, а интерфейсы дополнены или адаптированы только в той мере, которая необходима для поддержки конкурентного доступа. Отказ от некоторых непригодных для распараллеливания интерфейсов подталкивает нас к изучению эффективного параллельного программирования. В приложении В детали изложены исчерпывающим образом, а здесь отметим лишь три изменения с целью улучшить масштабируемость параллельного кода:

- опущены методы, возвращающие типы, добавленные в C++11 (например, ссылки на rvalue);
- методы `erase` стандартных контейнеров C++ имеют префикс `unsafe_`, чтобы подчеркнуть их небезопасность в конкурентном коде (т. к. конкурентное стирание поддерживается только в классе `concurrent_hash_map`). Это не относится к `concurrent_hash_map`, потому что он-то как раз поддерживает конкурентное стирание;
- методы, относящиеся к ячейкам (подсчет ячеек, максимальное число ячеек, размер ячеек и поддержка обхода ячеек), имеют префикс `unsafe_` как напоминание об их небезопасности относительно вставки. Они сохранены ради совместимости с STL, но по возможности их следует избегать. А уж если никак не обойтись, то они должны быть защищены от использования параллельно со вставкой. Эти интерфейсы отсутствуют в `concurrent_hash_map`, потому что проектировщики TBB сознательно избегали таких функций.

Встроенная блокировка и невидимая блокировка

Контейнеры `concurrent_hash_map` и `concurrent_unordered_*` различаются в том, что касается блокировки элементов при доступе. Поэтому они могут вести себя совершенно по-разному в условиях состязания. Акцессоры `concurrent_hash_map` по существу являются блокировками: тип `accessor` – монопольная блокировка, а `const_accessor` – разделяемая блокировка. Синхронизация на основе блокировок встроена в модели использования контейнера и защищает не только целостность самого контейнера, но и до некоторой степени целостность данных. В коде на рис. 6.2 акцессор используется при выполнении вставки в таблицу.

Обход этих структур – приглашение к неприятностям

Мы тайком включили небезопасный относительно конкурентности код в конец примера на рис. 6.3 – при обходе хеш-таблицы для ее распечатки. Если бы во время обхода производились операции вставки или удаления, то возникли бы проблемы. В свою защиту можем лишь сказать: «Это отладочный код – пле-

вать!» Но опыт научил нас, что такой код легко протащить и в производственный код. Будьте бдительны!

Проектировщики TBB оставили итераторы в `concurrent_hash_map` для целей отладки, но позаботились о том, чтобы не вводить нас в искушение вернуть итератор в качестве значения других функций-членов.

К сожалению, STL подталкивает нас совершить действия, которым лучше бы противиться. Контейнеры `concurrent_unordered_*` отличаются от `concurrent_hash_map` – их API следует стандарту C++ для ассоциативных контейнеров (помните, оригинальный контейнер `concurrent_hash_map` появился в TBB до стандартизации конкурентных контейнеров в C++). Операции добавления и поиска данных возвращают итератор, искушая нас воспользоваться им для обхода. В параллельной программе мы рискуем тем, что этот обход будет производиться одновременно с другими операциями над отображением (множеством). Если мы поддадимся искушению, то защита целостности данных целиком ложится на нас, API контейнера тут ничем не поможет. Можно было бы сказать, что стандарт контейнеров C++ допускает дополнительную гибкость по сравнению с `concurrent_hash_map`, но уступает ему во встроенной защите. Интерфейсы STL достаточно легко использовать конкурентно, если избегать соблазна воспользоваться итераторами, которые возвращают операции добавления и поиска, для чего-либо, кроме ссылки на найденный элемент. Если поддаться этому соблазну (не надо этого делать!), то нужно тщательно продумывать все конкурентные обновления. Разумеется, если никаких обновлений нет (а только поиск), то никаких осложнений с итераторами в параллельной программе не возникает.

Конкурентные очереди: обычные, ограниченные и с приоритетами

Очереди – полезные структуры данных, данные помещаются в них операцией `push` (`add`), а извлекаются – операцией `pop` (`remove`). В интерфейсе неограниченной очереди имеется операция `try pop`, которая может сообщить, что очередь пуста и никакого значения извлечь не удалось. Идея в том, чтобы отвадить нас от попыток писать собственный код, цель которого – избежать блокировки операции извлечения путем проверки очереди на пустоту – эта операция потоконебезопасна (см. рис. 6.9). Совместное использование очереди несколькими потоками может оказаться эффективным способом передачи данных от одного потока другому – в очереди «работ» могут храниться элементы, которые различные задачи извлекают для обработки. Обычно очередь обслуживается в порядке «первым пришел – первым ушел» (FIFO). Если вначале была пустая очередь, а затем были выполнены операции `push(10)` и `push(25)`, то первая операция `pop` вернет 10, а вторая – 25. Это прямо противоположно поведению стека, который обслуживается в порядке «первым пришел – последним ушел». Но мы сейчас говорим не о стеках!

На рис. 6.6 приведен простой пример, который ясно показывает, что операции `pop` возвращают значения в том же порядке, в котором операции `push` поместили их в очередь.

```
#include <tbb/concurrent_queue.h>
#include <tbb/concurrent_priority_queue.h>
#include <iostream>

int myarray[10] = { 16, 64, 32, 512, 1, 2, 512, 8, 4, 128 };

void pval(int test, int val) {
  if (test) {
    std::cout << " " << val;
  } else {
    std::cout << " ***";
  }
}

void simpleQ() {
  tbb::concurrent_queue<int> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Simple  Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

int main() {
  simpleQ();
//  boundedQ();
//  prioQ();
//  prioQgt();
  return 0;
}

Output is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
```

Рис. 6.6 ❖ Пример использования простой FIFO-очереди

Есть два специальных вида очередей: *ограниченная* и *с приоритетами*. Под *ограничением* понимается ограничение размера очереди. Это значит, что `push` не сможет поместить новый элемент, если очередь заполнена. Чтобы справиться с этой проблемой, интерфейс ограниченной очереди предлагает два выхода: подождать, пока в очереди появится место, или выполнить операцию `try_push`, которая помещает элемент, если может, а в противном случае уведомляет о том, что очередь заполнена. Ограниченная очередь по умолчанию неограниченна! Если нам все-таки нужна ограниченная очередь, то нужно использовать класс

concurrent_bounded_queue и вызвать метод set_capacity для задания размера очереди. На рис. 6.7 показан простой пример использования ограниченной очереди, в которую удалось успешно поместить только первые шесть элементов. Можно было бы проверить значение, возвращенное try_push, и что-то сделать. В данном случае программа печатает ***, когда операция pop обнаруживает, что очередь пуста.

```
void boundedQ() {
  tbb::concurrent_bounded_queue<int> queue;
  int val;

  queue.set_capacity(6);

  for( int i=0; i<10; ++i )
    queue.try_push(myarray[i]);

  std::cout << "Bounded Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

Output of the expanded program is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q   pops are 16 64 32 512 1 2 *** *** *** ***
```

Рис. 6.7 ❖ Эта функция добавляет в нашу программу пример использования ограниченной очереди

Очередь с приоритетами модифицирует дисциплину «первым пришел – первым ушел», добавляя сортировку. По умолчанию, если мы ничего не задали, для определения приоритета используется шаблон std::less<T>. Это означает, что операция pop вернет элемент с наибольшим значением.

На рис. 6.8 приведены два примера очереди с приоритетами, в одном по умолчанию используется std::less<int>, а в другом явно задана функция сравнения std::greater<int>.

Как видно из приведенных примеров, для реализации трех разновидностей очередей TBB предлагает три класса контейнеров: concurrent_queue, concurrent_bounded_queue и concurrent_priority_queue. Все конкурентные очереди разрешают нескольким потокам одновременно помещать и извлекать элементы. Интерфейсы похожи на имеющиеся в STL классы std::queue и std::priority_queue, но отличаются там, где для безопасного поведения в условиях конкурентности необходимы модификации.

Основными методами очереди являются push и try_pop. Метод push работает так же, как в std::queue. Важно отметить, что методы front и back не поддержи-

ваются, потому что в конкурентном окружении небезопасно возвращать ссылку на элемент очереди. В параллельной программе начало и конец очереди могут быть изменены другим потоком, так что сами понятия начала и конца становятся бессмысленными.

```
void prioQ() {
  tbb::concurrent_priority_queue<int> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Prio    Q   pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

void prioQgt() {
  tbb::concurrent_priority_queue<int,std::greater<int>> queue;
  int val;

  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);

  std::cout << "Prio    Qgt pops are";

  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );

  std::cout << std::endl;
}

Output of the expanded program is:
Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q   pops are 16 64 32 512 1 2 *** *** *** ***
Prio    Q   pops are 512 512 128 64 32 16 8 4 2 1
Prio    Qgt pops are 1 2 4 8 16 32 64 128 512 512
```

Рис. 6.8 ❖ Эти две функции добавляют в программу примеры очередей с приоритетами

Аналогично для неограниченных очередей не поддерживается метод `pop` и проверка на пустоту – вместо них определен метод `try_pop`, который извлекает элемент, если он есть, и тогда возвращает состояние `true`, а в противном случае ничего не извлекает и возвращает состояние `false`. Проверка на пустоту

и извлечение объединены в один метод, чтобы было проще писать потокобезопасный код. Для ограниченных очередей определен неблокирующий метод try_push в добавление к потенциально блокирующему методу push. Это позволяет не обращаться к методу size для запроса о длине очереди. В общем случае методов size следует избегать, особенно если это пережитки последовательной программы. Поскольку в параллельной программе длина очереди может изменяться одновременно с запросом, использование метода size следует тщательно продумывать. Кстати, TBB может вернуть отрицательное значение из метода size, если очередь пуста и существуют ожидающие вызовы метода pop. Метод empty возвращает true, когда размер меньше или равен нулю.

Ограничение размера

В классах concurrent_queue и concurrent_priority_queue емкость очереди ограничена только объемом памяти компьютера. Класс concurrent_bounded_queue позволяет контролировать размер – его главная особенность состоит в том, что метод push блокирует поток до тех пор, пока в очереди не освободится место. Ограниченная очередь полезна тем, что замедляет поставщика с учетом темпа потребления, не позволяя очереди расти до бесконечности.

Класс concurrent_bounded_queue – единственный контейнер вида concurrent_queue_*, в котором имеется метод pop. Этот метод блокирует выполнение, пока в очереди не появится элемент. Метод push может блокировать поток только в классе concurrent_bounded_queue, в связи с чем этот контейнер предлагает также неблокирующий метод try_push.

Идея ограничения с целью уравнивания темпов производства и потребления, чтобы избежать переполнения памяти или чрезмерной загруженности ядер, присутствует также в потоковом графе (глава 3) в виде узла limiter_node.

Упорядочение по приоритету

Очередь с приоритетами поддерживает упорядоченность элементов на основе их приоритетов. Как уже было сказано, обычная очередь обслуживается в порядке «первым пришел – первым ушел», тогда как очередь с приоритетами сортирует элементы. Мы можем предоставить собственный функтор Compare для изменения порядка (по умолчанию подразумевается std::less<T>). Например, если указать std::greater<T>, то метод pop будет возвращать наименьший элемент в очереди. Именно так мы и поступили в коде на рис. 6.8.

Как сохранить потокобезопасность: забудьте о методах top, size, empty, front, back

Отметим, что метода top нет вообще, а методов size и empty лучше избегать. Из-за конкурентности значения, возвращаемые этими тремя методами, могут измениться еще до того, как мы успеем ими воспользоваться, поскольку другие потоки в это время могут выполнить операции push или pop. Кроме того, методы clear и swap, хотя и поддерживаются, потокобезопасными не являются. TBB заставляет нас переписать код, содержащий top, при переходе от std::priority_

queue tbb::concurrent_priority_queue, потому что возвращенный элемент мог бы стать недействительным в результате параллельно выполненной операции pop. TBB все же поддерживает методы size, empty и swap, т. к. возвращаемым им значениям конкурентность не угрожает. Однако мы рекомендуем использовать их с осторожностью, т. к. зависимость от них, скорее всего, означает, что код нужно переписать для работы в конкурентном окружении.

std:: code, not thread safe

```
#include <iostream>
#include <queue>



void main() {
  int sum (0);
  int item;

  std::priority_queue<int> myPQ;

  for(int i=0; i<10001; i+=1) {
    myPQ.push(i);
  }

  while( !myPQ.empty() ) {
    sum += myPQ.top();
    myPQ.pop();
  }

  // печатается "total: 50005000"
  std::cout << "total: "
            << sum << '\n';
}
```

tbb:: code, thread safe

```
#include <iostream>

#include <tbb/concurrent_priority_queue.h>
#include <tbb/parallel_for.h>

void main() {
  int sum (0);
  int item;

  tbb::concurrent_priority_queue<int> myPQ;

  tbb::parallel_for(0,10001,1,
    [&](size_t i){myPQ.push(i);} );


  while( myPQ.try_pop(item) )
    sum += item;



  // печатается "total: 50005000"
  std::cout << "total: "
            << sum << '\n';
}
```

Рис. 6.9 ❖ Причины использования try_pop вместо top и pop. Сравнивается применение очереди с приоритетами в STL и TBB. Обе версии печатают 50005000 при запуске без параллелизма, но версия TBB масштабируется и потокобезопасна

Итераторы

Для отладки (и только с этой целью) все три конкурентные очереди предлагают ограниченную поддержку итераторов (типы iterator и const_iterator). Она оставлена только для того, чтобы можно было проверить содержимое очереди во время отладки. Оба типа соблюдают обычные соглашения STL для однонаправленных итераторов. Порядок обхода – от добавленных давно к добавленным недавно. Любое изменение очереди делает недействительными все ссылающиеся на нее итераторы. Итераторы работают сравнительно медленно и предназначены только для отладки. Пример использования приведен на рис. 6.10.

```
#include <tbb/concurrent_queue.h>
#include <iostream>

int main() {
  tbb::concurrent_queue<int> queue;
  for( int i=0; i<10; ++i )
    queue.push(i);
  for( tbb::concurrent_queue<int>::const_iterator
       i(queue.unsafe_begin());
       i!=queue.unsafe_end();
       ++i )
    std::cout << *i << " ";
  std::cout << std::endl;
  return 0;
}
Output of this program is:
0 1 2 3 4 5 6 7 8 9
```

Рис. 6.10 ❖ Пример отладочного кода для обхода конкурентной очереди – обратите внимание на префикс `unsafe_` в методах `begin` и `end`, подчеркивающий их отладочную, потоконебезопасную природу

Зачем нужна конкурентная очередь: проблема A-B-A

В начале этой главы мы упомянули о великой ценности контейнеров, написанных специалистами по параллелизму, чтобы мы ими «просто пользовались». Никто из нас не желал бы заново изобретать хорошую масштабируемую реализацию для каждого приложения. В качестве обоснования этого тезиса мы немного отвлечемся и рассмотрим проблему A-B-A – классический пример неправильно реализованного параллелизма! На первый взгляд, конкурентная очередь – вещь достаточно простая, так что ее можно написать и самостоятельно. Отнюдь. Гораздо правильнее воспользоваться классом `concurrent_queue` из библиотеки TBB или еще какой-нибудь хорошо исследованной и хорошо реализованной конкурентной очередью. Смиренно признаемся, что мы на собственном опыте, как и многие другие до нас, убедились, что эта штука совсем не такая тривиальная, как мы наивно полагали. Идиома обновления (`compare_and_swap`), описанная в главе 5, не годится, если наши планы расстраивает проблема A-B-A (см. врезку). Эта проблема часто возникает при попытке спроектировать неблокирующий алгоритм для связанных структур данных, в т. ч. конкурентной очереди. Проектировщики TBB нашли решение проблемы A-B-A и уже включили его в конкурентные очереди. Нам остается лишь полагаться на их мудрость. Разумеется, код открыт, поэтому, если любопытно, можете заглянуть в него и посмотреть, как выглядит это решение. И если вы это сделаете, то увидите, что и при управлении ареной (тема главы 12) приходится иметь дело с проблемой A-B-A. Конечно, использовать TBB можно, и не зная всего этого. Мы только хотели подчеркнуть, что конкурентные структуры данных не так просты, как может показаться, – отсюда и любовь, которую мы испытываем к конкурентным структурам данных, поддерживаемым TBB.

Проблема A-B-A

Разобраться в проблеме A-B-A – ключ к тому, чтобы научиться продумывать последствия конкурентности при проектировании своих алгоритмов. TBB избегает проблемы A-B-A в реализации конкурентных очередей и других структур данных, но все равно это напоминание – «Мысли параллельно!».

Проблема A-B-A возникает, когда поток проверяет, верно ли, что в какой-то ячейке хранится значение *A*, и выполняет обновление, только если это так. Вопрос – произойдет ли ошибка, если другая задача изменит эту ячейку таким образом, что первая задача этого не заметит:

1. Задача читает *A* из *globalx*.
2. Другие задачи изменяют *globalx* с *A* на *B*, а затем обратно на *A*.
3. Первая задача выполняет `compare_and_swap`, читает *A* и, стало быть, не замечает, что *globalx* в течение некоторого времени было равно *B*.

Если задача ошибочно продолжает выполнение в предположении, что ячейка не изменялась с момента первого чтения, то она может повредить объект или еще как-то получить неверный результат.

Рассмотрим пример со связанным списком. Пусть имеется связанный список *W(1)→X(9)→Y(7)→Z(4)*, где буквами обозначены адреса узлов, а числами – значения в этих узлах. Предположим, что задача обходит список в поисках узла *X*, который нужно исключить. Задача читает указатель на следующий элемент *X.next* (который содержит *Y*), намереваясь записать его в *W.next*. Но перед тем как произвести обмен, задача на некоторое время приостанавливается.

Пока она ждет, другие задачи делают свое дело. Они исключают *X* из списка, а затем ухитряются повторно использовать ту же самую память, включить в список новую версию узла *X*, а также добавить узел *Q*. Теперь список имеет вид *W(1)→X(2)→Q(3)→Z(4)*.

Когда первая задача наконец просыпается, она обнаруживает, что *W.next* по-прежнему указывает на *X*, поэтому записывает *Y* в *W.next*, после чего связанный список превращается в мешанину.

Атомарные операции чего-то стоят, если обеспечивают достаточную защиту алгоритма. Если же проблема A-B-A может все испортить, то нужно искать более сложное решение. И в классе `tbb::concurrent_queue` присутствует эта дополнительная сложность, исправляющая положение!

Когда не надо использовать очереди: думайте об алгоритмах!

Очереди часто используются в параллельных программах в качестве буфера между производителями и потребителями. Но прежде чем прибегнуть к явной очереди, подумайте, нельзя ли применить алгоритм `parallel_do` или `pipeline` (см. главу 2). Они эффективнее очередей по следующим причинам:

- очередь принципиально является узким местом, потому что обязана поддерживать упорядоченность;
- поток, пытающийся извлечь значение из очереди и обнаруживший, что она пуста, будет простаивать до тех пор, пока кто-то не поместит в очередь значение;
- очередь – пассивная структура данных. Если некоторый поток поместит в нее значение, то до его извлечения может пройти некоторое время, а за это время само значение (и то, на что оно ссылается) может *остынуть*, т. е. будет вытеснено из кеша. Бывает еще хуже – другой поток извлекает значение, и его (а также то, на что оно ссылается) приходится перемещать на иное процессорное ядро.

С другой стороны, алгоритмы parallel_do и pipeline избегают этих узких мест. Поскольку использование потоков в них неявное, они оптимизируют свои рабочие потоки, так чтобы те занимались другой работой, пока в очереди не появится значение. Кроме того, они стараются не дать *остыть* данным в кеше. Например, когда на вход parallel_do подается новая работа, она остается локальной в добавившем ее потоке, если только другой простаивающий поток не позаимствует ее раньше, чем *горячий* поток успеет обработать. Таким образом, данные чаще обрабатываются *горячим* потоком, что сокращает задержки на выборку данных из памяти.

Конкурентный вектор

TBB предлагает класс concurrent_vector. Тип concurrent_vector<T> – это динамически растущий массив элементов типа T. Увеличение размера concurrent_vector безопасно, даже когда другие потоки работают с его элементами или тоже пытаются увеличить размер. Для обеспечения безопасного роста в условиях конкурентности класс concurrent_vector предоставляет три метода, поддерживающих типичное использование динамических массивов: push_back, grow_by и grow_to_at_least.

На рис. 6.11 показано простое применение concurrent_vector, а на рис. 6.12 – результат конкурентного добавления элементов параллельно работающими потоками (в виде распечатки содержимого вектора). Результаты обоих прогонов будут одинаковы, если отсортировать их по значению.

Когда использовать tbb::concurrent_vector вместо std::vector

Главная ценность concurrent_vector<T> – возможность роста вектора при конкурентном доступе и гарантия того, что элементы не будут перемещены в памяти.

Накладные расходы concurrent_vector выше, чем у std::vector. Поэтому использовать concurrent_vector следует только тогда, когда действительно есть необходимость динамического роста при наличии конкурентных операций доступа или требуется, чтобы элементы никогда не перемещались.

Элементы никогда не перемещаются

Класс concurrent_vector никогда не перемещает элементы в другое место, если только массив не очищается. Это можно рассматривать как преимущество по сравнению с классом std::vector из STL даже в однопоточном случае. Контейнер выделяет память в виде последовательности непрерывных массивов. При первом выделении памяти, в процессе увеличения размера или операции присваивания определяется размер первого массива. Если начальный размер мал, то будет иметь место фрагментация строк кеша, что может увеличить время доступа к элементу. Метод shrink_to_fit() объединяет несколько небольших массивов в один непрерывный, что может уменьшить время доступа.

```
#include <iostream>

#include <tbb/concurrent_vector.h>
#include <tbb/parallel_for.h

void oneway() {
//  Создаем вектор целых чисел
    tbb::concurrent_vector<int> v = {3, 14, 15, 92};

    // Добавляем в вектор целые ПОСЛЕДОВАТЕЛЬНО
    for( int i = 100; i < 1000; ++i ) {
      v.push_back(i*100+11);
      v.push_back(i*100+22);
      v.push_back(i*100+33);
      v.push_back(i*100+44);
    }

    // Обходим и распечатываем вектор (только для отладки)
    for(int n : v) {
      std::cout << n << std::endl;
    }
}

void allways() {
// Создаем вектор целых чисел
    tbb::concurrent_vector<int> v = {3, 14, 15, 92};

    // Добавляем в вектор целые ПАРАЛЛЕЛЬНО
    tbb::parallel_for( 100, 999, [&](int i){
      v.push_back(i*100+11);
      v.push_back(i*100+22);
      v.push_back(i*100+33);
      v.push_back(i*100+44);
      });

    // Обходим и распечатываем вектор (только для отладки)
    for(int n : v) {
      std::cout << n << std::endl;
    }
}
```

Рис. 6.11 ❖ Простой пример конкурентного вектора

```
3                    3
14                   14
15                   15
92                   92
10011                10011
. . .                . . .
84911                72611
84922                91211
84933                87111
84944                72622
85011                91222
85022                87122
85033                72633
85044                91233
. . .                . . .
99933                99833
99944                99844
```

Рис. 6.12 ❖ В левой части показан результат при использовании цикла `for` (последовательного), в правой – при использовании `parallel_for` (конкурентное добавление в вектор)

Конкурентное увеличение размера concurrent_vector

Хотя конкурентное увеличение размера принципиально несовместимо с идеальной безопасностью относительно исключений, `concurrent_vector` все же предлагает практически полезный уровень такой безопасности. В типе элемента должен быть определен деструктор, не возбуждающий исключений, а если конструктор может возбуждать исключения, то деструктор обязан быть невиртуальным и корректно работать с памятью, заполненной нулями.

Метод `push_back(x)` безопасно добавляет `x` в конец вектора. Метод `grow_by(n)` безопасно добавляет в конец вектора `n` смежных в памяти элементов. Оба метода возвращают итератор, указывающий на последний добавленный элемент. Все новые элементы инициализируются значением `T()`. Следующая функция безопасно добавляет C-строку в конец разделяемого вектора:

```
void Append( concurrent_vector<char>& vector,
             const char* string ) {
    size_t n = strlen(string)+1;
    std::copy( string, string+n, vector.grow_by(n) );
}
```

Метод `grow_to_at_least(n)` увеличивает вектор до размера `n`, если он короче. Конкурентные обращения к методам, увеличивающим размер, необязательно возвращают управление в том же порядке, в каком добавлялись элементы.

Метод `size()` возвращает количество элементов в векторе. Оно может учитывать элементы, которые еще не до конца сконструированы после вызова методов `push_back`, `grow_by` или `grow_to_at_least`. В предыдущем примере используют-

ся std::copy и итераторы, а не strcpy и указатели, потому что соседние элементы concurrent_vector необязательно расположены по соседним адресам. Использовать итераторы во время увеличения размера concurrent_vector безопасно при условии, что итератор не выходит за границу, определяемую текущим значение end(). Однако итератор может ссылаться на элемент, конструирование которого еще не завершено. Поэтому от нас требуется синхронизировать конструирование и доступ.

Операции над объектами типа concurrent_vector потокобезопасны относительно роста вектора, но не его очистки или уничтожения. Никогда не вызывайте clear(), если имеются незавершенные операции над concurrent_vector.

Резюме

В этой главе мы обсудили три ключевые структуры данных (хеш/отображение/множество, очереди и вектор), поддерживаемые TBB. Эта поддержка обеспечивает потокобезопасность (возможность конкурентного доступа), а также масштабируемую реализацию. Мы посоветовали, каких вещей следует избегать, поскольку они могут стать причиной неприятностей в параллельных программах, – это, в частности, относится к использованию итераторов, возвращаемых методами отображений и множеств, для чего-либо, кроме ссылки на найденный элемент. Мы обсудили проблему A-B-A и объяснили, почему она является веской причиной для использования TBB вместо написания собственных алгоритмов и отличным примером того, о чем нужно думать при создании параллельных программ с разделяемыми данными.

Как обычно, полное описание API приведено в приложении B, а весь код, приведенный на рисунках, можно скачать.

Несмотря на отличную поддержку параллельного использования контейнеров, мы не устаем повторять, что тщательное обдумывание алгоритмов с целью сведения к минимуму любых видов синхронизации критически важно для высокопроизводительного параллельного программирования. Если вы сможете избежать разделения структур данных, используя алгоритмы parallel_do, pipeline, parallel_reduce и другие, о чем было сказано в разделе «Когда не надо использовать очереди: думайте об алгоритмах!», то увидите, что программа масштабируется лучше. Мы возвращаемся к этой мысли много раз и по разным поводам, поскольку ее усвоение крайне важно для эффективного параллельного программирования.

Глава 7

Масштабируемое выделение памяти

В этой главе обсуждается *критическая* часть любой параллельной программы: масштабируемое выделение памяти, в т. ч. оператором `new`, а также с помощью явных обращений к функциям `malloc`, `calloc` и т. д. Масштабируемое выделение памяти можно использовать вне зависимости от того, пользуемся мы другими частями библиотеки TBB или нет. Помимо интерфейсов, предназначенных для прямого применения, TBB предлагает метод «прокси» для автоматической замены функций динамического выделения памяти в C/C++ – простой, эффективный и популярный способ повысить производительность, не внося никаких изменений в код. Он работает при любом уровне «современности» кода на C++ – и когда вы пользуетесь современной и рекомендуемой функцией `std::make_shared`, и когда работаете с `new` и `malloc`, на которые теперь поглядывают косо. Выигрыш в производительности, который дает масштабируемый распределитель памяти, значителен, потому что он решает именно те проблемы, которые способны ограничить производительность и несут риск ложного разделения. Масштабируемый распределитель памяти, входящий в состав TBB, стал одним из первых нашедших широкое применение, в немалой степени потому, что он поставляется вместе с TBB бесплатно и подчеркивает важность учета вопросов выделения памяти в любой параллельной программе. И до сих пор он остается чрезвычайно популярным и одним из лучших имеющихся масштабируемых распределителей.

В современном программировании на C++ (которое отдает предпочтение интеллектуальным указателям), дополненном параллельным мышлением, рекомендуется использовать масштабируемые распределители памяти либо явно с помощью `std::allocate_shared`, либо неявно с помощью `std::make_shared`.

Выделение памяти в современном C++

Производительность представляет особый интерес для параллельного программирования, но *корректность* критически важна во *всех* приложениях. Выделение и освобождение памяти – известные источники ошибок в при-

ложениях, и это послужило причиной для многих добавлений в стандарт C++ и постепенного сдвига в сторону того, что принято называть современным программированием на C++!

В современном программировании на C++ *рекомендуется* использовать управляемое выделение памяти с помощью интеллектуальных указателей, вошедших в C++11 (`make_shared`, `allocate_shared` и т. д.), и *не рекомендуется* повсюду применять `malloc` или `new`. Примеры использования `std::make_shared` встречались нам с самой первой главы. Добавление функции `std::aligned_alloc` в C++17 решает вопрос о выравнивании на границу кеша с целью избежать ложного разделения, но не затрагивает проблему масштабируемого выделения памяти. Сейчас в работе находится много дополнений в C++20, но явной поддержки масштабируемости среди них нет.

TBB по-прежнему предлагает параллельным программистам этот критически важный инструмент: *масштабируемое выделение памяти*. Решение идеально сочетается со всеми версиями стандартов C++ и C. Основную идею поддержки в TBB можно описать как *организацию пула памяти потоками*. Пул предотвращает падение производительности, которое вызвано стратегией выделения памяти, не стремящейся избежать ненужного переноса данных между кешами. TBB предлагает также масштабируемое выделение памяти в сочетании с выравниванием на границу кеша, не ограничивающееся теми выгодами, которые сулит простое использование `std::aligned_alloc`. Выравнивание на границу кеша не производится по умолчанию, потому что безрассудное применение могло бы резко увеличить потребление памяти.

Как мы увидим в этой главе, масштабируемое выделение памяти может быть критическим фактором производительности. При вызове `std::make_shared` распределитель не указывается, но функция `std::allocate_shared` позволяет его задать.

Эта глава посвящена масштабируемым распределителям памяти, которые следует использовать, каким бы способом ни выделялась память в приложении на C++. Современное программирование на C++, дополненное параллельным мышлением, поощряет явное использование `std::allocate_shared` в комбинации с масштабируемыми распределителями памяти TBB или неявное использование `std::make_shared` совместно с TBB путем переопределения подразумеваемого по умолчанию оператора `new`. Отметим, что на функции `std::make_shared` определение оператора `new` в конкретном классе не отражается, потому что она в действительности выделяет больший блок памяти, включающий и содержимое класса, и дополнительное место для служебных целей (в частности, атомарной переменной, которая необходима интеллектуальному указателю). Поэтому переопределения оператора `new`, подразумеваемого по умолчанию (с целью использовать распределитель TBB), достаточно, чтобы повлиять на поведение `std::make_shared`.

Масштабируемое выделение памяти: что

В этой главе обсуждаются четыре категории средств масштабируемого выделения памяти, перечисленные на рис. 7.1. Средства из всех категорий можно

использовать в любом сочетании, на категории они разбиты только для удобства объяснения функциональности. Библиотека прокси для C/C++ – наиболее популярный способ использования масштабируемого распределителя памяти.

Способ использования	Краткое описание	Номер рисунка, где перечислены интерфейсы
Прокси C/C++	Наиболее популярный способ. Автоматически подменяются стандартные методы выделения памяти. Вносить изменения в код не нужно	На рис. 7.4 приведен список функций, подмененных библиотекой прокси
Функции C++	Стандартные интерфейсы C++ (`std::allocator`)	Список функций на рис. 7.12
Классы C++	Стандартные интерфейсы C++ (`std::allocator`)	Список классов на рис. 7.14
Приемы оптимизации производительности	Способы улучшить производительность (при любом способе использования) для удовлетворения конкретных потребностей, включая использование больших страниц. Полезно, когда нужно выжать все до последней капли	Функциональные интерфейсы и переменная среды перечислены на рис. 7.18

Рис. 7.1 ❖ Способы использования масштабируемого распределителя памяти

Масштабируемый распределитель памяти отделен от остальных частей TBB, так что мы можем включить в программу только его, не обременяя код параллельными алгоритмами и шаблонами контейнеров.

Масштабируемое выделение памяти: почему

Большая часть этой книги посвящена вопросам повышения скорости программ за счет распараллеливания, но потоконебезопасное выделение и освобождение памяти может свести на нет всю проделанную в муках работу! Есть два фактора, из-за которых так важно тщательно выбирать стратегию выделения памяти в параллельной программе: состязание за распределитель и эффекты кеша.

Если используется обыкновенный, не рассчитанный на многопоточность распределитель, то выделение памяти может стать узким местом в многопоточной программе, потому что все потоки состязаются за глобальную блокировку при каждом выделении и освобождении памяти в единственной глобальной куче. Работающие таким образом программы не масштабируются. На самом деле из-за этого состязания программы, в которых память выделяется очень интенсивно, могут даже работать медленнее при увеличении числа процессорных ядер! Масштабируемые распределители памяти решают эту проблему благодаря использованию более сложных структур данных, чтобы избежать состязания.

Второй вопрос – эффекты кеширования – возникает, потому что в основе работы с памятью лежит аппаратный механизм кеширования данных. Поэтому от характера использования данных в программе зависит, где будут кешироваться данные. Если мы выделяем память для потока B и распределитель

выдает нам память, недавно освобожденную потоком A, то велика вероятность, что это повлечет за собой копирование из одного кеша в другой, а это снизит производительность без всякой на то необходимости. Кроме того, если блоки памяти, выделенные разными потоками, расположены слишком близко друг к другу, то они могут оказаться в одной строке кеша. Это разделение может быть *истинным* (разделяется один и тот же объект) или *ложным* (никакие объекты не разделяются, но они случайно попали в одну строку кеша). Любой вид разделения может оказать сильное негативное влияние на производительность, но *ложное разделение* особенно интересно, потому что его можно избежать, если никакого разделения не планировалось. Масштабируемые распределители памяти избегают ложного разделения с помощью класса `cache_aligned_allocator<T>`, который располагает любой блок памяти в начале строки кеша и поддерживает отдельные кучи для каждого потока, перебалансируя их по мере необходимости. Такая организация позволяет заодно решить вышеупомянутую проблему состязания.

Масштабируемый распределитель памяти вполне способен повысить производительность на 20–30 %, и мы даже слышали о 4-кратном увеличении в экстремальных случаях – просто после перекомпоновки программы с масштабируемым распределителем.

Избежание ложного разделения с помощью дополнения

Дополнение необходимо, если содержимое структуры данных таково, что возможны проблемы из-за ложного разделения. В главе 5 рассматривалась задача о вычислении гистограммы. Как интервалы гистограммы, так и мьютексы для защиты этих интервалов – примеры структур данных, которые упакованы в памяти настолько плотно, что в одной строке кеша могут находиться структуры, обновляемые разными потоками.

Идея дополнения проста – вставить между элементами данных достаточно длинные промежутки, чтобы соседние элементы в одной строке кеша не обновлялись разными задачами.

Если говорить о ложном разделении, то самая первая мера противодействия – использовать `tbb::cache_aligned_allocator` вместо `std::allocator` или `malloc` при объявлении разделяемой гистограммы (см. рис. 5.20), как показано на рис. 7.2.

```
std::vector<tbb::atomic<int>,
tbb::cache_aligned_allocator<tbb::atomic<int>>> hist_p(num_bins);
```

Рис. 7.2 ❖ Простой вектор гистограммы, содержащий атомарные переменные

Однако это всего лишь выравнивание начала вектора гистограммы, так чтобы элемент `hist_p[0]` всегда оказывался в начале строки кеша. Это означает, что `hist_p[0]`, `hist_p[1]`, … , `hist_p[15]` хранятся в одной и той же строке кеша, и мы получаем ложное разделение, если один поток увеличивает `hist_p[0]`, а другой – `hist_p[15]`. Чтобы разрешить эту проблему, нужно, чтобы каждый элемент

гистограммы, каждый интервал занимал полную строку кеша, и достигается это с помощью стратегии дополнения, показанной на рис. 7.3.

```
struct bin {    // sizeof(bin) = 64 байта (длина строки кеша)
  tbb::atomic<int> count;  // 4 байта
  uint32_t padding[15];    // 60 байт
};
std::vector<bin, tbb::cache_aligned_allocator<bin>> hist_p(num_bins);
for (size_t i = r.begin(); i < r.end(); ++i)
   hist_p[image[i]].count++;
```

Рис. 7.3 ❖ Предотвращение ложного разделения с помощью дополнения вектора гистограммы

Как видно на рис. 7.3, массив интервалов `hist_p` теперь является вектором структур `struct`, каждая из которых содержит одну атомарную переменную и фиктивный массив длиной 60 байт, чтобы заполнить всю строку кеша. Таким образом, этот код зависит от архитектуры. В современных процессорах Intel длина строки кеша равна 64 байтам, но можно встретить защищенные от ложного разделения реализации, предполагающие длину 128 байт. Объясняется это распространенностью техники предвыборки (кеширование также строки i+1, если запрашивается строка i), которая в некотором смысле эквивалентна строкам кеша длиной 128 байт.

Наша защищенная от ложного разделения структура данных занимает в 16 раз больше места, чем исходная. Это пример компромисса между временем и памятью, который часто возникает в программировании: на этот раз мы потребляем больше памяти, но программа работает быстрее. Вот еще несколько примеров: меньший размер кода или разворачивание циклов, вызов или встраивание функций, обработка сжатых и несжатых данных.

Минуточку! А не слишком ли примитивна реализация структуры `bin`? Да, конечно! Вот решение, в котором меньше зашитых в код констант:

```
struct bin {    // sizeof(bin) = 64 байта (длина строки кеша)
  tbb::atomic<int> count;            // 4 байта
  uint8_t padding[64-sizeof(count)]; // 60 байт
};
```

Поскольку `sizeof()` вычисляется на этапе компиляции, мы можем использовать точно такую же конструкцию и для других дополненных структур данных, в которых рабочая нагрузка (в данном случае – счетчик) имеет другой размер. Но в стандарте C++ имеется решение получше:

```
struct bin {    // sizeof(bin) = 64 байта (длина строки кеша)
  alignas(64) tbb::atomic<int> count;
};
std::vector<bin, tbb::cache_aligned_allocator<bin>>
    hist_p(num_bins);
```

Этот код гарантирует, что каждый интервал в векторе `hist_p` занимает полную строку кеша – благодаря методу `alignas()`. И еще одно. Мы ведь любим пи-

сать переносимый код, правда? А что, если в другой или в будущей архитектуре длина строки кеша изменится? Не проблема, в стандарте C++17 есть решение и на этот случай:

```
struct bin {  // sizeof(bin) = длина строки кеша (зависит от реализации)
  // доступно начиная с C++17
  alignas(std::hardware_destructive_interference_size)
     tbb::atomic<int> count;
};
```

Отлично, можно считать, что проблему *ложного разделения* мы решили. А как насчет истинного?

Два разных потока рано или поздно начнут увеличивать один и тот же интервал, что приведет к перебрасыванию из одного кеша в другой. Чтобы решить эту проблему, нужна новая идея! В главе 5 мы показали, как это сделать с помощью *приватизации и редукции*.

Альтернативы масштабируемому выделению памяти: какие

В наши дни библиотека TBB – не единственный источник средств для масштабируемого выделения памяти. Хотя мы ее любим, в этом разделе познакомимся с другими популярными вариантами. Важно понимать, что при программировании с помощью TBB можно использовать любой масштабируемый распределитель памяти – входящий в состав TBB или иной.

TBB первой предложила ставший популярным метод параллельного программирования, в котором масштабируемое выделение памяти продвигалось наравне с другими приемами, поскольку создатели TBB понимали важность вопросов выделения памяти в параллельной программе. Распределитель памяти из TBB остается чрезвычайно популярным и сегодня и, без сомнения, является одним из лучших в этом классе продуктов.

Масштабируемый распределитель памяти TBB можно использовать отдельно от всех прочих частей библиотеки. И наоборот, TBB может работать с любым масштабируемым распределителем памяти.

Наиболее популярными альтернативами распределителю TBB являются `jemalloc` и `tcmalloc`. Как и распределитель TBB, они составляют альтернативу `malloc` и ставят на первое место уменьшение фрагментации и масштабируемую поддержку конкурентности. Все три распространяются в открытом виде по либеральной лицензии (BSD или Apache).

Кто-то обязательно расскажет вам, что сравнивал `tbbmalloc` с `tcmalloc` и `jemalloc` в своем приложении и нашел, что первая лучше. Это обычное дело. Но есть люди, отдающие предпочтение `jemalloc`, `tcmalloc` или `llalloc`, хотя активно пользуются другими частями TBB. Это тоже нормально. Выбор за вами.

`jemalloc` – распределитель памяти из библиотеки `libc`, входящей в состав FreeBSD. Недавно в него были добавлены такие средства разработки, как профилирование кучи и развитая система точек подключения для мониторинга и настройки. Этот распределитель используется в компании Facebook.

tcmalloc – часть комплекта инструментов gperftools от Google, включающего наряду с ним другие средства анализа производительности. Этот распределитель используется в Google.

llalloc – свободный, распространяемый в открытом виде безблокировочный распределитель памяти от компании Lockless Inc. Для использования в ПО с закрытым исходным кодом его можно купить.

Поскольку разные приложения ведут себя по-разному, особенно в части паттернов выделения и освобождения памяти, выбрать единственное решение, подходящее для всех случаев, невозможно. Мы уверены, что любой выбор, будь то tbbmalloc, jemalloc или tcmalloc, окажется намного лучше подразумеваемой по умолчанию функции malloc или оператора new, если те не рассчитаны на масштабируемость (в FreeBSD jemalloc используется в качестве malloc по умолчанию).

К вопросу о компиляции

Если для компиляции программы используется компилятор Intel или gcc, то рекомендуется задавать следующие флаги:

- -fno-builtin-malloc (в Windows: /Qfno-builtin-malloc);
- -fno-builtin-calloc (в Windows: /Qfno-builtin-calloc);
- -fno-builtin-realloc (в Windows: /Qfno-builtin-realloc);
- -fno-builtin-free (в Windows: /Qfno-builtin-free).

Это связано с тем, что компилятор может производить некоторые оптимизации в предположении, что используются встроенные в него же функции. Это предположение может оказаться неверным, если используются другие распределители памяти. Возможно, если опустить эти флаги, ничего страшного не случится, но лучше подстраховаться. Рекомендуем почитать, что сказано по этому поводу в документации по компилятору.

Самый популярный способ использования (библиотека прокси для C/C++): как

С помощью прокси-методов мы можем глобально подменить new и delete и функции malloc/calloc/realloc/free и т. д. Такой способ замены malloc и прочих функций C/C++ для динамического выделения памяти чаще всего применяется, чтобы воспользоваться масштабируемым распределителем памяти TBB. Он весьма эффективен.

Заменить malloc/calloc/realloc/free и т. д. (полный перечень см. на рис. 7.4) и new/delete позволяет библиотека tbbmalloc_proxy. Этот метод прост и годится для большинства программ. Детали немного разнятся в зависимости от операционной системы, но результат всегда одинаков. Имена библиотек приведены на рис. 7.5, а сводка методов – на рис. 7.6.

	Linux	macOS	Windows
Функции из стандартной библиотеки C: `malloc`, `calloc`, `realloc`, `free`	да	да	да
Функция из стандартной библиотеки C (добавлена в C11): `aligned_alloc`	да		
Функция из стандарта POSIX: `posix_memalign`	да	да	
В зависимости от платформы могут быть подменены и другие функции (ниже приведен актуальный на момент публикации перечень, добавления и изменения отражены в руководстве разработчика по TBB)			
Функции из библиотеки GNU C (glibc)	да		
Функции из библиотеки C времени выполнения от Microsoft			да
`valloc`			
`malloc_size`		да	
`memalign`, `pvalloc`, `mallopt`	да		

Рис. 7.4 ❖ Список функций, подменяемых библиотекой прокси

	Выпускная версия библиотеки	Отладочная версия библиотеки
Linux	`libtblmalloc_proxy.so.2`	`libtblmalloc_proxy_debug.so.2`
macOS	`libtblmalloc_proxy.dylib`	`libtblmalloc_proxy_debug.dylib`
Windows	`tbbmalloc_proxy.dll`	`tbbmalloc_proxy_debug.dll`

Рис. 7.5 ❖ Имена библиотек прокси

	Внедрение прокси
Linux	`LD_PRELOAD` (рис. 7.7)
macOS	`DYLD_INSERT_LIBRARIES` (рис. 7.7)
Windows	См. рис. 7.8

Рис. 7.6 ❖ Способы использования библиотеки прокси

Linux: использование библиотеки прокси

В Linux подмену можно совершить, либо загрузив библиотеку прокси на этапе загрузки программы с помощью переменной среды `LD_PRELOAD` (не изменяя исполняемый файл, как показано на рис. 7.7), либо путем компоновки программы с библиотекой прокси (флаг `-ltbbmalloc_proxy`). Загрузчик программ в Linux должен иметь возможность найти библиотеку прокси и библиотеку с масштабируемым распределителем памяти на этапе загрузки. Для этого мы можем включить каталог, содержащий эти библиотеки, в перечень путей, прописанный в переменной среды `LD_LIBRARY_PATH`, или добавить его в файл `/etc/ld.so.conf`. Существует два ограничения на подмену функций работы с динамической памятью: (1) точки подключения glibc, например `__malloc_hook`, не поддерживаются; (2) Mono (реализация Microsoft .NET Framework с открытым исходным кодом) не поддерживается.

macOS: использование библиотеки прокси

В macOS подмену можно совершить, либо загрузив библиотеку прокси на этапе загрузки программы с помощью переменной среды DYLD_INSERT_LIBRARIES (не изменяя исполняемый файл, как показано на рис. 7.7), либо путем компоновки программы с библиотекой прокси (флаг -ltbbmalloc_proxy). Загрузчик программ в macOS должен иметь возможность найти библиотеку прокси и библиотеку с масштабируемым распределителем памяти на этапе загрузки. Для этого мы можем включить каталог, содержащий эти библиотеки, в перечень путей, прописанный в переменной среды DYLD_LIBRARY_PATH.

```
# В Linux:
export LD_PRELOAD=libtblmalloc_proxy.so.2

# В macOS:
export DYLD_INSERT_LIBRARIES=$TBBROOT/lib/libtblmalloc_proxy.dylib

# В Windows не существует простого метода "внедрения DLL", хотя в "Википедии"
# имеется статья "DLL Injection", в которой обсуждаются более сложные способы.
# Мы вместо этого рекомендуем методы, описанные на рис. 7.8.
```

Рис. 7.7 ❖ Переменные среды для внедрения масштабируемого распределителя памяти TBB

Детали реализации для любопытных (можно не читать): в TBB применяется хитроумный способ обхода ограничения, согласно которому для использования DYLD_INSERT_LIBRARIES требуются плоские пространства имен, чтобы можно было получить доступ к символам разделяемой библиотеки. Обычно для приложений с двухуровневыми пространствами имен этот метод не работает, а принудительное использование плоских пространств имен, скорее всего, привело бы к краху. TBB решает проблему следующим образом: когда библиотека libtbbmalloc_proxy загружается в процесс, вызывается ее статический конструктор и регистрирует *зону malloc* для функций выделения памяти из TBB. Это позволяет переадресовать вызовы функций работы с памятью из стандартной библиотеки на функции из TBB. А значит, приложению вообще не нужно использовать символы из библиотеки TBB – оно продолжает вызывать стандартные библиотечные функции. Поэтому проблемы с пространствами имен не возникает. Механизм *зон malloc* в macOS также позволяет приложению работать сразу с несколькими распределителями памяти (например, если таковые используются в разных библиотеках). Тем самым гарантируется, что Intel TBB будет использовать один и тот же распределитель для выделения и освобождения памяти. Это защитная мера против краха вследствие вызова функции освобождения одного распределителя для памяти, выделенной другим распределителем.

Windows: использование библиотеки прокси

В Windows придется модифицировать исполняемый файл. Чтобы принудительно загрузить библиотеку прокси, мы можем либо вставить директиву #include в свой код, либо воспользоваться флагами компоновщика, как показано на рис. 7.8. Загрузчик программ в Windows должен иметь возможность найти

библиотеку прокси и библиотеку с масштабируемым распределителем памяти на этапе загрузки. Для этого мы можем включить каталог, содержащий эти библиотеки, в перечень путей, прописанный в переменной среды PATH.

Включение tbbmalloc_proxy.h в исходный код любого двоичного файла (загружаемого на этапе запуска приложения):

```
#include <tbb/tbbmalloc_proxy.h>
```

или добавление следующих флагов компоновщика для двоичного файла. Флаги можно задавать как для EXE-файла, так и для DLL, загружаемой при запуске приложения:

Для win32:
```
tbbmalloc_proxy.lib /INCLUDE:"___TBB_malloc_proxy"
```
Для win64:
```
tbbmalloc_proxy.lib /INCLUDE:"__TBB_malloc_proxy"
```

Рис. 7.8 ❖ Способы использования библиотеки прокси в Windows (примечание: в win32 на один знак подчеркивания больше, чем в win64)

Тестирование библиотеки прокси

Чтобы быстро проверить, действительно ли после наших действий выделение памяти стало работать быстрее, можно воспользоваться тестовой программой, показанной на рис. 7.9, на многоядерной машине. На рис. 7.10 показаны результаты хронометража на 4-ядерной виртуальной машине под управлением Ubuntu Linux. На рис. 7.11 показаны результаты хронометража на 4-ядерном iMac. В Windows профилировщик производительности в Visual Studio на 4-ядерном Intel NUC (Core i7) показывает 94 мс без масштабируемого распределителя памяти и 50 мс с ним (после добавления #include <tbb/tbbmalloc_proxy.h> в tbb_mem.cpp). Таким образом, мы убеждаемся, что внедрение масштабируемого распределителя памяти действительно работает (для new/delete) и дает нетривиальный выигрыш! Тест пары malloc()/free() дает аналогичные результаты. Мы включили его в виде файла tbb_malloc.cpp в состав загружаемого кода к этой главе.

```
#include <cstdio>
#include <tbb/tbb.h>

const int N = 1000000;
double *a[N];

int main() {
  tbb::parallel_for( 0, N-1, [&](int i) { a[i] = new double; } );
  tbb::parallel_for( 0, N-1, [&](int i) { delete a[i];       } );
  return 0;
}
```

Рис. 7.9 ❖ Небольшая тестовая программа (tbb_mem.cpp) для проверки быстродействия new/delete

Этим программам нужно очень много места в стеке, потому выполните команду `ulimit -s unlimited` (Linux/macOS) или задайте параметр `/STACK:10000000` (**Visual Studio: Свойства ➤ Свойства конфигурации ➤ Компоновщик ➤ Система ➤ Резервируемый размер стека**). В противном случае аварийное завершение неизбежно.

```
% g++ -o tbb_mem tbb_mem.cpp -std=c++11 -ltbb -O3
% time ./tbb_mem

real    0m0.160s
user    0m0.072s
sys     0m0.048s
%
% export LD_PRELOAD=$TBBROOT/lib/libtbbmalloc_proxy.so.2
or alternativelly
% g++ -o tbb_mem tbb_mem.cpp -std=c++11 -ltbb -O3 -ltbbmalloc_proxy
% time ./tbb_mem

real    0m0.043s
user    0m0.048s
sys     0m0.028s
```

Рис. 7.10 ❖ Запуск и результаты хронометража `tbb_mem.cpp` на 4-ядерной виртуальной Linux-машине

```
% time ./tbb_mem

real    0m0.046s
user    0m0.078s
sys     0m0.053s
%
% export DYLD_INSERT_LIBRARIES=$TBBROOT/lib/libtbbmalloc_proxy.dylib
%
% time ./tbb_mem

real    0m0.019s
user    0m0.032s
sys     0m0.009s
```

Рис. 7.11 ❖ Запуск и результаты хронометража `tbb_mem.cpp` на 4-ядерном iMac (macOS)

Функции C: масштабируемые распределители памяти для C

Набор функций на рис. 7.12 составляет интерфейс из языка C к масштабируемому распределителю памяти. Поскольку TBB предполагает программирование на C++, эти интерфейсы предназначены не для пользователей TBB, а для использования из кода, написанного на C.

Семейство	`void *scalable_malloc(size_t size)`	аналог `malloc`
	`void scalable_free(void *ptr)`	аналог `free`
	`void *scalable_realloc(void *ptr, size_t size)`	аналог `realloc`
	`void *scalable_calloc(size_t nobj, size_t size)`	аналог `realloc`, дополняющий `scalable_malloc`
	`void *scalable_posix_memalign(void **memptr, size_t alignment, size_t size )`	аналог `posix_memalign`
Семейство	`void* scalable_aligned_malloc( size_t size, size_t alignment)`	аналог `posix_memalign`
	`void* scalable_aligned_realloc(void* ptr, size_t size, size_t alignment)`	аналог `realloc`, дополняющий `scalable_aligned_malloc`
	`void scalable_aligned_free( void* ptr )`	аналог `free` для памяти, выделенной с помощью `scalable_aligned_malloc` или `scalable_aligned_realloc`
Все семейства	`size_t scalable_msize( void* ptr )`	аналог `msize`, `malloc_msize`, `malloc_usable_size`. Возвращает размер доступной для использования части блока, выделенного `scalable_x`, или 0, если `ptr` не указывает на такой блок
	`int scalable_allocation_mode(int param, intptr_t value)`	установить режим выделения памяти. *См. раздел «Настройка производительности: некоторые рычаги управления» ниже в этой главе*
	`int scalable_allocation_command(int cmd, void *param)`	вызов команд распределителя TBB. *См. раздел «Настройка производительности: некоторые рычаги управления» ниже в этой главе*

Рис. 7.12 ❖ Функции, предлагаемые масштабируемым распределителем памяти TBB

Все функции выделения вида `scalable_x` ведут себя аналогично библиотечной функции `x`. Эти функции разбиты на два семейства, как показано на рис. 7.13. Память, выделенная функцией `scalable_x` из одного семейства, должна освобождаться или перераспределяться функцией из того же семейства, а не стандартной библиотечной функцией C. Аналогично память, выделенная стандартной библиотечной функцией C или оператором C++ `new`, не должна освобождаться или перераспределяться функцией `scalable_x`.

Эти функции объявлены в заголовке `#include <tbb/scalable_allocator.h>`.

Семейство	Функция выделения	Функция освобождения	Библиотека с аналогами

1	`scalable_malloc`
	`scalable_calloc`
	`scalable_realloc`
		`scalable_free`	Стандартная библиотека C

	`scalable_posix_memalign`		POSIX
2	`scalable_aligned_malloc` `scalable_aligned_realloc`	`scalable_aligned_free`	Исполняющая среда Microsoft C

Рис. 7.13 ❖ Распределение функций выделения и освобождения памяти по семействам

Классы C++: масштабируемые распределители памяти для C++

Хотя библиотека прокси дает всеобъемлющее решение, позволяющее включить масштабируемый распределитель памяти в собственные программы, оно основано на конкретных средствах, которые можно использовать и напрямую.

TBB предлагает три вида классов C++ для выделения памяти: (1) распределители с сигнатурами, необходимыми для подстановки вместо `std::allocator<T>` в классы из STL; (2) поддержка пула памяти для STL-контейнеров; (3) специальный распределитель для выровненных массивов.

Распределители с сигнатурой std::allocator<T>

Набор классов на рис. 7.14 предлагает интерфейс к масштабируемому распределителю памяти из программы на C++. В TBB имеется четыре шаблонных класса (`tbb_allocator`, `cached_aligned_allocator`, `zero_allocator` и `scalable_allocator`) с такими же сигнатурами, как у класса `std::allocator<T>` по стандарту C++. Это означает, что помимо `<T>` поддерживается `<void>`, как требуется в C++11 и предшествующих стандартах, но объявлено нерекомендуемым в C++17 и, скорее всего, будет исключено в C++20. Поэтому любой из этих распределителей можно передать в качестве функции выделения STL-шаблонам, например `vector`. Все четыре класса моделируют концепцию Allocator, т. е. удовлетворяют всем «требованиям к распределителям» в C++, с дополнительными гарантиями, сформулированными в стандарте для работы с контейнерами ISO C++.

`tbb::aligned_space< T, N >`	Выровненный блок памяти достаточно большой для конструирования массива `N` элементов типа `T`. Этот класс не конструирует и не уничтожает сами элементы. Нечто аналогичное, но семантически несовместимое предлагает класс C++ `aligned_storage`
`tbb::cache_aligned_allocator< T >`	Масштабируемое выделение памяти, выровненной на начало строки кеша. Помогает избежать ложного разделения, но ценой дополнительного потребления памяти
`tbb::memory_pool_allocator< T, P >`	Класс предназначен для использования пулов памяти в сочетании с STL-контейнерами. На момент написания книги это средство считалось «ознакомительным» (но, вероятно, станет полноправным в будущем). Чтобы разрешить его, добавьте директиву `#define TBB_PREVIEW_MEMOKY_POOL`
`tbb::scalable_allocator< T >`	Масштабируемый распределитель памяти. Прямое обращение окажется неудачным, если библиотека `TBBmalloc` недоступна
`tbb::allocator< T >`	Этот класс выбирает `tbb::scalable_allocator`, если тот доступен, или стандартную `malloc` в противном случае. Прямое обращение работает, даже когда библиотека `TBBmalloc` недоступна
`tbb::zero_allocator< T [, Allocator] >`	Перенаправляет запросы на выделение памяти распределителю `Allocator` (по умолчанию `tbb_allocator`) и обнуляет ее перед возвратом запросившей стороне

Рис. 7.14 ❖ Классы, предлагаемые масштабируемым распределителем памяти TBB

scalable_allocator

Шаблон `scalable_allocator` выделяет и освобождает память, масштабируясь в соответствии с количеством процессоров. Использование `scalable_allocator` вместо `std::allocator` может повысить производительность программы. Па-

мять, выделенную с помощью scalable_allocator, следует освобождать также средствами scalable_allocator, а не std::allocator.

Шаблон scalable_allocator требует наличия библиотеки TBBmalloc. Если эта библиотека отсутствует, то обращения к scalable_allocator завершатся ошибкой. Но все остальные распределители (tbb_allocator, cached_aligned_allocator и zero_allocator) в этом случае обращаются к malloc и free.

Класс определен в заголовке #include <tbb/scalable_allocator.h>, который не включается файлом tbb/tbb.h, включающим все остальные заголовки.

tbb_allocator

Шаблон tbb_allocator выделяет и освобождает память с помощью библиотеки TBBmalloc, если она доступна, а в противном случае довольствуется malloc и free. Распределители cache_alligned_allocator и zero_allocator пользуются tbb_allocator, поэтому точно так же готовы довольствоваться malloc, но scalable_allocator не таков – если библиотека TBBmalloc отсутствует, обращение к нему завершается ошибкой. Класс определен в заголовке #include <tbb/tbb_allocator.h>.

zero_allocator

Шаблон zero_allocator выделяет память и обнуляет ее. Шаблон zero_allocator<T,A> можно конкретизировать любым классом A, моделирующим концепцию Allocator. По умолчанию в качестве A используется tbb_allocator. Распределитель zero_allocator перенаправляет запросы на выделение памяти A и обнуляет ее перед возвратом запросившей стороне. Класс определен в заголовке #include <tbb/tbb_allocator.h>.

cached_aligned_allocator

Шаблон cached_aligned_allocator предлагает как масштабируемость, так и защиту от ложного разделения – тем, что каждый выделенный блок памяти начинается на границе строки кеша.

Используйте cache_aligned_allocator, только когда ложное разделение представляет реальную проблему (см. рис. 7.2). За его функциональность приходится расплачиваться памятью, потому что даже для маленьких объектов память выделяется блоками, кратными размеру строки кеша. Обычно блок дополняется до 128 байт. Так что выделение большого количества маленьких объектов с помощью cache_aligned_allocator может заметно повысить потребление памяти.

Имеет смысл попробовать и сравнить производительность tbb_allocator и cache_aligned_allocator для одного и того же приложения.

Отметим, что защита от ложного разделения двух объектов гарантируется, только если память для обоих выделена с помощью cache_aligned_allocator. Например, если один объект выделен cache_aligned_allocator<T>, а другой – еще

каким-то способом, гарантии отсутствия ложного разделения нет, потому что блок, выделенный `cache_aligned_allocator<T>`, начинается на границе строки кеша, но не обязательно продолжается до конца строки кеша. Если выделяется память для массива или структуры, то, поскольку лишь начало блока выровнено, отдельные элементы массива или структуры могут оказаться в тех же строках кеша, что и другие элементы. Пример такого поведения вместе с применением дополнения для размещения элементов в отдельных строках кеша см. на рис. 7.3.

Класс определен в заголовке `#include <tbb/cache_alligned_allocator.h>`.

Поддержка пула памяти: memory_pool_allocator

Распределители из пула – очень эффективный метод выделения памяти для большого количества объектов фиксированного размера *P*. Первое обращение к распределителю особое, в нем требуется зарезервировать достаточно памяти для размещения *T* объектов размера *P*. При каждом следующем обращении, когда мы просим предоставить блок памяти, распределитель возвращает кратное *P* смещение относительно начала выделенной области. Это гораздо эффективнее вызова оператора `new` для каждого запроса, поскольку отсутствуют служебные накладные расходы, без которых универсальный распределитель не смог бы обслуживать многочисленные запросы блоков разного размера.

Этот класс предназначен главным образом для реализации пулов памяти в STL-контейнерах. На момент написания книги это средство считалось «ознакомительным» (но, вероятно, станет полноправным в будущем). Чтобы разрешить его, добавьте директиву `#define TBB_PREVIEW_MEMOKY_POOL`:

```
#define TBB_PREVIEW_MEMORY_POOL 1
#include <tbb/memory_pool.h>
```

Поддержка выделения памяти для массивов: aligned_space

Этот шаблонный класс (`aligned_space`) выделяет достаточно выровненной памяти для размещения массива *T[N]*. Он не конструирует и не уничтожает элементы – это функция клиента. Обычно объекты типа `aligned_space` используются как локальные переменные или поля в тех случаях, когда нужен блок неинициализированной памяти фиксированного размера. Класс определен в файле `#include <tbb/aligned_space.h>`.

Избирательная подмена new и delete

Есть много причин для разработки собственных операторов new/delete, в том числе контроль ошибок, отладка, оптимизация и сбор статистики об использовании.

Существуют варианты new/delete для отдельных объектов и для массивов: глобальные (::operator new, ::operator new[], ::operator delete и ::operator delete[]) или для конкретного класса (для класса X это будут X::operator new, X::operator new[], X::operator delete и X::operator delete[]). Кроме того, в C++11 для каждого из этих операторов определены версии с возбуждением и без возбуждения исключений и с размещением. Наконец, C++17 добавляет необязательный параметр выравнивания во все версии new.

Если мы хотим глобально подменить все операторы new/delete и не требуем ничего специфического, то следует использовать библиотеку прокси. Она заодно подменяет malloc/free и родственные им функции C.

Если специфические потребности есть, то обычно перегружают операторы в классе, а не глобальные. В этом разделе показано, как подменить глобальные операторы new и delete, но идею можно адаптировать и для отдельных классов. Мы продемонстрируем версии с возбуждением и без возбуждения исключений, но не станем переопределять версии с размещением, поскольку они никакой памяти не выделяют. Мы также не реализовали версии с параметром выравнивания (C++17). Применяя те же идеи, можно подменить операторы new/delete в отдельных классах, в таком случае вы можете реализовать версии с размещением и выравниванием. Всем этим занимается TBB, если подключена библиотека прокси.

На рис. 7.15 и 7.16 показана подмена new и delete, а на рис. 7.17 – их использование. Подмене подлежат все версии new и delete сразу, так что всего получается четыре варианта new и столько же delete. Разумеется, необходимо скомпоновать программу с библиотекой масштабируемого выделения памяти.

Мы решили в этом примере игнорировать обработчик new из-за проблем с потокобезопасностью, поэтому он всегда возбуждает исключение std::bad_alloc(). У базовой сигнатуры есть вариация, включающая дополнительный параметр типа const std::nothrow_t&, которая означает, что оператор не возбуждает исключения, а возвращает NULL в случае ошибки выделения памяти. Эти четыре оператора, не возбуждающих исключений, могут использоваться в библиотеках времени выполнения для C.

Мы не обязаны инициализировать планировщик задач, так чтобы он умел использовать распределитель памяти. Но в этом примере мы его инициализируем, потому что используем parallel_for для демонстрации выделения и освобождения памяти в нескольких задачах. Для работы с распределителем памяти нужно включить только заголовочный файл tbb/tbb_allocator.h.

```
#include <tbb/parallel_for.h>
#include <tbb/tbb_allocator.h>

// Повторные попытки в цикле не производятся,
// потому что предполагается, что scalable_malloc и так делает
// все необходимое для выделения памяти, поэтому его повторный
// вызов не улучшит ситуацию
//
// std::new_handler не используется, потому что это
// невозможно сделать переносимым и потокобезопасным способом
//
// Мы возбуждаем std::bad_alloc(), когда scalable_malloc
// возвращает NULL (и возвращаем NULL, если это не возбуждающая
// исключений версия)

void* operator new (size_t size) throw (std::bad_alloc)
{
    if (size == 0) size = 1;
    if (void* ptr = scalable_malloc (size))
        return ptr;
    throw std::bad_alloc (  );
}

void* operator new[] (size_t size) throw (std::bad_alloc)
{
    return operator new (size);
}

void* operator new (size_t size, const std::nothrow_t&) throw ()
{
    if (size == 0) size = 1;
    if (void* ptr = scalable_malloc (size))
        return ptr;
    return NULL;
}

void* operator new[] (size_t size, const std::nothrow_t&) throw ()
{
    return operator new (size, std::nothrow);
}
```

Рис. 7.15 ❖ Демонстрация подмены операторов new (tbb_nd.cpp)

```
void operator delete (void* ptr) throw ()
{
    if (ptr != 0) scalable_free (ptr);
}

void operator delete[] (void* ptr) throw ()
{
    operator delete (ptr);
}

void operator delete (void* ptr, const std::nothrow_t&) throw ()
{
    if (ptr != 0) scalable_free (ptr);
}

void operator delete[] (void* ptr, const std::nothrow_t&) throw ()
{
    operator delete (ptr, std::nothrow);
}
```

Рис. 7.16 ❖ Продолжение предыдущего листинга, подмена операторов delete

```
int main (int argc, char** argv)
{
  const size_t size = 1000;
  const size_t chunk = 100;

  // для выделения памяти под этот массив целых
  // будет вызываться scalable_malloc
  int *p = new int[size];

  tbb::parallel_for (size_t{0}, size, [=](size_t chunk)
                     {
                       // для выделения памяти под этот
                       // массив целых будет вызываться
                       // scalable_malloc
                       int *p = new int [chunk];

                       // для освобождения памяти, занятой
                       // этим массивом целых, будет
                       // вызываться scalable_free
                       delete[] p;
                     } );

  return 0;
}
```

Рис. 7.17 ❖ Программа для демонстрации подмены new/delete

Настройка производительности: некоторые рычаги управления

TBB предлагает несколько специальных средств, относящихся к получению памяти от ОС, поддержке больших страниц и сбросу внутренних буферов. Все они нужны для тонкой настройки производительности.

Большие страницы (huge pages или large pages в Windows) используются для повышения производительности программ, потребляющих очень много памяти. Для использования больших страниц необходима поддержка со стороны процессора и операционной системы, и только в этом случае мы можем включить работу с ними в приложение. К счастью, в большинстве систем все это есть, и TBB это поддерживает.

Что такое большие страницы?

Процессор, как правило, выделяет память блоками размера 4K, которые называются страницами. Система виртуальной памяти использует таблицы страниц для отображения виртуальных адресов на физические. Не слишком вдаваясь в детали, скажем, что чем больше страниц памяти приложение использует, тем больше необходимо дескрипторов, а когда дескрипторов слишком много, начинает страдать производительность – по разным причинам. Чтобы как-то решить эту проблему, современные процессоры поддерживают дополнительные, гораздо большие размеры страниц (например, 4 МБ). Если программа потребляет 2 ГБ памяти, то при размере страницы 4K ей понадобилось бы 524 288 дескрипторов страниц. И лишь 512 дескрипторов страниц размером 4 МБ, и всего два гигабайтных дескриптора.

Поддержка больших страниц в TBB

Если мы хотим, чтобы TBB использовала большие страницы при выделении памяти, нужно явно попросить об этом, вызвав функцию `scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES,1)` или присвоив переменной среды `TBB_MALLOC_USE_HUGE_PAGES` значение 1. Переменная среды полезна при подмене стандартных функций из семейства malloc с помощью библиотеки `tbbmalloc_proxy`.

В результате модифицируются все алгоритмы, в которых участвует масштабируемый распределитель памяти TBB (вне зависимости от способа использования: библиотека прокси, функции C или классы C++). Вызов функции приоритетнее установки переменной среды. Эти средства, безусловно, не предназначены для повседневного использования, они адресованы самопровозглашенным «фанатикам контроля» и предлагают действенные способы оптимизировать производительность в конкретных ситуациях. Мы рекомендуем тщательно оценить их влияние на производительность приложения в целевой среде, прежде чем решиться на включение в код.

Разумеется, оба метода предполагают, что система или ядро сконфигурированы для выделения больших страниц. Распределитель памяти TBB поддерживает также предварительно выделенные и прозрачные большие страницы,

которые автоматически выделяются ядром Linux, когда это оправдано. Большие страницы – не панацея, они могут навредить производительности, если применение не было тщательно взвешено.

Функции, перечисленные на рис. 7.18, определены в файле `tbb/tbb_allocator.h`.

`int scalable_allocation_mode(int mode, intptr_t value)`
`mode = TBBMALLOC_USE_HUGE_PAGES` или
`TBBMALLOC_SET_SOFT_HEAP_LIMIT`
	Задать режим работы распределителя памяти TBB

Переменные среды: `TBB_MALLOC_USE_HUGE_PAGES`	Когда равна 1, включается использование больших страниц распределителем, если это поддерживается операционной системой
`int scalable_allocation_command(int cmd, void *reserved)` `reserved` должен быть равен нулю	Вызов команд, управляющих поведением распределителя

Рис. 7.18 ❖ Способы уточнения поведения масштабируемого распределителя памяти TBB

scalable_allocation_mode(int mode, intptr_t value)

Функцию `scalable_allocation_mode` можно использовать для настройки поведения масштабируемого распределителя памяти. Этому служат описанные в следующих двух разделах аргументы. Функция возвращает `TBBMALLOC_OK`, если операция завершилась успешно, и `TBBMALLOC_INVALID_PARAM`, если значение параметра `mode` не совпадает ни с одним из допустимых или значение `value` не разрешено в данном режиме.

Может быть также возвращено значение `TBBMALLOC_NO_EFFECT` (объяснения см. ниже).

TBBMALLOC_USE_HUGE_PAGES

```
scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES,1)
```

Этот вызов разрешает распределителю использовать большие страницы, если это поддерживается операционной системой. Если значение второго параметра равно 0, то большие страницы запрещены. Присваивание переменной среды `TBB_MALLOC_USE_HUGE_PAGES` значения 1 оказывает тот же эффект, что вызов `scalable_allocation_mode`. Режим, установленный вызовом `scalable_allocation_mode`, отменяет режим, установленный переменной среды. Функция возвращает `TBBMALLOC_NO_EFFECT`, если на данной платформе большие страницы не поддерживаются.

TBBMALLOC_SET_SOFT_HEAP_LIMIT

```
scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)
```

Этот вызов задает пороговый размер в байтах памяти, получаемой распределителем от операционной системы. Превышение порога заставит распределитель освободить память, занятую его внутренними буферами, но не помешает ему запрашивать больше памяти при необходимости.

int scalable_allocation_command(int cmd, void *param)

Функция scalable_allocation_command заставляет распределитель выполнить действие, заданное первым параметром. Второй параметр зарезервирован и должен быть равен нулю. Функция возвращает TBBMALLOC_OK, если операция завершилась успешно, TBBMALLOC_INVALID_PARAM, если второй параметр не равен нулю или если cmd не совпадает ни с одним из допустимых значений (TBBMALLOC_CLEAN_ALL_BUFFERS или TBBMALLOC_CLEAN_THREAD_BUFFERS). Возможен также возврат значения TBBMALLOC_NO_EFFECT в случаях, описанных ниже.

TBBMALLOC_CLEAN_ALL_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)
```

Эта функция очищает внутренние буферы памяти распределителя и, возможно, снижает общее потребление памяти. Но время последующего выделения памяти может при этом возрасти. Команда не предназначена для частого использования, рекомендуется тщательно взвешивать ее влияние на производительность. Функция возвращает TBBMALLOC_NO_EFFECT, если не был освобожден ни один буфер.

TBBMALLOC_CLEAN_THREAD_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)
```

Эта функция очищает внутренние буферы памяти, но только для вызывающего потока. Время последующего выделения памяти может возрасти, рекомендуется тщательно взвешивать влияние на производительность. Функция возвращает TBBMALLOC_NO_EFFECT, если не был освобожден ни один буфер.

Резюме

Использование масштабируемого распределителя памяти – важный элемент любой параллельной программы. Влияние на производительность может быть очень значительным. Без масштабируемого распределителя памяти возникают серьезные проблемы из-за состязания за возможность выделения, ложного разделения и других бесполезных пересылок из одного кеша в другой. Средства масштабируемого выделения памяти в TBB (TBBmalloc) распространяются на использование оператора new, а также на явные вызовы функций из семейства malloc. Все их можно применять непосредственно или автоматически подменить с помощью библиотеки прокси. Средства масштабируемого выделения памяти в TBB можно использовать вне зависимости от того, используются прочие части TBB или нет. И наоборот, TBB можно применять совместно с любым другим распределителем памяти (TBBmalloc, tcmalloc, jemalloc, malloc и т. д.). Библиотека TBBmalloc и сегодня очень популярна и, безусловно, является одним из лучших масштабируемых распределителей памяти на рынке.

Глава 8

TBB и параллельные паттерны

Говорят, что история не повторяется, но рифмуется[1].

Можно было бы сказать, что и программное обеспечение рифмуется. Возможно, мы и не пишем один и тот же код раз за разом, но в решаемых нами задачах и написанном для их решения коде часто выявляются паттерны. Мы можем учиться на похожих решениях.

В этой главе мы рассмотрим паттерны, доказавшие свою эффективность и масштабируемость, и покажем, как их можно реализовать средствами TBB (рис. 8.1). Чтобы добиться масштабируемого распараллеливания, следует уделить особое внимание параллелизму данных, именно это является наилучшей стратегией. Способ кодирования должен поощрять разбиение задачи на несколько подзадач, количество которых может расти с увеличением размера задачи; чем больше задач, тем лучше масштабируемость. Опираясь на продвигаемые в этой главе паттерны, мы сможем обеспечить изобилие задач и добиться масштабируемости алгоритмов.

Мы можем научиться «мыслить параллельно», глядя, как другие достигли успехов в этом деле. Стоя на плечах гигантов, мы можем заглянуть дальше, чем они.

Эта глава посвящена обучению на опыте наших предшественников, и, разумеется, это поможет нам научиться лучше использовать TBB. Мы считаем паттерны вдохновляющим примером и полезным инструментом для «параллельного мыслителя», а не описываем их как идеальную таксономию программирования.

Параллельные паттерны и параллельные алгоритмы

В главе 2 мы говорили, что рецензенты предложили нам рассматривать «параллельные алгоритмы TBB» как *паттерны*, а не алгоритмы. Быть может, это и резонно, но чтобы не порывать с терминологией, которая использовалась в TBB на протяжении многих лет, мы все же продолжаем говорить об обобщенных параллельных алгоритмах – и в книге, и в документации. Суть дела от этого не

---

[1] Это высказывание приписывают Марку Твену. – *Прим. перев.*

меняется – они дают нам возможность учиться на опыте тех, кто исследовал до нас оптимальные решения, воплотившиеся в этих паттернах. Учиться не только *использовать* их, но и отдавать предпочтение этим конкретным паттернам (алгоритмам) перед другими возможными подходами, потому что они лучше работают (достигают более высокой степени масштабируемости).

Концепция паттерна	Название паттерна	Шаблон TBB
Мощный толчок для всех паттернов: возможность без ограничений вкладывать одни паттерны в другие	**вложенность**	*TBB целиком*
Лучший из всех паттернов: разбиение работы на одинаковые независимые задачи	**отображение**	`parallel_for`, `parallel_invoke`
Обобщение отображения на случай инкрементного добавления задач	**куча задач** (*обобщенное отображение с инкрементным добавлением задач*)	`parallel_reduce`
Типичная операция: разбиение работы на независимые задачи с целью вычисления частичных результатов, которые затем редуцируются (объединяются) в окончательный результат	**редукция**	`parallel_scan`
Частный случай редукции для параллельного вычисления префикса: `y[i]=y[i-1] op x[i]`	**сканирование** *иногда называется «префикс»*	`parallel_invoke`, `task_group`, `flow_graph`
Классический и мощный: поток управления разделяется на два потока (задачи), которые потом соединяются	**разветвление–соединение**	`parallel_for`, `parallel_reduce`, `parallel_invoke`, `task_group`, `flow_graph`, `parallel_sort`
Частный случай разветвления–соединения: работа разделяется на подзадачи рекурсивно	**разделяй и властвуй**	`parallel_for`, `parallel_reduce`, `parallel_invoke`, `task_group`, `flow_graph`, `parallel_sort` *плюс средства отмены (глава 15)*
Частный случай разветвления–соединения: работа разделяется на подзадачи рекурсивно с отсечением, чтобы сократить пространство поиска. Очень ценной может быть поддержка отмены (глава 15)	**метод ветвей и границ**	`parallel_pipeline`, `flow_graph`
Обманчиво мощный: задачи связываются как производитель и потребитель; поток данных регулярный и не изменяется	**конвейер**	`flow_graph`
Хорошо справляется с хаотичными реальными потоками: задачи связываются как производитель и потребитель; взаимодействие между подзадачами нерегулярное и может изменяться	**событийно-управляемая координация**	

Рис. 8.1 ❖ Шаблоны TBB, выражающие важные «паттерны, которые работают»

Паттерны определяют классификацию алгоритмов, проектных решений и т. д.

Ценность объектно-ориентированного программирования была описана «бандой четырех» (Гамма, Хелм, Джонсон и Влиссидес) и их знаменательной работой «Design Patterns: Elements of Reusable Object-Oriented Software» (Addison-Wesley)[1]. Многие отдают дань этой книге, считая, что она привнесла больше порядка в мир объектно-ориентированного программирования. В ней коллективная мудрость сообщества собрана и сведена к простым «паттернам», получившим названия, чтобы их можно было упоминать в разговоре.

Книга Mattson, Sanders, and Massingill «Patterns for Parallel Programming» точно так же вобрала в себя коллективную мудрость сообщества параллельного программирования. Специалисты применяют типичные приемы и для обсуждения выработали особую терминологию. Памятуя о параллельных паттернах, программисты могут быстро приобрести навыки параллельного программирования, как случилось с объектно-ориентированными программистами, прочитавшими знаменитую книгу банды четырех.

Книга «Patterns for Parallel Programming» толще этой, и читать ее очень непросто, но благодаря помощи Тима Мэттсона мы можем кратко изложить, как эти паттерны соотносятся с TBB.

Тим с коллегами считают, что программисту, разрабатывающему параллельную программу, нужно вести поиск в четырех пространствах проектирования:

1. Выявление конкурентности.
   В этом пространстве проектирования мы работаем внутри предметной области, стремясь выявить имеющийся параллелизм и раскрыть его для последующего проектирования алгоритма. TBB упрощает этот процесс, побуждая нас искать максимально много задач, не думая при этом, как отобразить их на аппаратные потоки. Мы также предоставляем информацию о том, как лучше разбить задачу на две, если одна считается слишком большой. Зная это, TBB автоматически разбивает задачи, стремясь равномерно распределить работу между процессорными ядрами. Чем больше задач, тем лучше масштабируется алгоритм.
2. Алгоритмические структуры.
   Это пространство проектирования воплощает высокоуровневую стратегию организации параллельного алгоритма. Мы должны решить, как хотим организовать технологический процесс. На рис. 8.1 перечислены важные паттерны, с этим списком можно сверяться при выборе паттерна, который лучше всего отвечает нашим целям. «Паттерны, которые работают» занимают центральное место в книге McCool, Robison, and Reinders «Structured Parallel Programming» (Elsevier).
3. Поддерживающие структуры.
   На этом шаге алгоритмическая стратегия превращается в реальный код. Мы думаем о том, как будет организована параллельная программа

[1] *Гамма Эрих, Хелм Ричард, Джонсон Ральф, Влиссидес Джон.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2016.

и с помощью каких технических средств управлять разделяемыми (особенно изменяемыми) данными. Эти соображения критически важны и оказывают влияние на весь процесс параллельного программирования. TBB хорошо спроектирована и побуждает рассуждать на подходящем уровне абстракции, так что этот процесс хорошо ложится на TBB (надеемся, что нам удалось объяснить это в книге).

4. Механизмы реализации.
   Это пространство проектирования включает управление потоками и синхронизацию. TBB берет на себя все управление потоками, оставляя нам продумывание задач на более высоком уровне проектирования. При использовании TBB большинству программистов удается избежать кодирования и отладки явных механизмов синхронизации. Алгоритмы TBB (глава 2) и потоковые графы (глава 3) также стремятся минимизировать явную синхронизацию. В главе 5 обсуждается, какие есть механизмы синхронизации, на случай, если без них все-таки не обойтись, а в главе 6 предлагается использовать контейнеры и поточно-локальную память, чтобы свести к минимуму потребности в явной синхронизации. Язык паттернов может послужить путеводной звездой при создании более качественных сред параллельного программирования и поможет наилучшим способом использовать TBB при написании параллельных программ.

Паттерны, которые работают

Вооружившись языком паттернов, мы должны рассматривать их как инструменты. Мы уделяем особое внимание паттернам, которые доказали свою полезность при разработке большинства масштабируемых алгоритмов. Мы знаем, что два обязательных условия для достижения масштабируемого параллелизма – хорошая локальность данных и предотвращение накладных расходов. По счастью, предложено много хороших стратегий для достижения этих целей, и благодаря TBB все они в нашем распоряжении (см. таблицу на рис. 8.1). В TBB также продуманы детали настройки под конкретную машину, в т. ч. управление уровнем зернистости и эффективное использование кеша.

В этих терминах TBB берет на себя детали реализации, а мы можем программировать на верхнем уровне. Именно поэтому код, написанный с помощью TBB, является переносимым – машинные зависимости остаются внутри TBB. А TBB, в свою очередь, благодаря таким алгоритмам, как заимствование работ, помогает минимизировать настройку, необходимую для переноса TBB на другую платформу. Разделение абстракции алгоритмической стратегии на семантику и реализацию оказалось исключительно действенным на практике. Это разделение позволяет рассуждать о высокоуровневой структуре алгоритма и о низкоуровневых деталях (часто машинно зависимых) по отдельности.

Паттерны предлагают общий словарь для обсуждения подходов к решению задач и позволяют повторно использовать передовые практики. Паттерны свободно пересекают границы языков, моделей программирования и даже

компьютерных архитектур, их можно использовать вне зависимости от того, есть ли в системе программирования конкретное средство, явно поддерживающее данный паттерн. К счастью, проектировщики TBB сознательно акцентировали внимание на паттернах, доказавших, что их применение ведет к хорошо структурированным, простым для сопровождения и эффективным программам. Многие из этих паттернов к тому же детерминированны (или могут работать в детерминированном режиме – см. главу 16), т. е. дают один и тот же результат при каждом выполнении. Детерминированность – полезное свойство, поскольку делает программы более понятными и простыми для тестирования, отладки и сопровождения.

Параллелизм данных одерживает победу

В общем и целом лучшая стратегия масштабируемого параллелизма – это параллелизм данных. Мы будем смотреть на вещи широко и определим параллелизм данных как любой вид параллелизма, который расширяется вместе с увеличением размера данных или вообще с размером задачи. В типичном случае данные разбиваются на порции, и каждая порция обрабатывается отдельной задачей. Иногда разбиение плоское, иногда – рекурсивное. Главное – чтобы при увеличении объема данных порождалось больше задач.

В нашем определении не важно, применяются ли к порциям одинаковые или разные операции. Вообще говоря, параллелизм данных применим как к регулярным, так и к нерегулярным задачам. Поскольку параллелизм данных – лучшая стратегия масштабируемого параллелизма, его аппаратная поддержка обычно присутствует во всех видах оборудования: ЦП, GPU, специализированных заказных интегральных схемах (ASIC) и программируемых пользователем вентильных матрицах (ППВМ). В главе 4 технология SIMD обсуждалась именно для того, чтобы составить представление о такой аппаратной поддержке.

Противоположностью параллелизму данных является функциональная декомпозиция (или параллелизм задач) – подход, ставящий во главу угла параллельное выполнение разных функций программы. В лучшем случае функциональная декомпозиция повышает производительность в постоянное число раз. Например, если три функции программы `f`, `g` и `h` выполнять параллельно, то производительность вырастет самое большее в три раза, а на практике и того меньше. Иногда функциональная декомпозиция помогает немного повысить степень параллелизма и достичь целевого показателя, но она не масштабируется и потому не должна быть нашей основной стратегией.

Паттерн Вложенность

Вложенность (рис. 8.2) может показаться очевидным и естественным делом, но в мире параллельного программирования это не так. TBB упрощает нам жизнь – вложенность просто работает, не вызывая серьезных проблем с перегруженностью, свойственных другим моделям, например OpenMP.

Отметим два следствия, вытекающих из поддержки вложенности:

- нам необязательно знать, находимся ли мы на параллельном или последовательном участке программы, принимая решение о вызове шаблона TBB. Поскольку при работе с TBB мы создаем только задачи, нет нужды беспокоиться о чрезмерном количестве потоков;
- не нужно думать о том, используется ли в библиотеке, с которой мы компонуем программу, TBB и вообще распараллеливание.

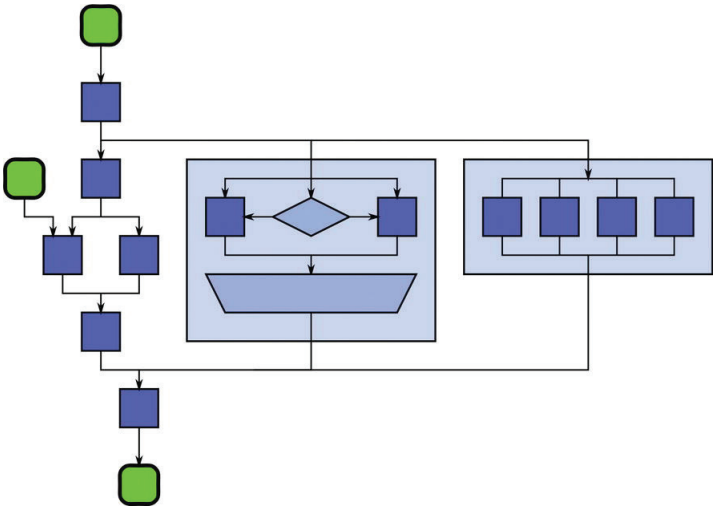


Рис. 8.2 ❖ Паттерн Вложенность: композиционный паттерн, позволяющий составлять иерархию из других паттернов. Вложенность означает, что любой блок задачи в паттерне можно заменить паттерном с такой же конфигурацией входов и выходов и зависимостями

Вложенность можно рассматривать как метапаттерн, поскольку она выражает мысль о том, что паттерны допускают иерархическую композицию. Это важно для модульного программирования. Вложенность сплошь и рядом используется в последовательном программировании для обеспечения компонуемости и сокрытия информации, но может стать проблемой при переходе к параллельному программированию. Ключ к реализации вложенности – использование факультативного, а не обязательного параллелизма. В этой области TBB выделяется на фоне других моделей.

Важность вложенности была ясна уже в момент появления TBB в 2006 году, так что она хорошо поддержана во всех частях библиотеки. С другой стороны, OpenMP API появился в 1997 году, когда важность этого паттерна для будущих машин еще не была в полной мере осознана. В результате паттерн Вложенность в OpenMP не поддерживается. Из-за этого OpenMP гораздо труднее использовать для чего-либо, кроме приложений, основная задача которых – вложенные циклы сплошных вычислений. Именно такого рода приложения занимали наши умы, когда в 1980-х и 1990-х годах создавались OpenMP и его предшественники. А во времена создания TBB во главу угла уже был постав-

лен паттерн Вложенность с его модульностью и компонуемостью (мы отдаем должное исследовательскому проекту Cilk в МТИ, поскольку эта пионерская работа сильно повлияла на направление наших размышлений – дополнительные сведения об истоках и влияниях, в т. ч. о Cilk, см. в приложении А).

Паттерн Отображение

Паттерн Отображение (рис. 8.3) лучше всего пригоден для параллельного программирования: разбиения работы на одинаковые независимые части, которые работают абсолютно независимо. Получается регулярное распараллеливание, которое часто называют *естественным*. Таким образом, этот паттерн представляется очевидным, когда имеются независимые параллельные работы. И он действительно допускает как эффективное распараллеливание, так и эффективную векторизацию.

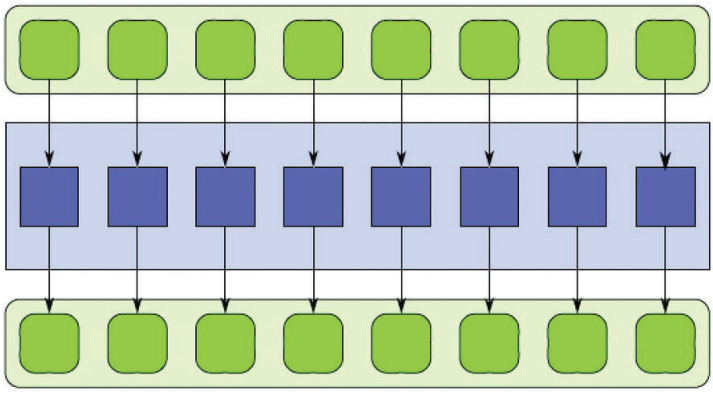


Рис. 8.3 ❖ Паттерн Отображение:
функция применяется ко всем элементам коллекции,
и обычно порождается новая коллекция той же формы, что исходная

В этом паттерне нет *никакого* изменяемого состояния, разделяемого частями: функция отображения (та, что выполняет независимые работы) должна быть «чистой» в том смысле, что не модифицирует разделяемое состояние, поскольку модификация нарушила бы идеальную независимость, что могло бы привести к недетерминированности вследствие гонки за данные, а стало быть, к неопределенному поведению, в т. ч. возможному сбою приложения. Скрытые разделяемые данные могут присутствовать при использовании сложных структур данных, например `std::share_ptr`.

Паттерн Отображение применяется, например, для гамма-коррекции и пороговой бинаризации изображений, преобразования цветового пространства, выборки по методу Монте-Карло и в алгоритме трассировки лучей. Для эффективной реализации отображения подходит алгоритм TBB `parallel_for` (см. пример на рис. 8.4). Кроме того, если степень параллелизма невысока, то можно использовать `parallel_invoke`, но тогда не удастся добиться высокой масштаби-

руемости, если только нет параллелизма на других уровнях (например, внутри вызываемых функций).

```
tbb::parallel_for( size_t(0), n, [&](size_t i) {
  Foo(i);
} );
```

Рис. 8.4 ❖ Паттерн Отображение, реализованный с помощью `parallel_for`

Паттерн Куча работ

Паттерн Куча работ (workpile) является обобщением паттерна Отображение на случай, когда каждый экземпляр (функция отображения) может генерировать дополнительные экземпляры. Иными словами, работу можно добавлять в «кучу» работ, ожидающих выполнения. Это можно использовать, например, при рекурсивном обходе дерева, когда мы хотим генерировать экземпляры для обработки всех потомков каждого узла. В отличие от паттерна Отображение, теперь общее количество экземпляров функции отображения заранее неизвестно, и структура работы не регулярна. Поэтому паттерн Куча работ труднее поддается векторизации (глава 4). Для эффективной реализации Кучи работ средствами TBB служит алгоритм `parallel_do` (глава 2).

Паттерны редукции (Редукция и Сканирование)

Паттерн Редукция (рис. 8.5) можно рассматривать как операцию отображения, в которой каждая подзадача порождает частичный результат, а все такие результаты затем следует объединить в окончательный ответ. Паттерн Редукция объединяет несколько частичных результатов с помощью ассоциативной «комбинирующей функции». Поскольку эта функция ассоциативна, допустимы различные порядки объединения при разных прогонах – это одновременно благословение и проклятие. Благословение – потому что реализация вольна выбирать наиболее эффективный порядок и тем добиваться максимальной производительности. А проклятие – потому что возникает недетерминированность, если результаты при разных прогонах могут отличаться из-за округления или переполнения. Объединение с целью нахождения максимума или логической конъюнкции частичных результатов не подвержено таким проблемам. Но сложение чисел с плавающей точкой недетерминированно из-за различных ошибок округления при разном порядке вычислений.

TBB предлагает недетерминированный (максимальная производительность) и детерминированный (с небольшой потерей производительности) варианты редукции. Термин «детерминированный» относится только к порядку вычислений при разных прогонах. Если комбинирующая функция детерминированная, как, например, логическая конъюнкция, то и при недетерминированном порядке вычислений в `parallel_reduce` результат все равно будет детерминирован.

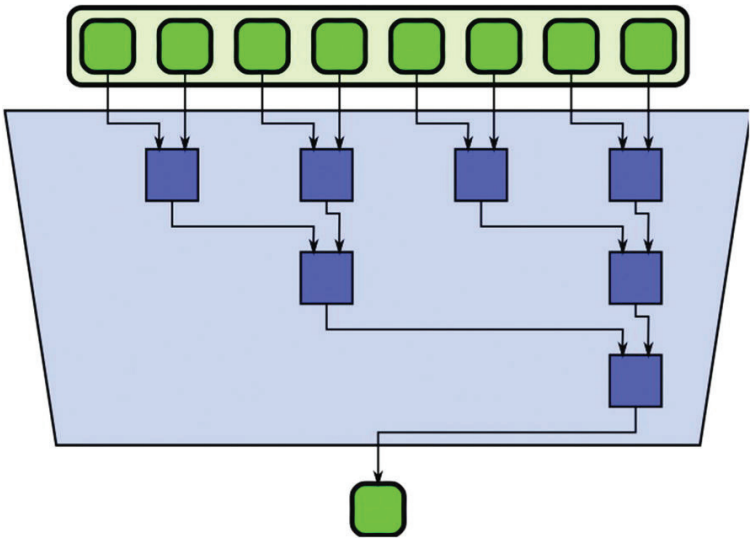


Рис. 8.5 ❖ Паттерн Редукция: подзадачи порождают частичные результаты, которые объединяются в окончательный ответ

Типичные комбинирующие функции – сложение, умножение, максимум, минимум и булевы операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ. Для реализации недетерминированной редукции можно использовать алгоритм `parallel_reduce` (глава 2), а для реализации детерминированной – алгоритм `parallel_deterministic_reduce` (глава 16). Тот и другой позволяют определять собственные комбинирующие функции.

Паттерн Сканирование (рис. 8.6) параллельно вычисляет префикс (`y[i]=y[i-1] op x[i]`). Как и в других видах редукции, распараллеливание возможно, только если операция `op` ассоциативна. Это может быть полезно в ситуациях, где зависимости, казалось бы, принципиально последовательные. Многие бывают удивлены, узнав, что это вообще можно сделать масштабируемым способом. На рис. 8.7 показано, как может выглядеть последовательный код. В параллельной версии операций больше, чем в последовательной, но зато она допускает масштабирование. Для реализации операций сканирования в TBB служит алгоритм `parallel_scan` (глава 2).

Паттерн Разветвление–соединение

Паттерн Разветвление–соединение (рис. 8.8) рекурсивно разбивает задачу на части и может использоваться как для регулярного, так и для нерегулярного распараллеливания. Полезен для реализации стратегии «разделяй и властвуй» (иногда так называется сам паттерн) и метода ветвей и границ (так тоже называют сам паттерн). Не следует путать Разветвление–соединение с барьерами. Барьер – это один из механизмов синхронизации нескольких потоков, когда каждый поток должен ждать, пока остальные подойдут к барьеру, который они

могут пересечь только все вместе. При соединении тоже нужно ждать, пока все потоки доберутся до общей точки, но разница в том, что после пересечения барьера продолжают работу все потоки, а после соединения – только один. Работа, которая некоторое время выполняется независимо, затем нуждается в барьерной синхронизации, а потом снова выполняется независимо, – по сути дела, то же самое, что многократно повторяемый паттерн Отображение со вставкой барьеров между итерациями. На такие программы распространяется закон Амдала (см. предисловие), поскольку время тратится на ожидание, а не на работу (т. е. имеет место сериализация).

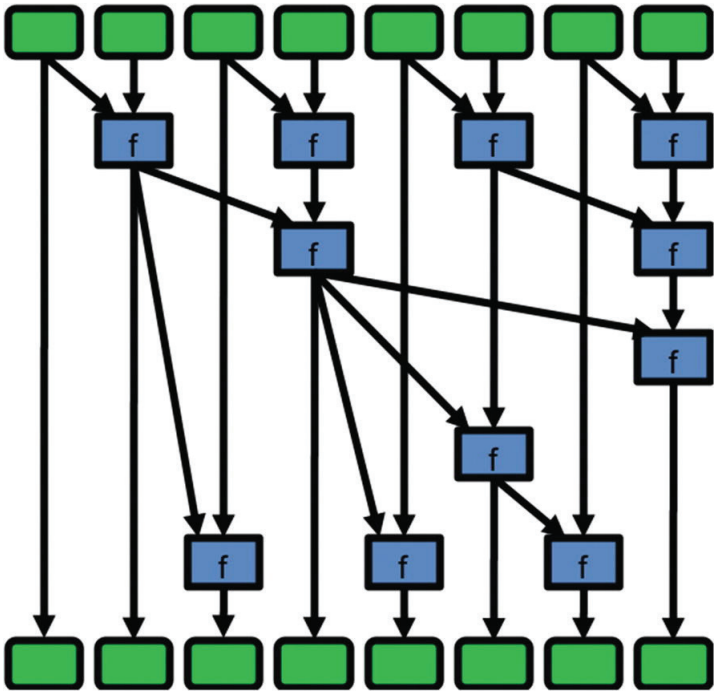


Рис. 8.6 ❖ Паттерн Сканирование: наглядное представление о дополнительных операциях, необходимых для масштабируемости

```
void my_add_iscan(
  const float a[],    // входной массив
  float b[],          // выходной массив
  size_t n ) {        // количество элементов
  if (n>0) b[0]=a[0]; // экв. предположению о том, что b[i-1] равно 0
  for (int i=1; i < n; ++i)
    b[i] = b[i-1] + a[i]; // последующие итерации зависят от предыдущих
}
```

Рис. 8.7 ❖ Последовательный код для выполнения операции сканирования

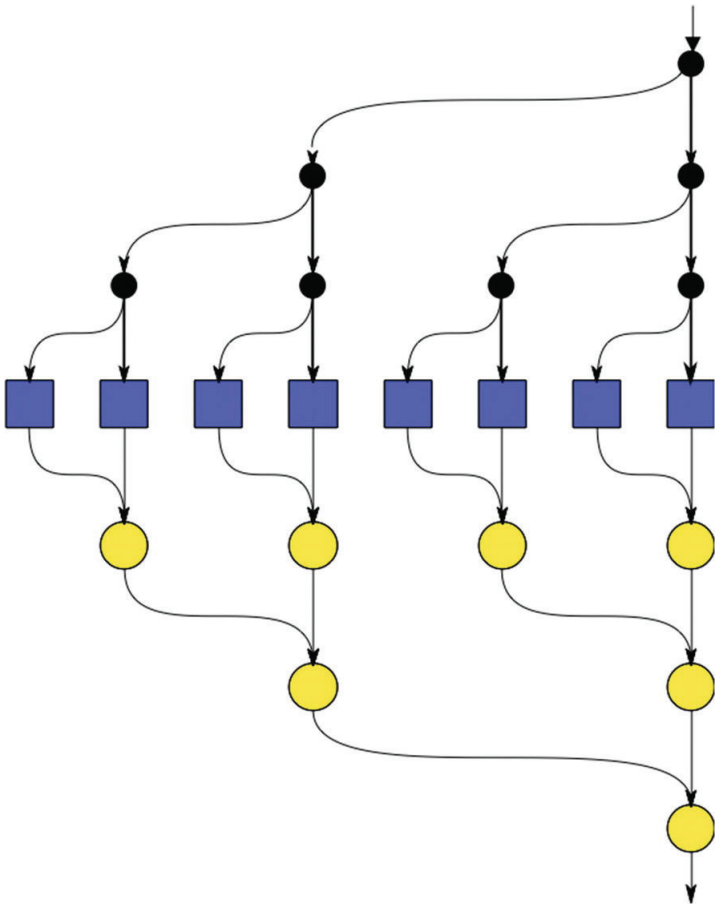


Рис. 8.8 ❖ Паттерн Разветвление–соединение:
поток управления разветвляется на несколько параллельных потоков,
которые впоследствии соединяются

Следует присмотреться к паттернам `parallel_for` и `parallel_reduce`, поскольку они автоматически реализуют то, что нужно, если только наши потребности не слишком нерегулярны. Вообще, для реализации паттерна Разветвление–соединение служат алгоритмы TBB `parallel_invoke` (глава 2), `task_group` (глава 10) и `flow_graph` (глава 3). Стоит также иметь в виду, что TBB устроена так, что преимущества разветвления–соединения и вложенности можно получить и без явного кодирования того и другого. В алгоритме `parallel_for` автоматически используется оптимизированная реализация разветвления–соединения, чтобы охватить имеющийся параллелизм, не жертвуя компонуемостью, т. е. так, чтобы одновременно можно было задействовать вложенность (в том числе вложенные циклы `parallel_for`) и другие формы параллелизма.

Паттерн Разделяй и властвуй

Паттерн Разветвление–соединение можно рассматривать как основной, а Разделяй и властвуй – как стратегию разделения и соединения. Считать ли его отдельным паттерном – вопрос терминологии, для наших целей это несущественно.

Паттерн Разделяй и властвуй применим, если задачу можно разбивать на меньшие подзадачи рекурсивно, пока не будет достигнут базовый случай, решаемый последовательно. Этот паттерн можно описать как разбиение задачи с последующим применением паттерна Отображение для решения каждой подзадачи. Результирующие решения подзадач объединяются в решение исходной задачи. Паттерн Разделяй и властвуй хорошо поддается распараллеливанию, потому что при разбиении легко получить выигрыш от наличия большего числа исполнителей (задач). Если этот паттерн напрашивается, то прежде всего следует обратить внимание на алгоритмы `parallel_for` и `parallel_reduce`. Кроме того, паттерн Разделяй и властвуй можно реализовать с помощью тех же алгоритмов, что используются для реализации паттерна Разветвление–соединение (`parallel_invoke`, `task_group` и `flow_graph`).

Паттерн Ветви и границы

Паттерн Разветвление–соединение можно рассматривать как основной, а Ветви и границы – как стратегию разделения и соединения. Считать ли его отдельным паттерном – вопрос терминологии, для наших целей это несущественно.

Паттерн Ветви и границы – *недетерминированный* метод поиска одного из нескольких удовлетворительных ответов. Слово «ветви» относится к конкурентности, а слово «границы» – к ограничению объема вычислений, например с помощью верхней границы (оставить наилучший из найденных к данному моменту результатов). Название «ветви и границы» проистекает из того факта, что мы рекурсивно разбиваем задачу на части, а затем ограничиваем решение в каждой части. В задачах искусственного интеллекта, например при вычислении ходов в шахматах и других играх, используются и другие родственные способы ограничения пространства решений, в т. ч. альфа-бета отсечение.

Метод ветвей и границ может давать суперлинейное ускорение, что нехарактерно для многих других параллельных алгоритмов. Но если возможных решений несколько, то этот паттерн недетерминирован – какое именно решение будет найдено, зависит от времени поиска в каждом подмножестве. Чтобы получить суперлинейное ускорение, необходимо эффективно реализовать отмену выполняющейся задачи (см. главу 15).

Задачи поиска вообще хорошо распараллеливаются, т. к. просматривать нужно много точек. Но поскольку исчерпывающий просмотр обходится слишком дорого, отдельные поиски следует каким-то образом координировать. Метод ветвей и границ предлагает хорошее решение. Вместо того чтобы исследовать все возможные точки в пространстве поиска, мы многократно разбиваем исходную задачу на меньшие части, вычисляем некоторые харак-

теристики уже завершенных подзадач, устанавливаем ограничения (границы) в соответствии с располагаемой информацией и исключаем подзадачи, не удовлетворяющие этим ограничениям. Такое исключение часто называют отсечением (pruning). Границы используются для исключения частей пространства поиска, т. е. областей, которые гарантированно не могут содержать оптимального решения. Применяя эту стратегию, мы постепенно сокращаем размер пространства потенциальных решений. И значит, для нахождения оптимального решения нам придется исследовать только малую часть всего множества входных данных.

Метод ветвей и границ недетерминирован и является отличным примером ситуации, в которой недетерминированность может быть полезна. Для выполнения параллельного поиска проще всего разбить множество и параллельно искать в каждом подмножестве. Рассмотрим случай, когда нужен только один результат и годятся любые данные, удовлетворяющие критерию поиска. Тогда, как только мы нашли элемент, отвечающий критерию поиска, в любом из параллельно просматриваемых подмножеств поиски в остальных подмножествах можно прекращать.

Метод ветвей и границ с некоторыми дополнениями полезен также в задачах математической оптимизации. В этом случае задана целевая функция, некоторые ограничения в виде уравнений и область определения. Функция имеет параметры. Область определения и уравнения ограничений определяют допустимые значения параметров. Цель оптимизации – найти такие значения параметров, при которых целевая функция принимает максимальное (или минимальное) значение в области определения.

Алгоритмы `parallel_for` и `parallel_reduce` реализуют те возможности, которые нужно рассмотреть в первую очередь при выборе паттерна Ветви и границы. Кроме того, он может быть реализован с помощью тех же алгоритмов, что паттерн Разветвление–соединение (`parallel_invoke`, `task_group` и `flow_graph`). При реализации следует хорошо понимать, как устроена отмена задач в TBB (глава 15).

Паттерн Конвейер

Паттерн Конвейер (рис. 8.9) легко недооценить. Возможности распараллеливания с помощью вложенности и организации конвейера огромны. Паттерн Конвейер устанавливает между задачами связь производитель–потребитель, организуя структурно неизменный поток данных.

Концептуально все этапы конвейера активны одновременно, и с каждым этапом может быть связано состояние, обновляемое по мере прохождения данных. Это открывает возможность для конвейерного распараллеливания. Кроме того, благодаря поддержке вложенности в TBB на каждом этапе может присутствовать собственный параллелизм. Базовые конвейеры поддерживают TBB-алгоритм `parallel_pipeline` (глава 2). В более общем случае множество этапов можно представить в виде ориентированного ациклического графа (сети). TBB-алгоритм `flow_graph` (глава 3) поддерживает как простые, так и обобщенные конвейеры.

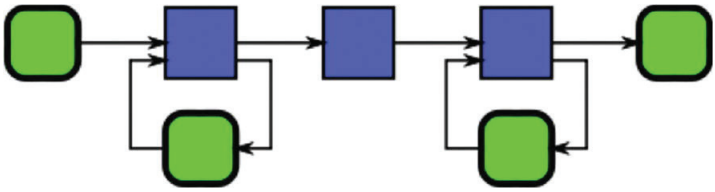


Рис. 8.9 ❖ Паттерн Конвейер:
между задачами имеется регулярная неизменяющаяся связь
производитель–потребитель

Паттерн Событийно-управляемая координация (реактивные потоки)

Паттерн Событийно-управляемая координация (рис. 8.10) устанавливает связь производитель–потребитель между задачами с нерегулярными, потенциально изменяющимися взаимодействиями. Обработка асинхронных действий – типичная проблема программирования.

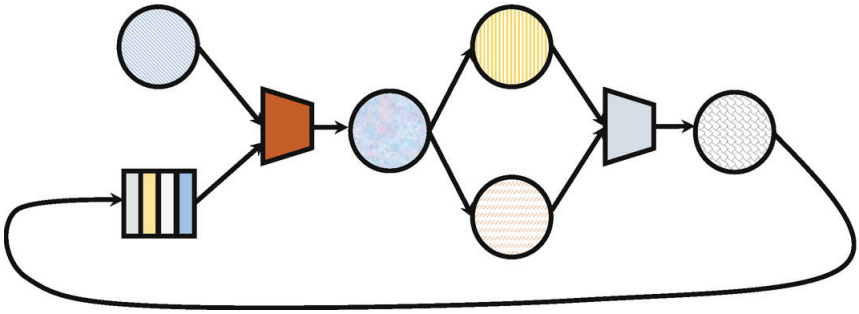


Рис. 8.10 ❖ Паттерн Событийно-управляемая координация:
между задачами имеется связь производитель–потребитель с нерегулярными,
потенциально изменяющимися взаимодействиями

Этот паттерн легко недооценить по тем же причинам, по которым многие недооценивают масштабируемость конвейера. Возможности распараллеливания с помощью вложенности и организации конвейера огромны.

Употребляя термин «событийно-управляемая координация», мы не стремимся акцентировать его отличия от «акторов», «реактивных потоков», «асинхронных потоков данных» или «событийно-ориентированной асинхронности».

Уникальные аспекты потока управления, необходимые для применения этого паттерна, привели к разработке алгоритма `flow_graph` (глава 3) в TBB.

Примерами асинхронных событий могут служить прерывания от работающих в режиме реального времени источников данных, например потока изображений или постов в Твиттере, а также действия пользователя в интерфейсе, например события мыши. Подробно о `flow_graph` написано в главе 3.

Резюме

TBB поощряет нас думать о паттернах, встречающихся в алгоритмах и приложениях, и сопоставлять их с предлагаемыми TBB средствами. TBB предлагает поддержку для паттернов, которые могут оказаться эффективными в масштабируемых приложениях, и при этом берет на себя заботу о деталях реализации, гарантируя, что результат будет модульным и компонуемым снизу доверху. «Суперпаттерн» вложенности очень хорошо поддержан в библиотеке TBB, и это дает ей возможность обеспечивать компонуемость, отсутствующую во многих других моделях параллельного программирования.

Дополнительная информация

TBB можно использовать и для реализации дополнительных паттернов, которые здесь не обсуждались. Мы акцентировали внимание на паттернах, которые считаем ключевыми, и их поддержке в TBB, но одна глава вряд ли может конкурировать с целыми книгами, посвященными исключительно паттернам.

В книге McCool, Robison, and Reinders «Structured Parallel Programming» (Elsevier, 2012) предлагается практическое обсуждение «паттернов, которые работают». Это книга для программистов, ищущих углубленное изложение паттернов с практическими примерами.

Книга Mattson, Sanders, and Massingill «Patterns for Parallel Programming» (Addison-Wesley, 2004) содержит гораздо более глубокое и академическое изложение паттернов, их таксономии и компонентов.

Часть II

Глава 9

Столпы компонуемости

В этой главе мы обсудим компонуемость: что это такое, благодаря каким характеристикам библиотека TBB является компонуемой и как использовать TBB и подобные ей библиотеки для создания масштабируемых приложений. Сам язык C++ допускает композицию, а TBB добавляет компонуемые средства параллелизма. Компонуемость TBB – весьма ценное свойство, поскольку позволяет пользоваться всеми преимуществами распараллеливания, не опасаясь перегрузить систему. Не задействуя параллелизм полностью, мы ограничиваем масштабируемость.

Называя TBB компонуемой параллельной библиотекой, мы имеем в виду, что разработчик может комбинировать написанный с помощью TBB код любым способом. Фрагменты кода могут быть последовательными, следующими один за другим, или вложенными, или конкурентными. Приложение может быть монолитным или разбросанным по разным библиотекам. Код может находиться в разных процессах, исполняемых одновременно.

Хотя это не сразу понятно, модели параллельного программирования в прошлом часто налагали ограничения, с которыми трудно было бороться при создании сложных приложений. Представьте, что было было, если бы не могли использовать предложения while внутри if, даже если они встречаются в вызываемых функциях. До появления TBB подобные ограничения существовали в некоторых моделях параллельного программирования, в частности OpenMP. Даже более современному стандарту OpenCL недостает полноценной компонуемости.

Самый обескураживающий аспект не допускающих композиции моделей параллельного программирования заключается в том, что можно запросить слишком много параллелизма. Это ужасно, и TBB сделала все, чтобы этого избежать. Наш опыт показывает, что наивные пользователи некомпонуемых моделей часто злоупотребляют параллелизмом, и тогда программа «падает» из-за нехватки памяти или начинает работать нестерпимо медленно из-за безумных накладных расходов на синхронизацию. Памятуя об этих проблемах, опытные программисты чрезмерно осторожничают с распараллеливанием, что приводит к несбалансированной нагрузке и плохой масштабируемости. Компонуемая модель программирования позволяет не думать об этом сложном поиске баланса.

Благодаря компонуемости библиотека TBB исключительно надежна как в простых, так и в сложных приложениях. Компонуемость – это философия проектирования, позволяющая создавать масштабируемые программы, не

опасаясь задействовать весь имеющийся параллелизм. В главе 1 мы представили идею трехуровневого пирога параллелизма, встречающуюся во многих приложениях. Повторяем ее на рис. 9.1.

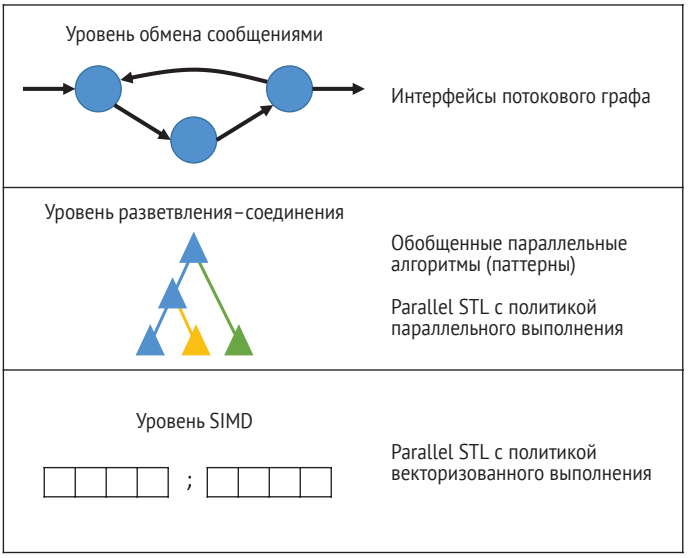


Рис. 9.1 ❖ Три уровня параллелизма, часто встречающихся в приложениях, и их отображение на высокоуровневые интерфейсы параллельного выполнения в TBB

Мы рассмотрели основы высокоуровневых интерфейсов, показанные на рис. 9.1, при обсуждении обобщенных параллельных алгоритмов в главе 2, потоковых графов в главе 3 и библиотеки Parallel STL в главе 4. Каждый из них играет важную роль в возведении этих уровней параллелизма. А поскольку все они реализованы с помощью задач TBB и допускают композицию, мы можем безопасно комбинировать их для построения сложных масштабируемых приложений.

Что такое компонуемость?

К сожалению, компонуемость нельзя назвать простым свойством программирования, которое либо есть, либо нет. Хотя в OpenMP имеются известные проблемы с вложенным параллелизмом, было бы неправильно называть OpenMP моделью программирования, вообще не допускающей композиции. Если приложение вызывает одну конструкцию OpenMP за другой, то такая последовательная композиция отлично работает. Также было бы преувеличением утверждать, что TBB – полностью компонуемая модель программирования, которая работает без нареканий со всеми другими моделями во всех ситуациях. Правильнее считать компонуемость мерой того, насколько хорошо работают две модели программирования при композиции определенными способами.

Например, рассмотрим две модели параллельного программирования: A и B. Обозначим T_A пропускную способность приложения, когда для выражения параллелизма на внешнем уровне используется модель A, и T_B пропускную способность того же приложения при использовании модели B (без использования A) для выражения параллелизма на внутреннем уровне. Если модели программирования компонуемы, то следует ожидать, что производительность ядра при использовании как внешнего, так и внутреннего параллелизма $T_{AB} >= \max(T_A, T_B)$. Насколько T_{AB} больше $\max(T_A, T_B)$ зависит от того, как эффективно модели компонуются друг с другом, и от физических свойств целевой платформы: количества ядер, объема памяти и т. д.

На рис. 9.2 показаны три общих типа композиции, используемой для комбинирования программных конструкций: вложенная, конкурентная и последовательное выполнение. Мы говорим, что TBB – компонуемая многопоточная библиотека, поскольку в случае, когда параллельный алгоритм, написанный с использованием TBB, компонуется с другими параллельными алгоритмами одним из трех способов, показанных на рис. 9.2, результирующий код демонстрирует хорошую производительность, т. е. $T_{TBB+другая} >= \max(T_{TBB}, T_{другая})$.

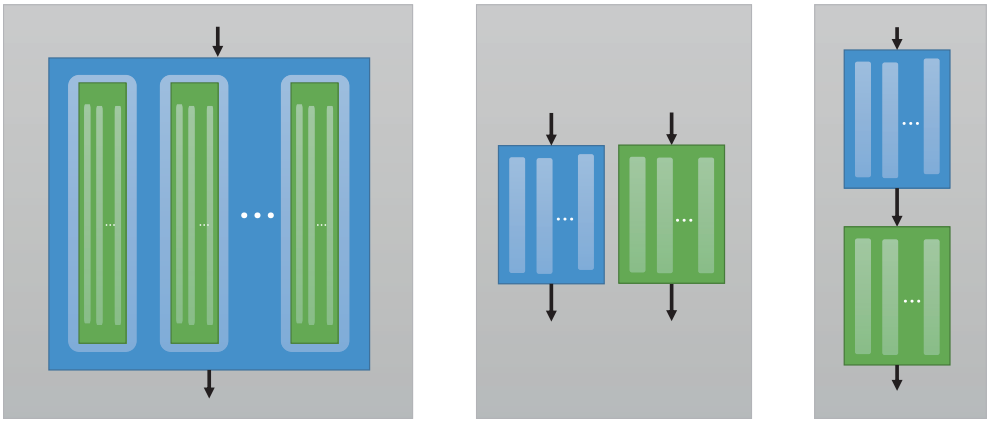


Рис. 9.2 ❖ Способы композиции программных конструкций

Прежде чем переходить к обсуждению свойств TBB, придающих ее хорошую компонуемость, рассмотрим каждый тип композиции: потенциальные проблемы и ожидаемое влияние на производительность.

Вложенная композиция

В случае *вложенной* композиции компьютер исполняет один параллельный алгоритм внутри другого, тоже параллельного. Назначение вложенной композиции почти всегда заключается в добавлении дополнительного параллелизма, иногда так даже удается экспоненциально увеличить объем параллельно выполняемой работы, как показано на рис. 9.3. Эффективная обработка вложенного параллелизма была главной целью при проектировании TBB.

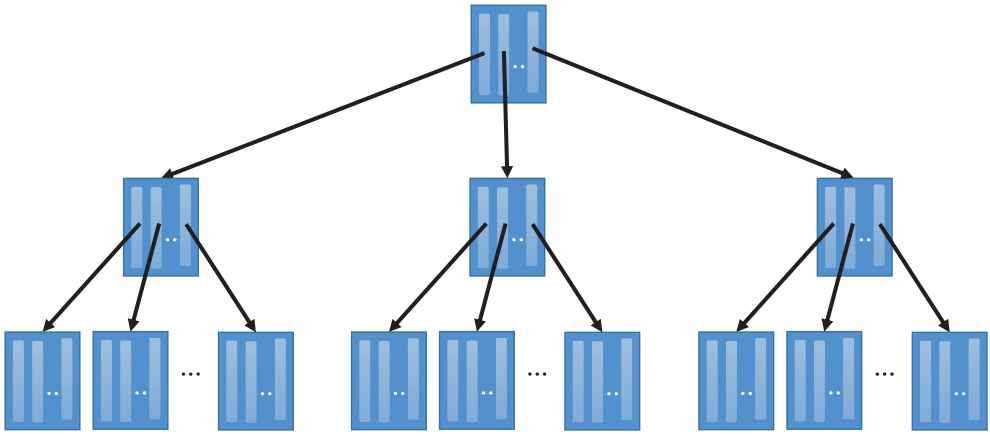


Рис. 9.3 ❖ Благодаря вложенному параллелизму количество доступных параллельных задач (или потоков, если речь идет о некомпонуемой библиотеке) может расти экспоненциально

Вообще, алгоритмы, предоставляемые библиотекой TBB, часто опираются на вложенный параллелизм. Например, в главе 2 мы обсуждали, как вложенные вызовы `parallel_invoke` можно использовать для создания масштабируемой параллельной версии быстрой сортировки. TBB сверху донизу спроектирована с учетом эффективного исполнения вложенного параллелизма.

В отличие от TBB, другие модели могут показывать катастрофически плохие результаты при наличии вложенного параллелизма. Конкретный пример – OpenMP API. OpenMP – широко распространенная модель параллельного программирования на основе разделяемой памяти, и если параллелизм одноуровневый, то она очень эффективна. В то же время она печально известна как непригодная для вложенного параллелизма, поскольку неотъемлемой частью ее определения является обязательность параллелизма. В приложениях с несколькими уровнями параллелизма каждая конструкция OpenMP создает дополнительную группу потоков. А каждому потоку нужно место в стеке и внимание со стороны планировщика потоков в ОС. Если количество потоков слишком велико, то приложению не хватит памяти. Если потоков больше, чем ядер, то несколько потоков вынуждены разделять одно ядро, и в таком случае выигрыш уменьшается, поскольку аппаратных ресурсов не хватает, и мы только несем накладные расходы.

Обычно самым практичным решением при работе с OpenMP является полное отключение вложенного параллелизма. В OpenMP API даже есть переменная среды `OMP_NESTED`, позволяющая включить или выключить вложенный параллелизм. Поскольку TBB обладает ослабленной последовательной семантикой и задачи в ней служат для выражения параллелизма, а не потоков, она способна гибко адаптировать степень параллелизма к располагаемым аппаратным ресурсам. В TBB отсутствует механизм отключения вложенного параллелизма – в нем просто нет необходимости!

Ниже в этой главе мы обсудим ключевые особенности TBB, благодаря которым она так эффективна при выполнении вложенного параллелизма, в т. ч.

пул потоков и планировщик с заимствованием работ. В главе 12 обсуждаются средства, позволяющие оказывать влияние на поведение TBB с целью изолирования задач и улучшения локальности данных.

Конкурентная композиция

Как показано на рис. 9.4, *конкурентная* композиция имеет место, когда выполнение параллельных алгоритмов совмещено во времени. Этот вид композиции можно использовать для намеренного увеличения степени параллелизма, но иногда он возникает случайно, если два не связанных между собой приложения (или конструкции в одной и той же программе) конкурентно выполняются на одной машине.

Конкурентное и параллельное выполнение – не всегда одно и то же! Как показано на рис. 9.4, *конкурентное выполнение* имеет место, когда несколько конструкций выполняются на одном и том же временном отрезке, а *параллельное* – когда несколько конструкций выполняются строго одновременно. Это означает, что параллельное выполнение – частный случай конкурентного, но не наоборот. Конкурентная композиция повышает производительность, только если ее можно эффективно превратить в параллельное выполнение.

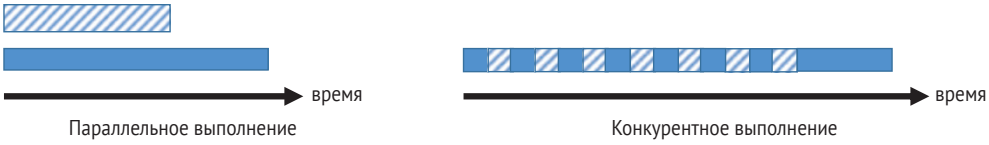


Рис. 9.4 ❖ Параллельное и конкурентное выполнение

Композиция двух циклов на рис. 9.5 будет конкурентной, если параллельная реализация цикла 1 выполняется конкурентно с параллельной реализацией цикла 2 независимо от того, работают они в разных процессах или в разных потоках одного процесса.

```
// цикл 1
for (int i = 0; i < N; ++i) {
  b[i] = f(a[i]);
}
```

```
// цикл 2
for (int i = 0; i < M; ++i) {
  d[i] = g(c[i]);
}
```

Рис. 9.5 ❖ Два цикла, выполняемых конкурентно

При конкурентном выполнении конструкций имеется некий арбитр (библиотека типа TBB, операционная система или комбинация того и другого), отвечающий за выделение ресурсов различным конструкциям. Если две конструкции нуждаются в доступе к одним и тем же ресурсам в одно и то же время, то доступ к ресурсам должен чередоваться.

Хорошая производительность конкурентной композиции могла бы означать, что физическое время выполнения не дольше времени выполнения самой длинной конструкции, поскольку все остальные конструкции выпол-

няются параллельно с ней (как при параллельном выполнении на рис. 9.4). Возможно и другое определение хорошей производительности – физическое время выполнения не больше суммы времен выполнения всех конструкций, если их выполнение должно чередоваться (как при конкурентном выполнении на рис. 9.4). Но идеальных систем не бывает, из-за деструктивной и конструктивной интерференции производительность никогда не отвечает приведенным выше определениям.

Во-первых, добавляются затраты на арбитраж. Например, если арбитром является планировщик потоков в ОС, то следует учитывать накладные расходы алгоритма планирования, накладные расходы на вытесняющую многозадачность, в частности на контекстное переключение, а также на механизмы обеспечения безопасности и изоляции. Если же арбитром является планировщик задач в пользовательской библиотеке типа ТВВ, то эти затраты ограничены отображением задач на потоки. Если порции работы очень мелкозернистые, то использование большого количества задач при небольшом количестве потоков обходится гораздо дешевле, с точки зрения накладных расходов, чем непосредственное использование многих потоков, хотя в конечном итоге задачи исполняются потоками.

Во-вторых, на производительность влияет конкурентное использование разделяемых системных ресурсов, например функциональных устройств, памяти и кешей данных. Перекрывающееся во времени выполнение конструкций может, к примеру, приводить к изменениям в производительности кешей – обычно количество непопаданий в кеш увеличивается, но в редких случаях конструктивной интерференции возможно даже уменьшение.

Пул потоков ТВВ и планировщик с заимствованием работ, обсуждаемые ниже в этой главе, помогают реализовать конкурентную композицию, уменьшить накладные расходы на арбитраж и зачастую ведут к такому распределению задач, при котором оптимизируется потребление ресурсов. Если подразумеваемое по умолчанию поведение ТВВ не устраивает, то можно воспользоваться средствами, описанными в главах 11–14, чтобы смягчить негативное влияние разделения ресурсов.

Последовательная композиция

Последний способ композиции двух конструкций – выполнить их последовательно, одну за другой, без совмещения во времени. Может показаться, что это тривиальный вид композиции, никак не влияющий на производительность, но (к сожалению) это не так. Используя последовательную композицию, мы, как правило, ожидаем, что при хорошей производительности между конструкциями не будет никакой интерференции.

Например, для циклов на рис. 9.6 последовательная композиция означает, что сначала выполняется цикл 3, а за ним цикл 4. Можно ожидать, что время завершения каждой параллельной конструкции при последовательном исполнении не отличается от времени исполнения этой конструкции по отдельности. Если время выполнения одного лишь цикла 3 после добавления параллелизма с применением модели программирования A равно $t_{3,A}$, а время выполнения одного лишь цикла 4 с применением модели программирования B равно $t_{4,B}$, то

мы ожидаем, что полное время последовательного выполнения конструкций не превышает суммы времен их выполнения, $t_{3,A} + t_{4,B}$.

```
// цикл 3
for (int i = 0; i < N; ++i) {
  b[i] = f(a[i]);
}

// цикл 3
for (int i = 0; i < N; ++i) {
  c[i] = f(b[i]);
}
```

Рис. 9.6 ❖ Два цикла, выполняемых один за другим

Однако, как и при конкурентном выполнении, имеются источники деструктивной и конструктивной интерференции, из-за которых фактическое время выполнения может отличаться от ожидаемого.

При последовательной композиции приложение должно перейти от одной параллельной конструкции к следующей. На рис. 9.7 показан идеальный и неидеальный переход при использовании одной и той же или разных моделей программирования. В обоих идеальных случаях нет никаких накладных расходов, и мы сразу переходим от одной конструкции к другой. Но на практике часто требуется некоторое время, чтобы освободить ресурсы после параллельного выполнения конструкции и подготовить ресурсы для выполнения следующей.

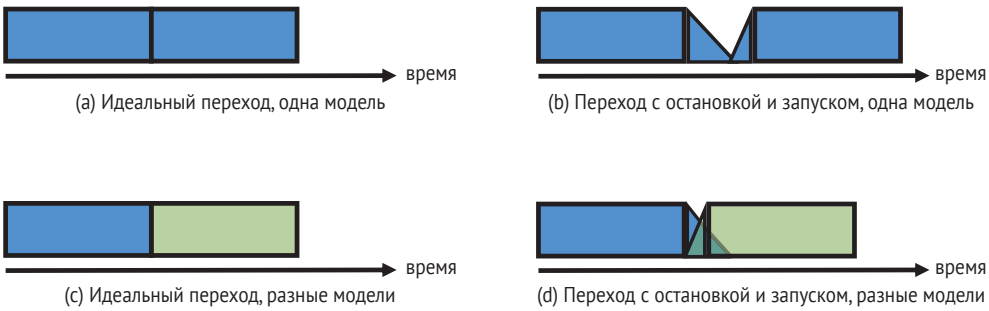


Рис. 9.7 ❖ Переход от выполнения одной конструкции к выполнению следующей

При использовании одной модели, как показано на рис. 9.7(b), библиотека может выполнить некоторые действия, чтобы остановить среду параллельного выполнения только для того, чтобы сразу же запустить ее снова. На рис. 9.7(d) мы видим, что если для выполнения конструкций используются разные модели, ничего не знающие друг о друге, то остановка первой конструкции может перекрываться с запуском и даже выполнением следующей, что вполне может привести к снижению производительности. Оба этих случая можно оптимизировать – и при проектировании TBB такие переходы учитывались.

Как и при любой композиции, на производительность может оказать влияние разделение ресурсов между обеими конструкциями. В отличие от вло-

женной или конкурентной композиции, конструкции не пользуются общими ресурсами одновременно или даже в режиме чередования, но все равно конечное состояние ресурсов после завершения одной конструкции может влиять на производительность следующей. Например, на рис. 9.6 мы видим, что цикл 3 пишет в массив `b`, а затем цикл 4 читает из массива `b`. Если соответственные итерации циклов 3 и 4 выполняются на одном и том же ядре, то увеличенная степень локальности данных может сократить количество непопаданий в кеш. Напротив, выполнение соответственных итераций на разных ядрах может без необходимости увеличить количество непопаданий.

Благодаря каким особенностям библиотека TBB является компонуемой

Библиотека TBB является компонуемой согласно замыслу. Когда она только появилась (свыше 10 лет назад), было понимание того, что параллельное программирование – удел всех, а не только разработчиков плоских монолитных приложений, поэтому проблемы компонуемости нужно решать с самого начала. Приложения, в которых используется TBB, часто модульные, в них применяются и другие библиотеки, которые сами могут содержать средства параллелизма. Эти сторонние параллельные алгоритмы могут специально или непреднамеренно компоноваться с алгоритмами на основе TBB. Кроме того, приложения обычно работают в многопрограммной среде, например на разделяемых серверах или на персональных ноутбуках, когда несколько процессов выполняются одновременно. Чтобы все разработчики могли пользоваться библиотекой эффективно, TBB должна организовать компонуемость хорошо. И она с этим справляется!

Хотя для создания масштабируемых параллельных приложений необязательно досконально разбираться в устройстве TBB, мы в этом разделе все же рассмотрим некоторые детали для любопытствующих читателей. Если вы и так верите, что TBB все делает правильно, и не очень интересуетесь тем, как это достигается, то можете спокойно пропустить данный раздел.

Пул потоков TBB (рынок) и арены задач

В основе компонуемости TBB лежат два механизма: *глобальный пул потоков* (*рынок*) и *арены задач*. На рис. 9.8 показано, как глобальный пул потоков и одна подразумеваемая по умолчанию арена задач взаимодействуют в приложении, имеющем один главный поток. Для простоты предполагается, что в целевой системе P = 4 логических ядра. На рис. 9.8(а) видно, что в приложении имеется один главный поток и глобальный пул, содержащий P – 1 рабочих потоков. В рабочих потоках исполняются диспетчеры (представленные сплошными прямоугольниками). В начальный момент все потоки в глобальном пуле спят, ожидая возможности принять участие в параллельной работе. На рис. 9.8(а) показано также, что по умолчанию создана одна арена задач. Каждому потоку, использующему TBB, выделяется его собственная арена задач, чтобы изоли-

ровать его работу от работы других потоков приложения. На рис. 9.8(a) арена всего одна, потому что в приложении имеется единственный поток. Когда поток приложения исполняет параллельный алгоритм TBB, диспетчер, связанный с этой ареной задач, работает, пока алгоритм не завершится. Ожидая завершения алгоритма, мастер-поток может участвовать в выполнении задач, запущенных на арене. На рисунке показано, как главный поток занимает слот, зарезервированный для мастер-потока.

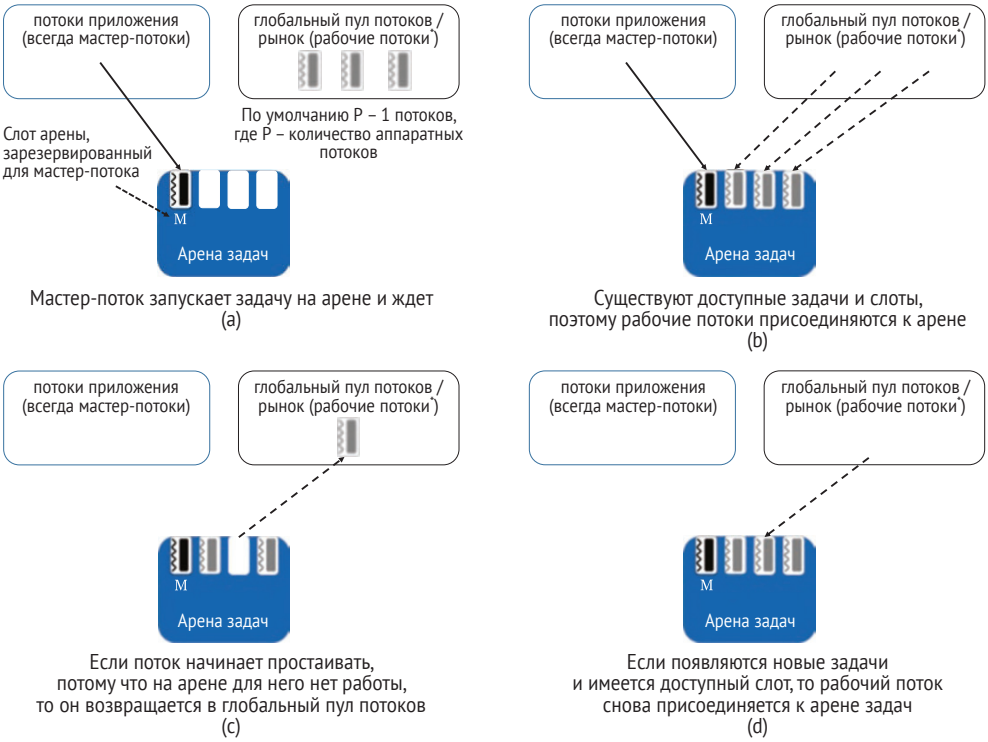


Рис. 9.8 ❖ Во многих приложениях существует единственный главный поток, и TBB по умолчанию создает P–1 рабочих потоков, участвующих в выполнении параллельных алгоритмов

Когда мастер-поток присоединяется к арене и впервые запускает задачу, рабочие потоки, спящие в глобальном пуле, просыпаются и отправляются на арену, как показано на рис. 9.8(b). Когда поток присоединяется к арене задач, занимая один из ее слотов, его диспетчер может принимать участие в выполнении задач, запускаемых другими потоками на этой арене, а также запускать задачи, которые могут видеть и заимствовать диспетчеры иных потоков, подключившихся к арене. На рис. 9.8 потоков как раз хватает, чтобы занять все слоты на арене задач, поскольку в глобальном пуле имеется P – 1 потоков и арена задач по умолчанию имеет достаточно слотов для P – 1 потоков. Обычно именно столько потоков нам и нужно, потому что главный поток плюс P – 1 рабочих занимают все имеющиеся процессорные ядра, не перегружая их. После того

как арена полностью занята, запуск новых задач не приводит к пробуждению дополнительных потоков, ожидающих в глобальном пуле.

На рис. 9.8(c) видно, что когда рабочий поток начинает простаивать и не может найти для себя работы на текущей арене задач, он возвращается в глобальный пул потоков. В этот момент рабочий поток мог бы присоединиться к другой арене задач, нуждающейся в «работягах», если таковая существует, но на рис. 9.8 арена всего одна, поэтому поток просто засыпает. Если впоследствии появятся новые задачи, то потоки, вернувшиеся в глобальный пул, снова проснутся и присоединятся к арене, чтобы помочь ей в выполнении дополнительной работы (рис. 9.8(d)).

Ситуация, изображенная на рис. 9.8, типична для приложений с одним главным потоком, когда не применялись продвинутые средства TBB для изменения параметров по умолчанию. В главах 11 и 12 мы обсудим средства, позволяющие создавать более сложные примеры, например такие, как на рис. 9.9. В этом сценарии имеется несколько потоков приложения и несколько арен задач. Если слотов на аренах задач больше, чем рабочих потоков, то рабочие потоки делятся пропорционально потребностям каждой арены. Так, арена, на которой свободных слотов вдвое больше, чем на другой арене, получит примерно вдвое больше рабочих потоков.

На рис. 9.9 демонстрируется еще несколько интересных особенностей арен задач. По умолчанию имеется только один слот, зарезервированный для мастер-потока, как на рис. 9.8. Но, как видно на примере двух правых арен на рис. 9.9, арена задач может быть создана (с помощью средств, которые мы обсудим в последующих главах) с несколькими слотами, зарезервированными для мастер-потоков, или вообще без таких слотов. Мастер-поток может занять любой слот, а потоки, попадающие на арену из глобального пула потоков, не могут занимать слоты, зарезервированные для мастеров.

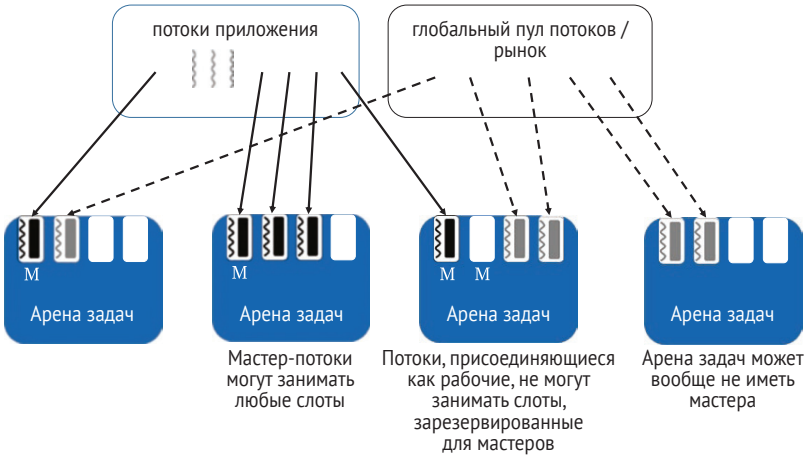


Рис. 9.9 ❖ Более сложное приложение с несколькими платформенными потоками и аренами задач

Но вне зависимости от сложности приложения всегда существует один глобальный пул потоков. На этапе инициализации TBB создает потоки, поме-

щаемые в этот пул. В главе 11 мы обсудим, как можно изменить размер пула при инициализации или даже динамически. Но именно ограниченность этого множества потоков и является причиной компонуемости TBB, поскольку предотвращает непреднамеренную перегруженность платформенных ядер.

Каждый поток приложения также неявно получает свою собственную арену задач. Поток не может заимствовать задачу у потока, находящегося на другой арене, и это изолирует работы, выполняемые разными потоками приложения (по умолчанию). В главе 12 мы обсудим, как поток приложения может изъявить желание присоединиться к другим аренам, – но по умолчанию у каждого арена своя.

TBB спроектирована так, чтобы написанные на ее основе приложения и алгоритмы хорошо компоновались любым способом – вложенным, конкурентным или последовательным. В случае вложенной композиции задачи TBB, генерируемые на всех уровнях, выполняются на одной и той же арене, занимая лишь ограниченное множество потоков, назначенных этой арене библиотекой. Это предотвращает экспоненциальный рост количества потоков. При конкурентной работе нескольких мастер-потоков рабочие потоки разделяются между аренами. А при последовательном выполнении рабочие потоки повторно используются конструкциями.

Хотя библиотека TBB не знает о решениях, принимаемых другими моделями параллельного программирования, ограничения на количество потоков, создаваемых ей в глобальном пуле, снижают нагрузку и на другие модели тоже. Мы еще вернемся к этому вопросу подробнее ниже в данной главе.

Диспетчер задач в TBB: заимствование работ, и не только

Стратегию планирования в TBB часто называют *заимствованием работ*. И это почти правда. Стратегия заимствования предназначена для работы в динамических средах и приложениях, когда задачи запускаются динамически, а приложение работает в многопрограммном окружении. Если работа распределяется с помощью заимствования, то рабочий поток активно ищет новую работу, когда переходит в состояние простоя, а не пассивно ждет, что ему будет назначена какая-то работа. Такой подход к распределению работ – оплата по факту потребления – эффективен, потому что не вынуждает потоки прекращать полезную работу, которой они заняты, только для того чтобы передать часть своей работы другим простаивающим потокам. При стратегии заимствования эти накладные расходы несут простаивающие потоки, а им все равно нечего делать! Планировщикам с заимствованием работ противоположны планировщики *с разделением работ*, которые назначают задачи рабочим потокам заранее, когда задачи впервые запускаются. В динамической среде, где задачи запускаются динамически, а некоторые аппаратные потоки могут быть загружены сильнее других, планировщики с заимствованием работ реагируют быстрее, что ведет к лучшей балансировке нагрузки и более высокой производительности.

В приложении TBB поток участвует в выполнении задач TBB, исполняя диспетчер задач, присоединенный к конкретной арене задач. На рис. 9.10 показано несколько важных структур данных, связанных с ареной задач и диспетчером.

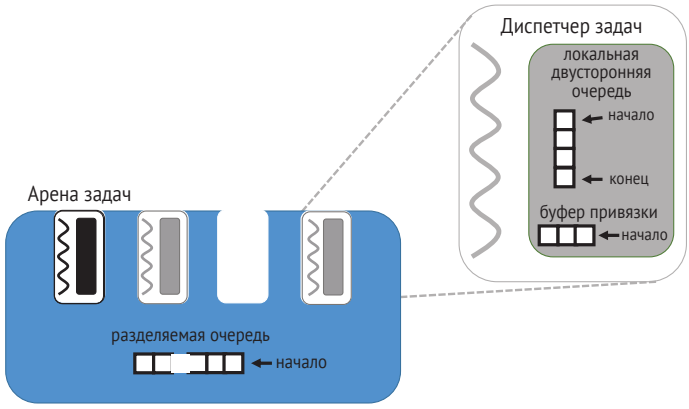


Рис. 9.10 ❖ Очереди на арене задач и в потоковых диспетчерах задач

Временно оставим в стороне разделяемую очередь на арене задач и буфер привязки в диспетчере задач, а сосредоточимся на локальной двусторонней очереди в диспетчере. Именно она используется в TBB для реализации стратегии планирования с заимствованием работ. Остальные структуры данных нужны для реализации расширений заимствования работ, мы вернемся к ним позже.

В главе 2 мы обсуждали разные виды циклов, реализованные с помощью обобщенных параллельных алгоритмов, включенных в библиотеку TBB. Многие из них опираются на понятие диапазона – рекурсивно разбиваемого множества значений, представляющего пространство итераций цикла. Алгоритмы рекурсивно разбивают диапазон цикла, применяя *задачи расщепления*, пока размер не станет пригодным для выполнения *задачи тела*. На рис. 9.11 показан пример распределения задач, реализующих паттерн цикла. Задача верхнего уровня t_0 представляет разбиение всего диапазона, который далее рекурсивно разбивается до уровня листьев, где к каждому поддиапазону применяется тело цикла. В распределении на рис. 9.11 каждый поток выполняет задачи цикла, охватывающие множество соседних итераций. Поскольку на близких итерациях часто производится доступ к близким данным, это распределение, скорее всего, оптимизирует локальность доступа. А поскольку потоки исполняют задачи, расположенные в изолированных частях дерева задач, то, получив начальный диапазон, поток может обрабатывать свою часть дерева, почти не взаимодействуя с другими потоками.

Циклические алгоритмы в TBB – примеры кеш-независимых алгоритмов. Так уж получилось, что кеш-независимые алгоритмы предназначены для эффективной оптимизации использования процессорных кешей данных – просто они делают это, ничего не зная о размерах кешей и их строк. Как и циклические алгоритмы в TBB, эти алгоритмы обычно реализованы с помощью метода «разделяй и властвуй», т. е. рекурсивно разбивают набор данных на все меньшие части, которые в конце концов помещаются в кеш независимо от его размера. В главе 16 мы рассмотрим кеш-независимые алгоритмы более подробно.

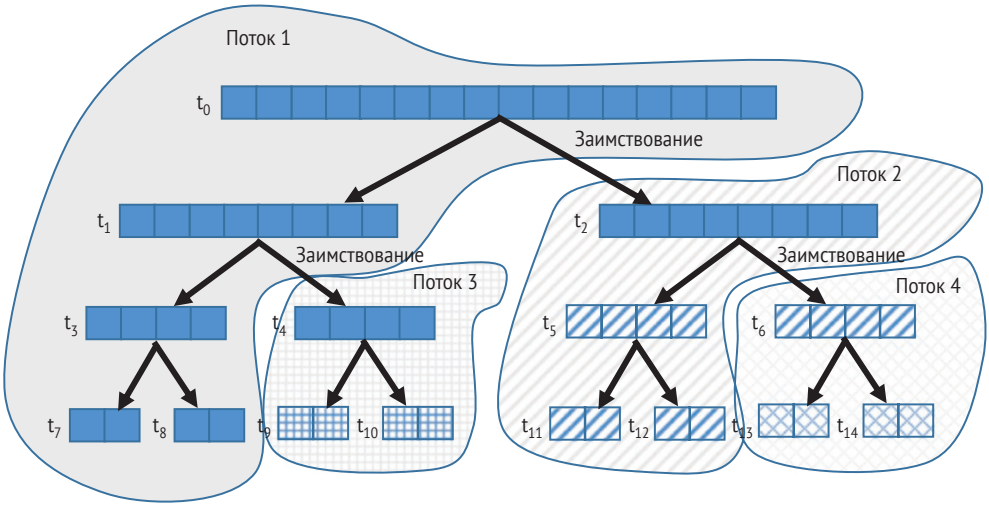


Рис. 9.11 ❖ Распределение задач, реализующих паттерн цикла

Диспетчеры задач пользуются своими двусторонними очередями для реализации стратегии планирования, оптимизированной для работы с кеш-независимыми алгоритмами, и создают такие распределения, как на рис. 9.11. Иногда эту стратегию называют политикой работы в глубину и заимствования в ширину (depth-first work, breadth-first steal policy). Всякий раз, как поток *запускает* новую задачу, т. е. делает ее доступной для своей арены задач, эта задача помещается в начало локальной двусторонней очереди его диспетчера задач. Затем, закончив текущую задачу, он пытается выбрать работу из начала своей двусторонней очереди, т. е. ту, которая была запущена последней (рис. 9.12). Но если в локальной очереди диспетчера задач не оказалось, то поток ищет нелокальную работу, случайно выбирая какой-нибудь другой поток на своей арене задач. Выбранный поток называется *жертвой*, потому что диспетчер планирует позаимствовать (украсть) у него задачу. Если локальная двусторонняя очередь жертвы не пуста, то диспетчер забирает задачу из конца этой очереди (см. рис. 9.12), т. е. ту, которая была запущена этим потоком раньше прочих.

На рис. 9.13 показан мгновенный снимок возможного распределения задач политикой планирования TBB при наличии всего двух потоков. Это упрощенное приближение к циклическому алгоритму TBB. Реализации алгоритмов в TBB хорошо оптимизированы и могут рекурсивно разбивать некоторые задачи, не запуская новых задач или в обход планировщика (как описано в главе 10). В примере на рис. 9.13 предполагается, что каждая задача расщепления и каждая задача тела запускаются на арене задач. Для оптимизированных алгоритмов это не так, но в целях иллюстрации подобное упрощение полезно.

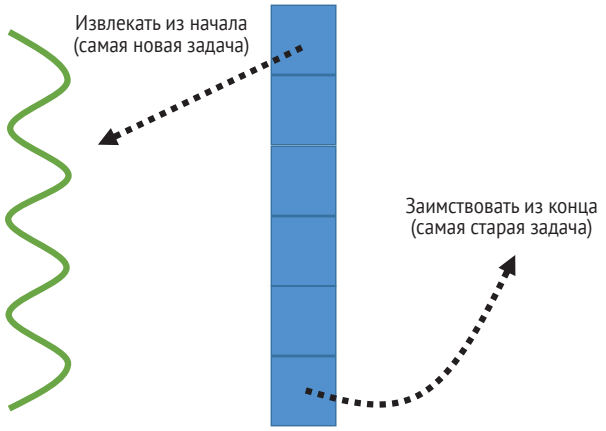
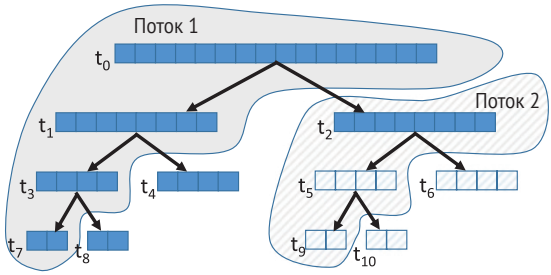
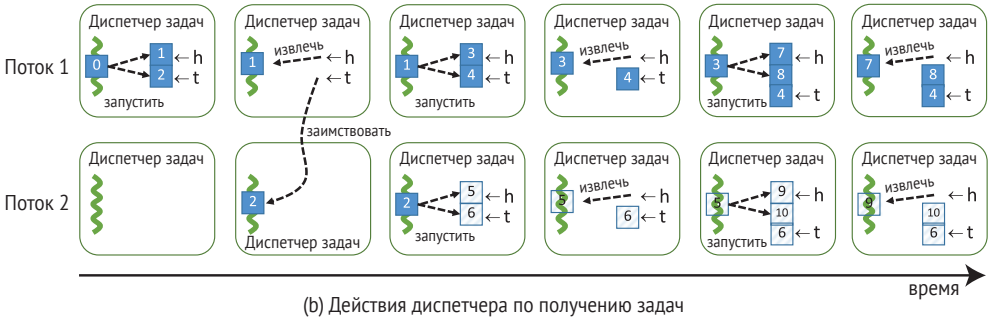


Рис. 9.12 ❖ Политика, применяемая диспетчером задач: извлечение локальных задач из начала локальной очереди и заимствование задач из конца очереди потока-жертвы



(a) Распределение задач между двумя потоками в результате заимствования работ



(b) Действия диспетчера по получению задач

Рис. 9.13 ❖ Мгновенный снимок распределения задач между двумя потоками и действия, которые оба диспетчера задач предпринимали для получения задач. Примечание: в текущей реализации циклических TBB-алгоритмов применяется обход планировщика и другие оптимизации, устраняющие часть запусков задач. Но все равно порядок выполнения и заимствования похож на то, что изображен на рисунке

На рис. 9.13 поток 1 начинает работу с корневой задачи и для начала разбивает диапазон на два больших куска. Затем он спускается вдоль одной стороны дерева задач, порождая новые задачи, пока не дойдет до листа, где тело применяется к конечному поддиапазону. Поток 2, который в начальный момент простаивает, заимствует работу из конца локальной двусторонней очереди потока 1, обеспечив себя вторым из двух больших кусков, созданных потоком 1 из исходного диапазона. На рис. 9.13(a) показан мгновенный снимок, на котором задачи t_4 и t_6 еще не выбраны ни одним потоком. Если бы в наличии было еще два потока, то вполне можно было бы получить распределение, показанное на рис. 9.11. В конце временного интервала, показанного на рис. 9.13(b), в локальных очередях потоков 1 и 2 все еще есть задачи. Когда каждый из них извлечет следующую задачу, он получит листья, соседние с только что завершенными задачами.

Глядя на рис. 9.11 и 9.13, не следует забывать, что показанное распределение – лишь одна из возможностей. Если на каждой итерации общий объем работы примерно одинаков и ни одно из ядер не перегружено, то, скорее всего, распределение будет равномерным. Но механизм заимствования работ означает, что если один из потоков выполняется на перегруженном ядре, то заимствовать он будет реже и, стало быть, получит меньше работы. Другие потоки заменят «выбывшего бойца». Модель программирования, поддерживающая только статическое равное назначение итераций ядрам, адаптироваться к такой ситуации не смогла бы.

Однако, как мы уже отмечали, диспетчеры задач TBB – не просто планировщики с заимствованием работ. На рис. 9.14 приведен упрощенный псевдокод цикла диспетчеризации внутри задачи. Мы видим строки с комментариями «выполнить задачу», «взять задачу, запущенную этим потоком» и «заимствовать задачу». В этих точках реализуется описанная выше стратегия заимствования задач, но в цикле имеются и другие действия.

В строке «обход планировщика» реализована оптимизация с целью избежать накладных расходов на планирование задачи. Если некоторая задача точно знает, какую задачу вызывающий поток будет исполнять следующей, то она может передать ей управление напрямую и тем самым устранить некоторые накладные расходы. Нам, пользователям TBB, делать это, скорее всего, не придется, но в главе 10 имеются дополнительные сведения по этому поводу. В хорошо оптимизированных алгоритмах и потоковых графах TBB не используются такие прямолинейные реализации, как на рис. 9.13. Для достижения максимальной производительности применяются различные оптимизации типа обхода планировщика.

В строке с комментарием «взять задачу с привязкой к этому пакету» мы смотрим, нет ли задачи в буфере привязки внутри диспетчера, и лишь потом пытаемся позаимствовать работу у случайно выбранной жертвы. Эта уловка позволяет реализовать привязку задач к потокам – механизм, подробно описанный в главе 13.

И наконец, строка «взять задачу из разделяемой очереди арены» служит для поддержки очередей задач, т. е. задач, которые попадают на арену, минуя

обычный механизм запуска. Такие задачи используются, когда работу нужно обслужить приблизительно в порядке «первым пришел – первым ушел» или по принципу «выстрелил и забыл», т. е. когда-нибудь выполнить ее надо, но частью структурированного алгоритма она не является. Постановка задач в очередь будет рассмотрена в главе 10.

```
if это мастер-поток
  t_next = первая задача
else if это рабочий поток
  t_next = задача, заимствованная из локальной двусторонней очереди случайного потока
end if
do
  do
    do
      do
        do
          do
            // выполнить задачу
            t = t_next
            t_next = t->execute()
          while t_next - действительная задача  // обход планировщика
          if я мастер, и мой алгоритм завершен
            return
          else if это был последний потомок родительской задачи
            t_next = t->parent
          end if
        while t_next - действительная задача
        // взять задачу, запущенную этим потоком
        t_next = извлечь из локальной двусторонней очереди
      while t_next - действительная задача
      // взять задачу с привязкой к этому потоку
      t_next = взять задачу из буфера привязки
    while t_next - действительная задача
    // взять задачу из разделяемой очереди арены
    t_next = извлечь из начала (приблизительного) разделяемой очереди
  while t_next - действительная задача
  // позаимствовать задачу
  t_next = позаимствовать задачу из начала локальной двусторонней очереди случайного потока
while t_next - действительная задача

// я рабочий поток, и больше делать нечего
вернуться в глобальный пул потоков
```

Рис. 9.14 ❖ Приблизительный псевдокод цикла диспетчеризации задачи TBB

Диспетчер TBB, показанный на рис. 9.14, – это невытесняющий диспетчер пользовательского уровня. Диспетчер потоков в ОС гораздо сложнее, поскольку должен иметь дело не только с алгоритмом планирования, но и с вытеснением, миграцией, изоляцией и безопасностью потоков.

Соберем все вместе

В предыдущем разделе описаны особенности дизайна TBB, благодаря которым алгоритмы и задачи допускают эффективную композицию различными способами. Выше мы также утверждали, что TBB хорошо работает и в комбинации с другими параллельными моделями. Теперь, вооружившись новыми знаниями, вернемся к типам композиции и убедимся, что модель TBB действительно компонуемая в силу своего дизайна.

В этом обсуждении мы сравним TBB с гипотетической некомпонуемой многопоточной библиотекой Non-Composable Runtime (NCR), которая включает параллельные конструкции, требующие принудительного параллелизма. Для каждой конструкции NCR нужна группа из P потоков, выделяемых только для выполнения этой конструкции, пока оно не завершится; никакие другие одновременно выполняющиеся или вложенные конструкции NCR использовать их не могут. Кроме того, NCR создает потоки при первом использовании конструкции, но не усыпляет их по завершении, а оставляет в состоянии активного ожидания, в котором они потребляют процессорное время, – чтобы как можно быстрее отреагировать на появление другой конструкции. Такое поведение не является диковинкой в других моделях программирования. В параллельных регионах OpenMP действительно используется принудительный параллелизм, что может стать причиной серьезных неприятностей, когда переменная среды `OMP_NESTED` установлена равной `true`. Библиотека Intel OpenMP предлагает также возможность оставить рабочие потоки в состоянии активного ожидания между регионами, для чего следует присвоить переменной среды `OMP_WAIT_POLICY` значение `active`. Справедливости ради следует отметить, что по умолчанию в Intel OpenMP предполагаются значения `OMP_NESTED=false` и `OMP_WAIT_POLICY=passive`, т. е. описанное некомпонуемое поведение умалчиваемым не является. Но для сравнения будем считать NCR образчиком некомпонуемой модели с исключительно мерзким поведением.

Теперь посмотрим, насколько хорошо TBB компонуется сама с собой и с NCR. В качестве меры производительности возьмем перегруженность, поскольку чем сильнее перегружена система (разность между количеством потоков и ядер), тем выше накладные расходы на планирование и деструктивное разделение. На рис. 9.15 показано, как устроена вложенность в обеих моделях. Когда TBB-алгоритм вкладывается в другой TBB-алгоритм, все порожденные задачи исполняются на одной и той же арене и разделяют все P потоков. А в случае NCR мы наблюдаем экспоненциальный рост количества потоков, поскольку для каждой вложенной конструкции необходима отдельная группа из P потоков, т. е. даже при двухуровневой вложенности потребуется P^2 потоков.

На рис. 9.16 показано, что происходит, когда модели комбинируются. Не важно, сколько потоков конкурентно исполняют TBB-алгоритмы – количество рабочих потоков TBB всегда ограничено величиной $P-1$! Таким образом, если TBB вложена в NCR, то используется не более $2P-1$ потоков: P потоков создает NCR, и все они играют роль мастер-потоков во вложенных TBB-алгоритмах, а еще $P-1$ являются рабочими потоками TBB. Но если конструкция NCR вло-

жена в TBB, то каждая задача TBB, исполняющая конструкцию NCR, должна будет собрать группу из P потоков. Одним из них может быть поток, исполняющий внешнюю задачу TBB, а остальные P – 1 придется создать или получить от библиотеки NCR. Таким образом, мы получаем P исполняемых параллельно потоков, созданных TBB, каждый из которых использует дополнительно P – 1 потоков, что в сумме дает примерно P^2 потоков. По рис. 9.15 и 9.16 видно, что даже когда TBB вложена в модель с плохим поведением, она ведет себя хорошо – в отличие от некомпонуемой модели типа NCR.

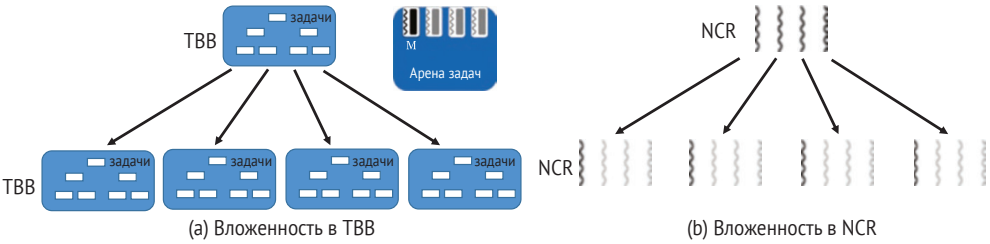


Рис. 9.15 ❖ Количество потоков при реализации вложенности в библиотеках TBB и NCR (без комбинирования их между собой)

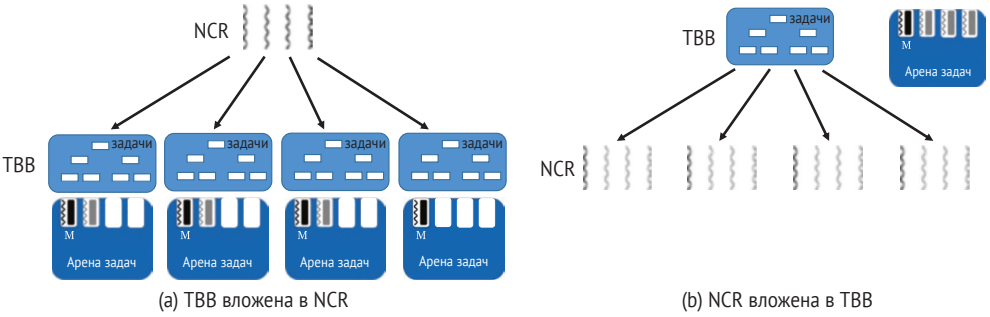


Рис. 9.16 ❖ Вложенность TBB и NCR друг в друга

Анализируя конкурентное выполнение, нужно рассматривать как однопроцессную конкурентность, когда параллельные алгоритмы выполняются разными потоками в одном процессе, так и многопроцессную конкурентность. В библиотеке TBB имеется единственный глобальный пул потоков на каждый процесс, но между процессами эти пулы не разделяются. На рис. 9.17 показано количество потоков при различных комбинациях конкурентного выполнения в однопроцессном случае. Когда TBB конкурентно выполняется сама с собой двумя потоками, каждый получает свою неявную арену задач, но обе арены совместно используют все P – 1 рабочих потоков; следовательно, общее количество потоков равно P + 1. В NCR используется группа P потоков на каждую конструкцию, т. е. всего 2P потоков. И аналогично, поскольку пулы потоков TBB и NCR не разделяются, всего при совместном конкурентном выполнении в одном процессе им потребуется 2P потоков.

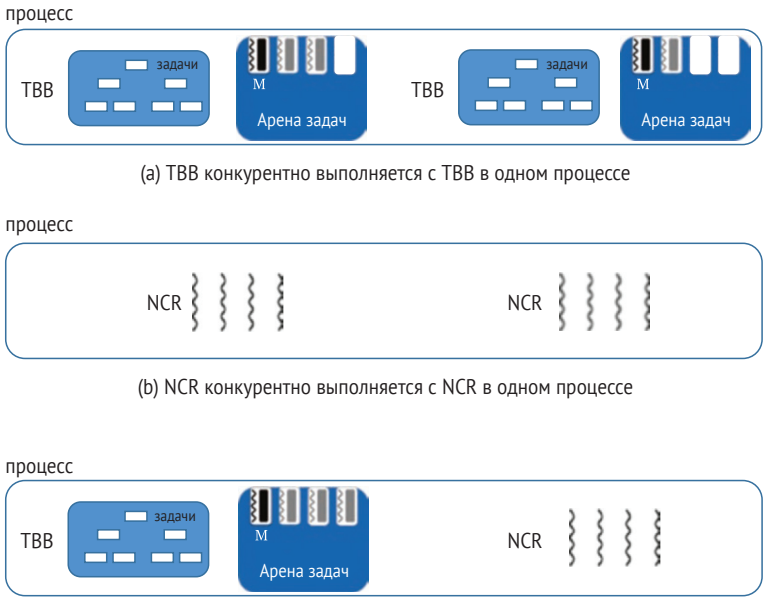
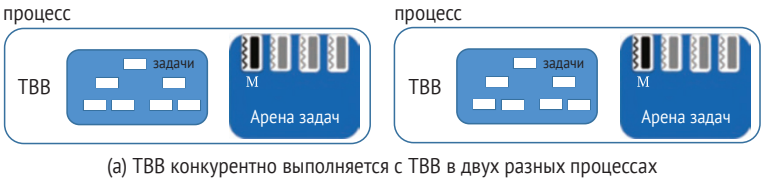


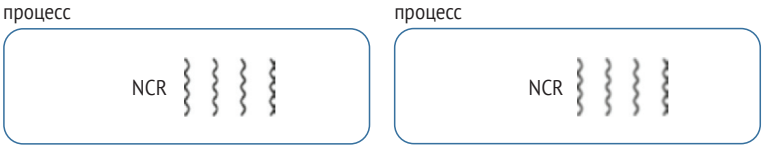
Рис. 9.17 ❖ Количество потоков, используемых при конкурентном выполнении TBB-алгоритмов и NCR-конструкций в одном процессе

На рис. 9.18 показано количество потоков при различных комбинациях конкурентного выполнения в многопроцессном случае. Поскольку TBB создает глобальный пул потоков в каждом процессе, она теряет преимущество над NCR. Во всех трех случаях используется 2P потоков.

Наконец, рассмотрим случай последовательной композиции, когда один алгоритм или конструкция выполняется вслед за другим. Как TBB, так и NCR хорошо компонуются последовательно в рамках одной библиотеки. Если задержка короткая, то потоки TBB все еще находятся на арене задач, потому что в течение очень непродолжительного времени после завершения очередной работы активно ищут следующую. Если же интервал между TBB-алгоритмами длинный, то рабочие потоки TBB вернутся в глобальный пул потоков и попадут на арену, когда станет доступна новая работа. Накладные расходы на такую миграцию очень малы, но пренебречь ими все же нельзя. Тем не менее негативные последствия крайне незначительны. Наша гипотетическая некомпонуемая библиотека (NCR) никогда не засыпает, т. е. всегда остается наготове для выполнения следующей конструкции, и величина задержки тут роли не играет. С точки зрения компонуемости более интересны случаи комбинирования NCR и TBB друг с другом (рис. 9.17). TBB быстро усыпляет свои потоки по завершении алгоритма, поэтому не оказывает негативного воздействия на следующую за ней конструкцию NCR. С другой стороны, гиперреактивная библиотека NCR держит свои потоки в активном состоянии, так что TBB-алгоритм, следующий за конструкцией NCR, будет вынужден бороться с ними за ресурсы процессора. Очевидно, что TBB более добропорядочна, потому что при ее проектировании учтена компонуемость с другими параллельными моделями.



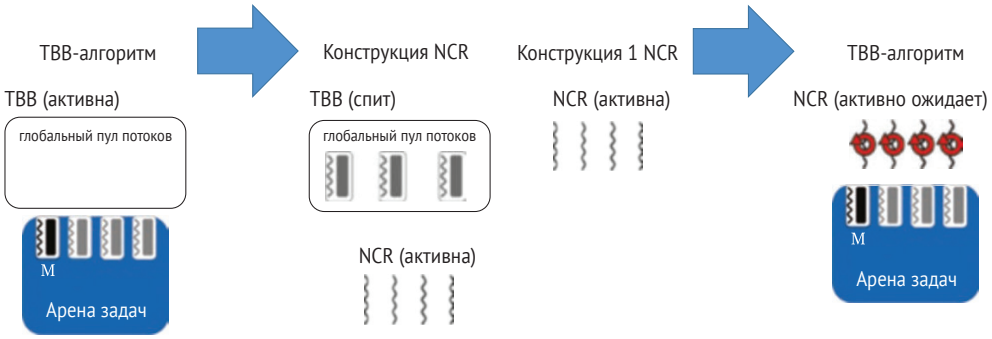
(a) TBB конкурентно выполняется с TBB в двух разных процессах



(b) NCR конкурентно выполняется с NCR в двух разных процессах

(c) NCR конкурентно выполняется с TBB в двух разных процессах

Рис. 9.18 ❖ Количество потоков, используемых при конкурентном выполнении TBB-алгоритмов и NCR-конструкций в двух разных процессах



(a) TBB, затем NCR

(a) NCR, затем TBB

Рис. 9.19 ❖ Количество потоков при последовательном выполнении TBB-алгоритмов и конструкций NCR

Рисунки с 9.15 по 9.19 показывают, что TBB хорошо компонуется сама с собой, а ее негативное воздействие на другие параллельные модели ограничено вследствие компонуемого дизайна. TBB-алгоритмы эффективно компонуются с другими TBB-алгоритмами, да и вообще являются добропорядочными членами общества.

Забегая вперед

В следующих главах мы рассмотрим ряд тем в развитие вопросов, поднятых в этой главе.

Управление количеством потоков

В главе 11 мы опишем, как использовать классы `task_scheduler_init`, `task_arena` и `global_control` для изменения количества потоков в глобальном пуле и управления числом слотов на аренах задач. Часто умолчаний, подразумеваемых TBB, вполне хватает, но при необходимости их можно изменить.

Изоляция работ

В этой главе мы видели, что каждый поток приложения по умолчанию получает свою собственную арену, позволяющую изолировать его от остальных потоков. В главе 12 мы обсудим функцию `this_task_arena::isolate`, которую можно использовать в нетипичных ситуациях, когда для корректности необходима изоляция работ. Мы также обсудим класс `task_arena`, применяемый для явного создания арен задач, которые можно использовать для изоляции работ ради повышения производительности.

Привязка задачи к потоку и потока к ядру

На рис. 9.10 мы видели, что в каждом диспетчере задач имеется не только локальная двусторонняя очередь, но и буфер привязки. На рис. 9.14 мы также видели, что когда у потока не остается работ в его собственной локальной очереди, он сначала проверяет этот буфер привязки, а только потом пытается заимствовать работу у другого потока. В главе 13 мы обсудим способы организовать привязку задачи к потоку и потока к ядру с помощью низкоуровневых средств, предлагаемых задачами TBB. В главе 16 мы обсудим диапазоны и разбиватели, которые используются в высокоуровневых TBB-алгоритмах для задействования локальности данных.

Приоритеты задач

В главе 14 мы обсудим приоритеты задач. По умолчанию диспетчер считает все задачи одинаково важными и просто пытается выполнить их как можно быстрее, не отдавая предпочтения ни одной. Однако TBB также позволяет назначить задаче низкий, средний или высокий приоритет. Мы обсудим, как этим воспользоваться и каковы последствия приоритетов для планирования.

Резюме

В этой главе мы подчеркнули важность компонуемости и отметили, что автоматически получаем ее, если в качестве модели параллельного программирования используем TBB. Мы начали с обсуждения различных способов композиции параллельных конструкций друг с другом и проблем, которые при этом возникают. Затем описали дизайн библиотеки TBB и причины, по которым он приводит к компонуемому параллелизму. И в заключение мы сравнили различные типы композиции в TBB с гипотетической некомпонуемой библиотекой NCR. Мы видели, что TBB отлично компонуется сама с собой и при этом хорошо себя ведет при комбинировании с другими моделями параллельного программирования.

Дополнительная информация

Cilk – параллельная модель и платформа, послужившая одним из образцов при проектировании первой версии планировщика TBB. Она содержит эффективную с точки зрения потребления памяти реализацию планировщика с заимствованием работ, описанную в статье

> *Robert D. Blumofe and Charles E. Leiserson.* Space-efficient scheduling of multithreaded computations. In Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC '93). ACM, New York, NY, USA, 362–371.

TBB предоставляет обобщенные алгоритмы, реализованные с помощью задач, исполняемых поверх потоков. Благодаря TBB разработчики могут использовать эти высокоуровневые алгоритмы вместо низкоуровневых потоков. Общее обсуждение вопроса о том, почему следует избегать непосредственного использования потоков в качестве модели программирования, см. в статье

> *Edward A. Lee.* The Problem with Threads // Computer, 39, 5 (May 2006), 33–42.

В этой главе мы использовали OpenMP API как образчик некомпонуемой модели. На самом деле OpenMP – очень эффективная модель программирования, у которой много пользователей, особенно в области высокопроизводительных вычислений. Дополнительные сведения об OpenMP можно почерпнуть на сайте www.openmp.org.

Глава 10

Использование задач для создания собственных алгоритмов

Среди вещей, которые нам больше всего нравятся в TBB, стоит отметить ее «многомасштабную» природу. В контексте моделей параллельного программирования многомасштабность означает, что при кодировании алгоритма можно выбирать различные уровни абстракции. В TBB имеются высокоуровневые шаблоны, как, например, `parallel_for` или `pipeline` (см. главу 2), которыми можно воспользоваться непосредственно, если алгоритм удовлетворяет определенным условиям. Но что, если алгоритм не настолько простой? Или если имеющаяся высокоуровневая абстракция не позволяет выжать из оборудования всю производительность до последней капли? Сдаться и остаться пленниками высокоуровневых средств модели программирования? Ни в коем случае! У нас должна быть возможность подобраться ближе к «железу», построить собственные шаблоны с нуля и тщательно оптимизировать свою реализацию, пользуясь низкоуровневыми средствами модели программирования, допускающими более тонкую настройку. И в TBB такая возможность есть. В этой главе мы займемся одним из самых мощных низкоуровневых средств TBB – интерфейсом программирования задач. Как уже много раз говорилось в этой книге, задачи – плоть и кровь TBB, это строительные блоки, из которых собираются такие высокоуровневые шаблоны, как `parallel_for` и `pipeline`. Но ничто не мешает нам отправиться в эти опасные земли и начать кодировать алгоритмы непосредственно на уровне задач – надстроить над ними собственные высокоуровневые шаблоны или, как будет описано в следующих главах, оптимизировать реализацию, точно настроив способ выполнения задач. Именно этому вы научитесь, прочитав эту и последующие главы. Удачного путешествия!

Сквозной пример: вычисление последовательности

Реализации, основанные на задачах TBB, особенно подходят для алгоритмов, в которых проблему можно рекурсивно разбить на меньшие части, построив древовидную декомпозицию. Таких проблем полно. Параллельные паттерны

Разделяй и властвуй и Ветви и границы – примеры классов таких алгоритмов. Если проблема достаточно велика, то обычно она хорошо масштабируется для параллельной архитектуры, поскольку ее легко разбить на достаточное количество задач, чтобы полностью загрузить оборудование, избежав при этом несбалансированной нагрузки.

Для этой главы мы выбрали одну из простейших задач, допускающих реализацию на основе такого древовидного подхода. Это вычисление последовательности чисел Фибоначчи, первые два члена которой – 0 и 1, а каждый последующий равен сумме двух предыдущих:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

Математически n-й член последовательности F_n определяется рекуррентной формулой

$$F_n = F_{n-1} + F_{n-2},$$

и $F_0 = 0, F_1 = 1$. Есть несколько алгоритмов вычисления F_n, но для иллюстрации того, как работают задачи TBB, мы выбрали показанный на рис. 10.1, хотя он не самый эффективный.

```
long fib(long n) {
  if(n<2)
    return n;
  else
    return fib(n-1)+fib(n-2);
}
```

Рис. 10.1 ❖ Рекурсивное вычисление F_n

Вычисление чисел Фибоначчи – классический пример рекурсии в информатике, но это также классический пример неэффективности простого алгоритма. Эффективнее было бы вычислить матрицу

$$F_n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

и взять ее левый верхний элемент. Для быстрого возведения матрицы в степень нужно последовательно возводить ее в квадрат. Но в этом разделе мы продолжим рассматривать классический пример рекурсии.

Код на рис. 10.1 четко следует рекуррентной формуле $F_n = F_{n-1} + F_{n-2}$. Он и так понятен, но для пущей наглядности мы изобразили дерево рекурсивных вызовов, получающееся при вызове `fib(4)`.

Строка `if (n<2)` в начале последовательного кода на рис. 10.1 называется *базой рекурсии*, она необходима в любой рекурсивной программе, чтобы предотвратить бесконечную рекурсию. Мы же не хотим исчерпать стек, правда?

Эту первую последовательную реализацию мы распараллелим, применяя различные подходы на основе задач – от самого простого до более изощ-

ренного и оптимизированного. Полученные уроки можно будет применить и к другим древовидным или рекурсивным алгоритмам, а представленные оптимизации – использовать для максимально эффективного использования параллельной архитектуры в похожих ситуациях.

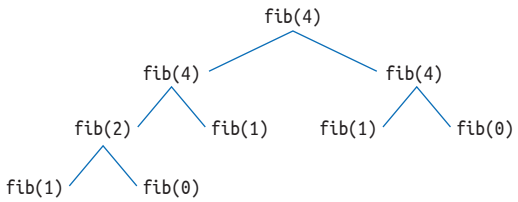


Рис. 10.2 ❖ Дерево рекурсивных вызовов при вычислении fib(4)

Высокоуровневый подход: parallel_invoke

В главе 2 мы уже познакомились с высокоуровневым алгоритмом, которым можно воспользоваться для запуска параллельных задач: `parallel_invoke`. Опираясь на него, мы можем написать первую параллельную реализацию алгоритма для вычисления чисел Фибоначчи (рис. 10.3).

```
#include <tbb/parallel_invoke.h>
long parallel_fib(long n) {
  if(n<2) {
    return n;
  }
  else {
    long x, y;
    tbb::parallel_invoke([&]{x=parallel_fib(n-1);},
                         [&]{y=parallel_fib(n-2);});
    return x+y;
  }
}
```

Рис. 10.3 ❖ Параллельная реализация чисел Фибоначчи с помощью `parallel_invoke`

Функция `parallel_invoke` рекурсивно запускает `parallel_fib(n-1)` и `parallel_fib(n-2)` и возвращает результат в локальных переменных `x` и `y`, которые автоматически захватываются по ссылке двумя лямбда-выражениями. Когда эти задачи завершаются, вызывающая задача просто возвращает сумму `x+y`. Благодаря рекурсивной природе эта реализация продолжает создавать параллельные задачи, пока не дойдет до базового случая `n<2`. Это означает, что TBB создаст задачи даже для вычисления `parallel_fib(1)` и `parallel_fib(0)`, которые просто вернут 1 и 0 соответственно. Как уже не раз было сказано, мы хотим раскрыть как можно больше параллелизма для создания достаточно большого

количества задач, но в то же время поддержать минимальный уровень зернистости задач (> 1 мкс, см. далее главы 16 и 17), чтобы накладные расходы на создание задачи окупились. Обычно для реализации этого компромисса заводят какой-нибудь параметр «отсечения», как показано на рис. 10.4.

```
long parallel_fib(long n) {
  if(n<cutoff) {
    return fib(n);
  }
  else {
    long x, y;
    tbb::parallel_invoke([&]{x=parallel_fib(n-1);},
                         [&]{y=parallel_fib(n-2);});
    return x+y;
  }
}
```

Рис. 10.4 ❖ Реализация с помощью `parallel_invoke` с параметром отсечения

Идея в том, чтобы модифицировать базовый случай: не создавать новые задачи, когда `n` недостаточно велико (`n<cutoff`), а прибегнуть к последовательному выполнению. Для определения оптимального параметра отсечения нужно поэкспериментировать, поэтому рекомендуется писать код, так чтобы эта величина была входным параметром. Например, в наших тестах для вычисления `fib(30)` требуется около 1 мс, т. е. это достаточно мелкозернистая задача, и дальнейшее разбиение можно прекратить. При `cutoff=30` мы будем вызывать последовательную версию для задач, получающих на входе `n=29` и `n=28`, как показано на рис. 10.5.

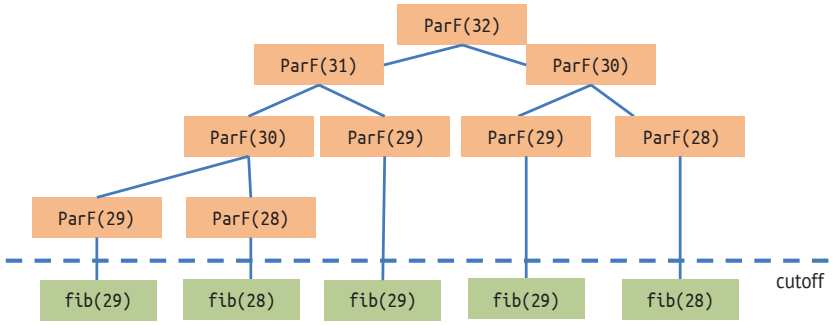


Рис. 10.5 ❖ Дерево вызовов для вычисления `parallel_fib(32)` – для экономии места на рисунке этот вызов обозначен `ParF(32)`, – в базовом случае вычисление `fib()` реализовано последовательно

Если, взглянув на рис. 10.5, вы пришли к выводу, что глупо вычислять `fib(29)` в трех разных задачах и `fib(28)` еще в двух, то вы абсолютно правы – глупее

не придумаешь! В свое оправдание напомним: мы сразу предупредили, что эта реализация не оптимальна, а используется в качестве примера рекурсии в педагогических целях. Очевидная оптимизация – организовать рекурсию, так чтобы уже вычисленные числа Фибоначчи заново не вычислялись. Тогда мы получим оптимальную сложность $O(\log n)$, но сейчас наша цель не в этом.

Глядя на рис. 10.4, вы, наверное, недоумеваете, с какой стати мы вернулись к алгоритму `parallel_invoke`, который уже изучили в главе 2. Да добрались ли мы до второй, более продвинутой части книги? Добрались! Э-э-э... но где же тогда продвинутые средства, где обещанные низкоуровневые рычаги и столь любимые нами возможности для оптимизации??? Спокойно, начинаем погружение!!!

Высший среди низших: task_group

Если есть возможность обойтись без рычагов и средств оптимизации, которые будут представлены ниже, то к нашим услугам класс `task_group`. Это, если хотите, абстракция среднего уровня, которой удобно пользоваться. На рис. 10.6 показана другая реализация чисел Фибоначчи, основанная на `task_group`.

```
#include <tbb/task_group.h>
long parallel_fib(long n) {
  if(n<cutoff) {
    return fib(n);
  }
  else {
    long x, y;
    tbb::task_group g;
    g.run([&]{x=parallel_fib(n-1);}); // запустить задачу
    g.run([&]{y=parallel_fib(n-2);}); // запустить другую задачу
    g.wait();                 // ждать завершения обеих задач
    return x+y;
  }
}
```

Рис. 10.6 ❖ Параллельное вычисление чисел Фибоначчи с помощью класса `task_group`

На первый взгляд, это просто более пространный способ реализации кода на рис. 10.4. Но мы хотим подчеркнуть, что, в отличие от версии с `parallel_invoke`, здесь мы имеем дело с группой задач, `g`, и, как будет показано ниже, это открывает дополнительные возможности по отмене задач. Кроме того, явно вызывая функции-члены `g.run()` и `g.wait()`, мы запускаем новые задачи и ждем их завершения в двух разных точках программы, тогда как функция `parallel_invoke` устанавливает неявный барьер после запуска задачи. Среди прочего такое отделение `run()` от `wait()` позволяет вызывающему потоку выполнить какие-то вычисления между запуском задачи и ожиданием ее завершения в блокирую-

щем вызове `wait()`. Этот класс предоставляет и другие интересные функции-члены, которые могут пригодиться в определенных ситуациях:

- `void run_and_wait( const Func& f )` – эквивалент `{run(f); wait();}`, но с гарантией, что `f` исполняется в текущем потоке. Ниже (в разделе «Низкоуровневый интерфейс задач: часть вторая – продолжение задачи») мы увидим, что это удобный способ обойти планировщик TBB. Если мы сначала вызовем `run(f)`, то по существу запустим задачу, которая будет помещена в локальную очередь рабочего потока. Вызывая `wait()`, мы обращаемся к планировщику, который извлекает из очереди только что помещенную туда задачу, если никто не позаимствовал ее в промежутке. У `run_and_wait` двоякая цель: во-первых, избежать накладных расходов на постановку в очередь, планирование и извлечение из очереди, а во-вторых, избежать потенциального заимствования, которое может иметь место, пока задача находится в очереди;
- `void cancel()` – отмена всех задач в данной группе. Быть может, вычисление было инициировано из пользовательского интерфейса, в котором имеется также кнопка «Отмена». Теперь у нас есть возможность прервать вычисление, если пользователь нажмет эту кнопку. В главе 15 мы еще вернемся к вопросу об отмене и обработке исключений;
- `task_group_status wait()` – вернуть состояние группы задачи, которое может принимать следующие значения: `complete` (все задачи в группе завершились), `canceled` (`task_group` получила запрос на отмену), `not_completed` (не все задачи в группе завершились).

Заметим, что в параллельной реализации на рис. 10.6 каждое обращение к `parallel_fib` создает новый объект `task_group`, так что мы можем отменить одну ветвь, не трогая остальных. Это будет продемонстрировано в главе 15. У создания единственной группы задач есть и еще один недостаток: если в одной группе слишком много задач, то это приведет к сериализации создания задач и утрате масштабируемости. Рассмотрим такой пример:

```
tbb::task_group g;
for (int i=0; i < n; i++) g.run([]{foo();});
g.wait();
```

Как видим, `n` задач запускаются одна за другой в одном и том же потоке. Другие рабочие потоки будут вынуждены заимствовать каждую задачу, созданную той, что исполняет `g.run()`. Это без сомнения резко снизит производительность, особенно если `foo()` – мелкозернистая задача, а количество рабочих потоков `n` велико. Рекомендуемая альтернатива – рекурсивное создание задач – показана на рис. 10.6. При таком подходе рабочие потоки заимствуют работы в начале вычисления, и в идеале через $\log_2(n)$ шагов все `n` рабочих потоков будут заниматься своими задачами, которые будут помещать дополнительные задачи в локальные очереди своего потока. Например, при `n=4` первый поток, `A`, запускает две задачи и начинает исполнять одну из них, тогда как поток `B` заимствует другую. Далее потоки `A` и `B` запускают по две задачи (всего четыре) и начинают исполнять две из них, а остальные две заимствуются потоками `C` и `D`. Начиная с этого момента все четыре потока работают и помещают допол-

нительные задачи в свои локальные очереди, а к заимствованию прибегают, только если локальные задачи кончились.

Осторожно! Под свою ответственность: низкоуровневый интерфейс задач

В классе task много разных возможностей, а значит, и много способов допустить ошибку. Если требуемый параллельный паттерн широко распространен, то наверняка имеется готовый высокоуровневый шаблон, реализованный и оптимизированный квалифицированными разработчиками на основе интерфейса задач. В большинстве случаев рекомендуется выбирать высокоуровневый алгоритм. Поэтому оставшаяся часть этой главы служит двум целям. Во-первых, дать вам средства для разработки собственного основанного на задачах параллельного алгоритма или высокоуровневого шаблона, если уже имеющихся в TBB недостаточно. Во-вторых, раскрыть низкоуровневые детали механизмов TBB, чтобы вы понимали некоторые оптимизации и приемы, упоминаемые в последующих главах. Например, ниже мы еще вернемся к этой главе, когда будем объяснять, как parallel_pipeline и потоковый граф могут лучше задействовать локальность благодаря технике обхода планировщика. А здесь мы опишем, как эта техника работает и почему может дать выигрыш.

Низкоуровневый интерфейс: часть первая – блокировка задач

В классе TBB task немало средств и рычагов управления для точной настройки поведения программы, основанной на задачах. Медленно, но верно мы познакомимся с различными функциями-членами, постепенно увеличивая сложность программы вычисления чисел Фибоначчи. Для начала на рис. 10.7 и 10.8 показана реализация с помощью низкоуровневых задач. Это точка отсчета, которую мы будем оптимизировать в последующих версиях.

```
#include <tbb/task.h>

long parallel_fib(long n) {
  long sum;
  FibTask& a = *new(tbb::task::allocate_root()) FibTask{n,&sum};
  tbb::task::spawn_root_and_wait(a);
  return sum;
}
```

Рис 10.7 ❖ Реализация parallel_fib с помощью класса task

Код на рис. 10.7 состоит из следующих шагов.

1. Выделить память для задачи. Память должна выделяться специальными функциями-членами, чтобы ее можно было эффективно использовать повторно по завершении задачи. Для этой цели предназначены перегруженный оператор new и функция-член task::allocate_root. Суффикс _root

в имени означает, что у созданной задачи нет родителя. Это корень дерева задач.

2. Сконструировать задачу с помощью конструктора `FibTask{n,&sum}` (определение класса приведено на рисунке ниже), вызываемого оператором `new`. Когда задача начинает исполняться (шаг 3), она вычисляет `n`-е число Фибоначчи и сохраняет его в члене `sum`.
3. Выполнить задачу до конца с помощью функции `task::spawn_root_and_wait`.

```
class FibTask: public tbb::task {
public:
  long const n;
  long* const sum;
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // Переопределяет виртуальную функцию task::execute
    if(n<cutoff) {
      *sum = fib(n);
    }
    else {
      long x, y;
      FibTask& a = *new(tbb::task::allocate_child()) FibTask{n-1,&x};
      FibTask& b = *new(tbb::task::allocate_child()) FibTask{n-2,&y};
      // Установить ref_count в состояние «два потомка плюс один для ожидания»
      tbb::task::set_ref_count(3);
      // Запустить b
      tbb::task::spawn(b);
      // Запустить a и ждать всех потомков (a и b)
      tbb::task::spawn_and_wait_for_all(a);
      // Вычислить сумму
      *sum = x+y;
    }
    return nullptr;
  }
};
```

Рис 10.8 ❖ Определение класса `FibTask`, использованного на рис. 10.7

Собственно работа выполняется в классе `FibTask`, определенном на рис. 10.8. По сравнению с функцией `fib` и двумя предыдущими реализациями `parallel_fib`, это довольный длинный код. Но мы предупреждали – это низкоуровневый код, поэтому писать его не так удобно и быстро, как с помощью высокоуровневых абстракций. Но чтобы компенсировать дополнительное бремя, мы далее покажем, что этот класс позволяет вволю покопаться под капотом и настроить поведение и производительность, как нам будет угодно.

Как и все задачи, планируемые TBB, `FibTask` наследует классу `tbb::task`. В полях `n` и `sum` хранятся соответственно входное значение и указатель на результат.

Они инициализируются аргументами, переданными конструктору `FibTask(long n_, long *sum_)`.

Вычисление производит функция-член `execute`. Любая задача должна предоставить функцию `execute`, переопределяющую чисто виртуальную функцию-член `tbb::task::execute`. Эта функция должна выполнить работу задачи и вернуть либо `nullptr`, либо указатель на следующую подлежащую выполнению задачу, как было показано на рис. 9.14. В этом простом примере возвращается `nullptr`.

Вот что делает функция-член `FibTask::execute()`:

1) проверить условие `n<cutoff`, и если оно выполнено, то вызвать последовательную версию;
2) в противном случае перейти к ветви `else`. Создать и запустить две дочерние задачи, которые вычисляют соответственно F_{n-1} и F_{n-2}. Унаследованная функция-член `allocate_child()` выделяет память для задачи. Напомним, что верхнеуровневая функция `parallel_fib` вызвала `allocate_root()`, чтобы выделить память для корневой задачи. Разница в том, что теперь создаются дочерние задачи. Эта связь подчеркивается выбором метода выделения памяти. Все методы выделения перечислены в приложении В на рис. В.76;
3) вызвать `set_ref_count(3)`. Число 3 говорит о том, что есть два потомка и дополнительная неявная ссылка, необходимая функции-члену `spawn_and_wait_for_all`. Функция `set_ref_count` инициализирует атрибут `ref_count` задачи TBB. Всякий раз, как очередной потомок завершает вычисление, он уменьшает на 1 атрибут `ref_count` своего родителя. Не забывайте вызывать `set_reference_count(k+1)` перед запуском `k` дочерних задач, если данная задача пользуется функцией `wait_for_all` для возобновления работы после завершения всех потомков. В противном случае поведение не определено. Отладочная версия библиотеки обычно обнаруживает такую ошибку и сообщает о ней;
4) запустить две дочерние задачи. Запуск задачи – это указание планировщику, что он может начать ее выполнение, когда и как сочтет удобным, быть может, параллельно с другими задачами. Первый вызов функции запуска `tbb::task::spawn(b)` возвращает управление немедленно, не дожидаясь начала выполнения дочерней задачи. Второй вызов функции запуска, `tbb::task::spawn_and_wait_for_all(a)`, эквивалентен цепочке из двух вызовов `tbb::task::spawn(a); tbb::task::wait_for_all()`. Последняя функция-член заставляет родителя ждать завершения всех работающих дочерних задач. По этой причине мы говорим, что это реализация с блокирующими задачами;
5) по завершении обеих дочерних задач атрибут `ref_count` родительской задачи будет уменьшен дважды и теперь равен 1. Поэтому родительская задача возобновляется с точки, следующей за вызовом `spawn_and_wait_for_all(a)`, вычисляет сумму `x+y` и записывает ее в `*sum`.

На рис. 10.9 показано создание и выполнение задач в случае, когда сначала была вызвана функция `root_task FibTask(8, &sum)` и `cutoff=7`. В предположении, что все задачи исполняет один поток, и несколько упростив представление операций со стеком, на рис. 10.9 показано, как выполняются вычисления. Пос-

ле вызова parallel_fib(8) переменная sum сохраняется в стеке, а корневой задаче выделяется место в памяти, где она конструируется с помощью вызова FibTask(8, &sum). Эта корневая задача выполняется рабочим потоком, который исполняет переопределенную функцию-член execute(). Внутри этой функции объявлены две локальные переменные x и y и созданы две новые дочерние задачи a и b, которые помещены в локальную двустороннюю очередь потока. Конструкторы этих задач вызываются как FibTask(7, &x) и FibTask(6, &y), это означает, что переменные-члены sum вновь созданных задач будут указывать соответственно на локальные переменные x и y в стеке задачи FibTask(8).

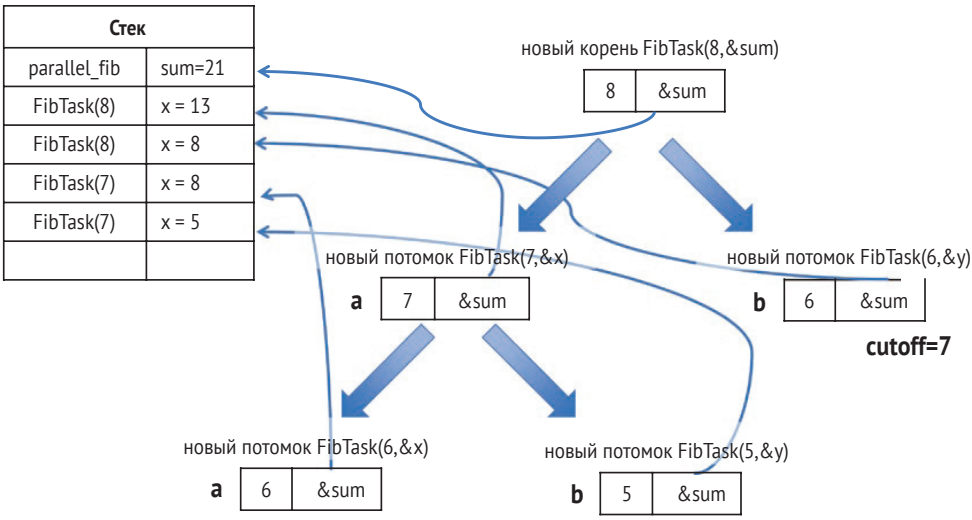


Рис. 10.9 ❖ Дерево рекурсивных вызовов для parallel_fib(8) с cutoff=7

Функция-член execute() продолжает работу: присваивает счетчику ref_count задачи значение 3, запускает сначала b, затем a и ждет завершения обеих. В этот момент корневая задача приостанавливается, ожидая завершения всех потомков. Напомним, что это блокирующий метод. Рабочий поток возвращается к планировщику, где сначала извлекает из очереди задачу a (поскольку она была помещена в очередь последней). Эта задача (FibTask(7,&x)) рекурсивно повторяет тот же самый процесс, приостанавливая свое выполнение после размещения переменных x и y в стеке и запуска задач FibTask(5,&x) и FibTask(6,&y). Поскольку cutoff=7, обе новые задачи подпадают под определение базового случая и вызывают соответственно fib(5) и fib(6). Первой извлекается из очереди задача FibTask(6,&x), она записывает 8 в *sum (сейчас sum указывает на x в стеке задачи FibTask(7)) и возвращает nullptr. Затем объект FibTask(6,&x) уничтожается, и по ходу дела переменная ref_count родительской задачи (FibTask(7,&x)) уменьшается на единицу. Затем рабочий поток извлекает из очереди задачу FibTask(5,&y), которая записывает 5 в *sum (теперь это указатель на y в стеке) и тоже возвращает nullptr. В результате ref_count оказывается равно 1, поэтому пробуждается родительский поток FibTask(7,&x), который

должен просто сложить 5+8, записать сумму в *sum (указатель на x в стеке задачи FibTask(8)) и вернуть nullptr. Это приводит к уменьшению ref_count корневой задачи до 2. Далее рабочий поток извлекает из очереди задачу FibTask(6,&y), которая вызывает fib(6), записывает y=8 в стек, возвращает управление и умирает. После этого остается только корневая задача без потомков (ref_count=1), поэтому она может продолжить выполнение с точки, следующей за вызовом spawn_and_wait_for_all(), т. е. вычислить 8+13, записать сумму в *sum (указатель на переменную sum в стеке функции parallel_fib) и умереть. Вам, вероятно, уже наскучило читать описание этого процесса, как и нам – рассказывать о нем, но потерпите еще немного – осталось чуть-чуть. Представим теперь, что есть еще один рабочий поток. У каждого потока свой стек, и оба борются за задачи. Результат будет тот же – 21, и в общем-то будут выполняться те же задачи, только теперь мы не знаем, какой поток какую задачу будет исполнять. Но мы точно знаем, что если размер проблемы и количество задач достаточно велики и порог отсечения cutoff выбран разумно, то параллельный код будет работать быстрее, чем последовательный.

Примечание. Как мы видели, планировщик с заимствованием работ вычисляет граф задач. Это ориентированный граф, узлами которого являются задачи. Каждая задача указывает на своего родителя, которого можно назвать задачей-преемником, поскольку она ждет завершения данной. Если у задачи нет родителя (преемника), то поле parent содержит nullptr. Метод tbb::task::parent() дает доступ для чтения к указателю на преемника. У каждой задачи имеется поле ref_count, где учитывается количество задач, для которых она является преемником (т. е. количество потомков, завершения которых должен дождаться родитель, перед тем как продолжить выполнение).

А где же так разрекламированные рычаги управления и возможности настройки? Действительно, только что рассмотренный код на основе низкоуровневых задач делает по сути дела то же самое, что предыдущие версии с использованием parallel_invoke и task_group, только программировать пришлось дольше. Так что же мы получили взамен? Да просто в классе task имеются и другие функции-члены, которые мы вскоре рассмотрим, а описанная реализация – лишь фундамент, на котором мы возведем оптимизированную версию. Не уходите.

Низкоуровневый интерфейс задач: часть вторая – продолжение задачи

Описанный выше подход с блокирующими задачами может стать проблематичным, если в теле задачи много локальных переменных. Эти переменные размещаются в стеке и остаются там, пока объект задачи не уничтожен. Но задача не будет уничтожена, пока не завершатся все ее потомки. Это может стать препятствием, если проблема очень велика и трудно найти порог отсечения, который не ограничивал бы степень параллелизма. Такое случается в проблемах, решаемых методом ветвей и границ, где требуется найти оптимальное значение, разумно организовав древовидный обход пространства поиска.

В таких случаях дерево может оказаться очень большим и несбалансированным (некоторые ветви гораздо глубже других), а его глубина заранее неизвестна. Если применить к таким проблемам блокирующий подход, то количество задач может быть настолько велико, что стек переполнится.

Еще один тонкий недостаток блокирующего подхода – управление рабочим потоком, который наткнулся на вызов `wait_for_all()` в родительской задаче. Нет никакого смысла в том, чтобы рабочий поток простаивал в ожидании завершения дочерних задач, поэтому мы поручаем ему выполнение других задач. Это значит, что когда родительская задача будет готова продолжить выполнение, тот рабочий поток, который пестовал ее раньше, может быть занят другими делами и отреагирует не сразу.

Примечание. *Продолжение*, *продолжение*, *продолжение*!!! Авторы TBB и другие специалисты по распараллеливанию обожают стиль программирования, основанный на продолжении. Почему? Оказывается, что от этого зависит разница между программой, которую легко писать, и программой, которая «падает» из-за переполнения стека. Хуже того, если не пользоваться продолжениями, то код, предотвращающий такие падения, трудно понять, поэтому он ухудшает реноме параллельного программирования. По счастью, при проектировании TBB продолжения были приняты в расчет, так что по умолчанию поощряется именно такой подход. В потоковых графах (главы 3 и 17) всячески приветствуется использование узла типа `continue_node` (и других узлов, потенциально способных работать в обход планировщика). Параллельному программисту надлежит знать о мощи продолжений (и рециклинга задач, о котором мы поговорим ниже) – мы не должны позволять задаче ждать (и напрасно транжирить ценные ресурсы)!

Чтобы избежать такого рода трудностей, мы можем принять другой подход к кодированию – передача управления через продолжение. На рис. 10.10 показано определение нового класса задачи, который будем называть задачей-продолжением, а на рис. 10.11 рамкой обведены изменения в классе `FibTask`, необходимые для реализации передачи управления через продолжение.

```
class FibCont: public tbb::task {
public:
  long* const sum;
  long x, y;
  FibCont(long* sum_) : sum{sum_} {}
  tbb::task* execute(){
    *sum = x+y;
    return nullptr;
  }
};
```

Рис. 10.10 ❖ Класс задачи-продолжения `FibCont` для вычисления чисел Фибоначчи

В задаче-продолжении FibCont тоже имеется функция-член execute(), но теперь она включает только код, который следует выполнить после завершения дочерних задач. В нашем примере после завершения дочерних задач нужно лишь сложить вычисленные ими результаты и вернуть сумму, а это всего две строки после вызова spawn_and_wait_for_all(a) в коде на рис. 10.8. В классе задачи-продолжения объявлены три переменные-члена: указатель на конечную сумму sum и две частичные суммы, вычисляемые дочерними задачами, x и y. Конструктор FibCont(long* sum) инициализирует указатель. Теперь осталось изменить класс FibTask, так чтобы он правильно создавал и инициализировал объект задачи-продолжения FibCont.

```
class FibTask: public tbb::task {
public:
  long const n;
  long* const sum;
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // Переопределяет виртуальную функцию task::execute
    if(n<cutoff) {
      *sum = fib(n);
      return nullptr;
    }
    else {
      // long x, y: больше не нужны

      FibCont& c = *new(allocate_continuation()) FibCont{sum};
      FibTask& a = *new(c.allocate_child()) FibTask{n-1, &c.x};
      FibTask& b = *new(c.allocate_child()) FibTask{n-2, &c.y};
      // Установить ref_count в состояние «два потомка»
      c.set_ref_count(2);
      tbb::task::spawn(b);
      tbb::task::spawn(a);
      return nullptr;
    }
  }
};
```

Рис. 10.11 ❖ Подход к параллельному вычислению чисел Фибоначчи с передачей управления через продолжение

На рис. 10.11 базовый случай вообще не изменился, а в ветви `else` локальные переменные `x` и `y` больше не нужны и закомментированы. Зато появилась новая задача `c` типа `FibCont&`. Память для этой задачи выделяется функцией `allocate_continuation()`, которая похожа на функцию `allocate_child()`, но передает `c` ссылку на родителя вызывающей задачи (`this`) и устанавливает атрибут `parent` задачи `this` в `nullptr`. Счетчик ссылок `ref_count` родителя `this` не изменяется, потому что родитель имеет столько же потомков, сколько и раньше, только вот сам родитель поменял тип с `FibTask` на `FibCont`. Если вы счастливый родитель, не пытайтесь сделать это дома!

В этот момент задача `FibTask` еще жива, но скоро мы с ней простимся. У `FibTask` больше нет родителя, но перед смертью она должна привести в порядок свои дела. Сначала `FibTask` создает две дочерние задачи того же типа `FibTask`, однако следите за руками!

- Новые задачи `a` и `b` теперь являются потомками `c` (а не `this`), поскольку мы создавали их функцией `c.allocate_child()`, а не просто `allocate_child()`. Иными словами, `c` является преемником `a` и `b`.
- Результат, вычисленный дочерними задачами, теперь не записывается в находящиеся в стеке переменные. Конструктор, инициализирующий `a`, вызывается как `FibTask(n-1,&c.x)`, так что указатель, по которому нужно сохранить сумму, вычисленную дочерней задачей `a` (`a.sum`), на самом деле указывает на `c.x`. Аналогично `b.sum` указывает на `c.y`.
- Счетчику ссылок в `c` (закрытой переменной-члену `c.ref_count`) присваивается значение 2, а не 3 (`c.set_ref_count(2)`), поскольку задача `FibCont c` имеет всего двух потомков (`a` и `b`).

Теперь дочерние задачи `a` и `b` готовы к запуску, так что все долги `FibTask` в этом мире оплачены. Она может со спокойной душой умереть, а занятая ей память будет использована для других целей. Покойся с миром.

Примечание. В предыдущем разделе было сказано, что в подходе с блокирующими задачами, если задача `A` запускает `k` потомков и ждет их, вызвав функцию-член `wait_for_all`, то счетчику `A.ref_count` нужно присвоить значение `k+1`. Лишняя единица учитывает дополнительную работу, которую задача `A` должна доделать, прежде чем завершиться и передать управление своему родителю. Эта единица не нужна, когда управление передается через продолжение, поскольку вся дополнительная работа возлагается на задачу-продолжение `C`. В таком случае в `C.ref_count` нужно записать ровно `k`, если количество дочерних задач равно `k`.

Чтобы наглядно проиллюстрировать, как все работает при передаче управления через продолжение, мы привели несколько мгновенных снимков процесса на рис. 10.12 и 10.13.

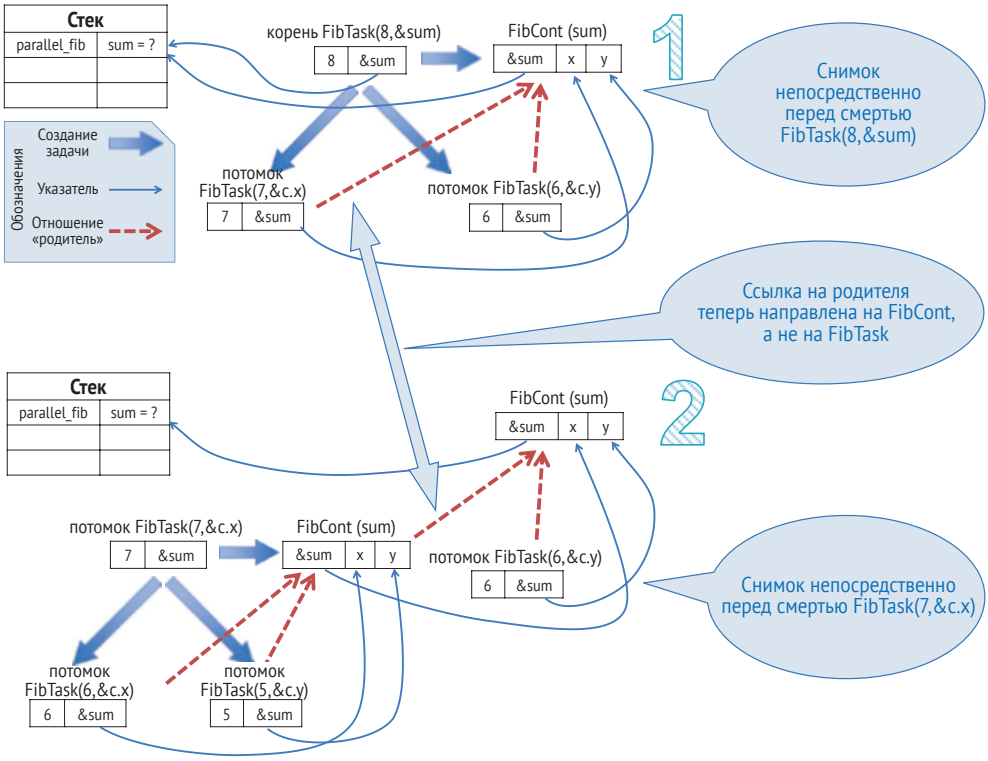


Рис. 10.12 ❖ Подход с передачей управления через продолжение к реализации `parallel_fib(8)` с `cutoff=7`

В правом верхнем углу рис. 10.12 корневая задача `FibTask(8,&sum)` уже создала продолжение `FibCont(sum)` и задачи `FibTask(7,&c.x)` и `FibTask(6,&c.y)`, которые в действительности являются потомками `FibCont`. Как видим, в стеке находится только конечный результат `sum`, объявленный в функции `parallel_fib`, поскольку при таком подходе переменные `x` и `y` не занимают места в стеке. Теперь `x` и `y` – переменные-члены `FibCont` и размещены в куче. В нижней части рисунка мы видим, что прежняя корневая задача исчезла, а вся занятая ей память освобождена. По сути дела мы обменяли место в стеке на место в куче, а объекты типа `FibTask` на объекты типа `FibCont`, что выгодно, если объекты `FibCont` меньше по размеру. Мы также видим, что ссылка на родителя из `FibTask(7,&c.x)` теперь направлена на более молодую задачу `FibCont(&sum)`.

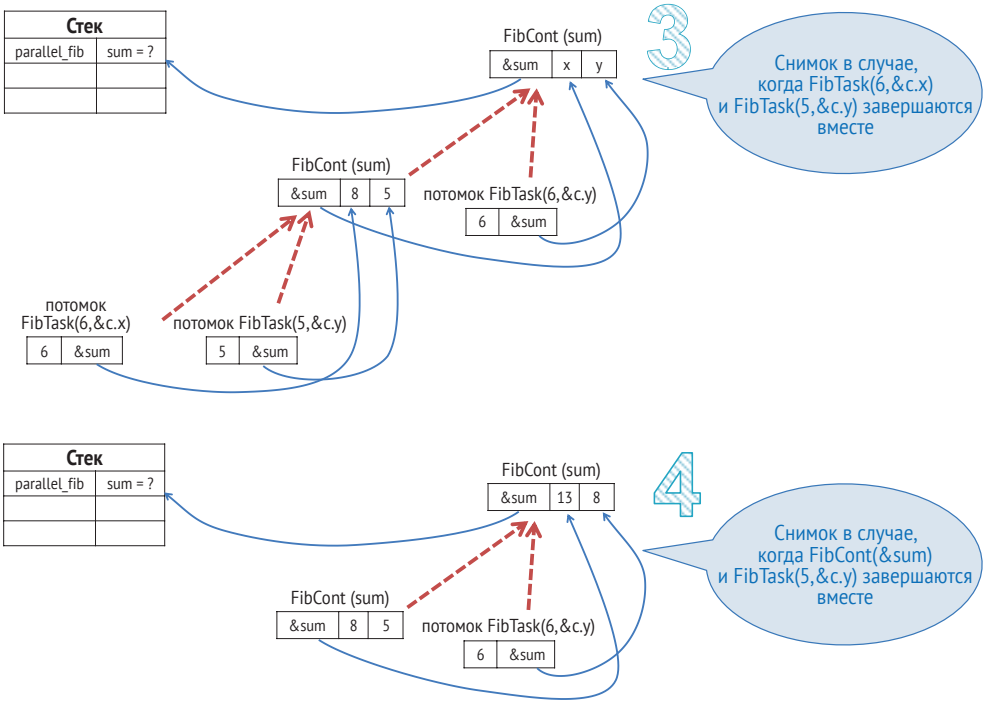


Рис. 10.13 ❖ Подход с передачей управления через продолжение (продолжение!)

В верхней части рис. 10.13 начинается раскрутка рекурсивного алгоритма. Никаких следов от объектов `FibTask` больше не осталось. Дочерние задачи `FibTask(6,&c.x)` и `FibTask(5,&c.y)` свелись к базовому случаю (`n<cutoff` в предположении, что `cutoff=7`) и вот-вот вернут управление, записав в `*sum` 8 и 5 соответственно. Оба потомка вернут `nullptr`, так что управление снова получает рабочий поток, который возвращается к планировщику, уменьшает `ref_count` родительской задачи и проверяет, равен `ref_count` нулю или еще нет. В таком случае – в полном соответствии с высокоуровневым описанием цикла диспетчеризации задачи в TBB, изображенным на рис. 9.14, – следующей будет выбрана родительская задача (в данном случае `FibCont`). В отличие от блокирующего подхода, теперь она выполняется немедленно. В нижней части рис. 10.13 мы видим обоих потомков исходной корневой задачи в момент, когда они уже записали свои результаты.

Может возникнуть вопрос: ждет ли еще функция `parallel_fib`, вызвавшая `spawn_root_and_wait(a)`, завершения первой корневой задачи – ведь исходная задача `FibTask` была заменена первым объектом `FibCont`, а затем скончалась (см. рис. 10.12)? Отвечаем: ждет, потому что `spawn_root_and_wait` спроектирована так, чтобы корректно обрабатывалась передача управления через продолжение. Вызов `spawn_root_and_wait(x)` на самом деле не ждет завершения `x`. Вместо этого он конструирует фиктивного преемника `x` и ждет, пока `ref_count` этого преемника обратится в 0. Поскольку `allocate_continuation` перенаправляет ссылку на родителя на задачу-продолжение, счетчик `ref_count` фиктивного

преемника не уменьшается, пока не завершится задача-продолжение FibCont. Это показано на рис. 10.14.

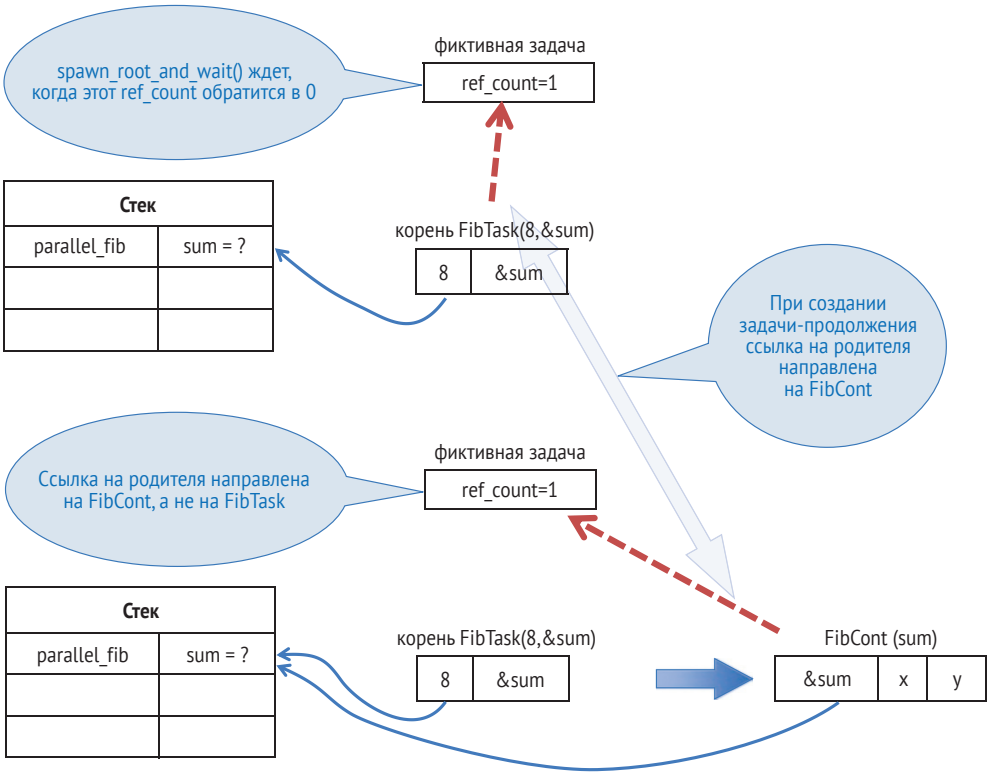


Рис. 10.14 ❖ parallel_fib ждет завершения FibCont благодаря фиктивной задаче со своим собственным счетчиком ref_count

Обход планировщика

Обход планировщика – оптимизация, при которой мы сами указываем, какую задачу выполнять следующей, а не отдаем выбор на откуп планировщику. Передача управления через продолжение зачастую открывает возможность для такого обхода. Например, в рассмотренном выше примере оказывается, что после возврата из FibTask::execute() по правилам заимствования работ, описанным в главе 9, всегда выбирается следующая задача из пула готовых, поскольку именно она запускалась последней (если только не была заимствована другим рабочим потоком). Точнее, последовательность событий выглядит следующим образом:

- поместить задачу в двустороннюю очередь потока;
- вернуть управление из функции-члена execute();
- извлечь задачу из двусторонней очереди потока, если только она не была заимствована другим потоком.

Помещение задачи в очередь и последующее извлечение ее оттуда, во-первых, сопряжено с накладными расходами, которых можно избежать, а во-вторых, и это еще хуже, открывает возможность заимствования, а значит, нарушения локальности без сколько-нибудь существенного увеличения степени параллелизма. Для предотвращения обеих проблем нужно, чтобы `execute()` не запускала задачу, а возвращала указатель на нее. Тогда гарантируется, что `a` будет исполняться в том же самом потоке, причем немедленно, а не в каком-то другом. Таким образом, в коде на рис. 10.11 следует заменить две строки:

```
spawn(a);          →          //spawn(a); закомментировано!
return nullptr;               return &a;
```

Низкоуровневый интерфейс задач: часть третья — рециклинг задач

Возникает желание обойти не только планировщик, но и процедуры выделения и освобождения памяти для задачи. Такая возможность часто возникает для рекурсивных задач, которые обходят планировщик, потому что дочерняя задача инициируется сразу после возврата, как только родитель завершается. На рис. 10.15 показано, какие изменения следует внести в пример вычисления чисел Фибоначчи, чтобы реализовать рециклинг задач.

Место потомка, который раньше назывался `a`, теперь занял возвращенный в оборот `this`. У вызова `recycle_as_child_of(c)` несколько последствий:

- он говорит, что `this` не следует автоматически уничтожать после возврата из `execute`;
- он делает `c` преемником `this`. Чтобы предотвратить возможные проблемы с подсчетом ссылок, функция `recycle_as_child_of` требует, чтобы в момент ее вызова преемником `this` был `nullptr` (указатель на родителя в объекте `this` должен содержать `nullptr`). Именно так обстоит дело после вызова `allocate_continuation`.

Переменные-члены необходимо переустановить, так чтобы имитировать ситуацию, которую раньше создавал конструктор `FibTask(n-1,&c.x)`. В данном случае `this->n` уменьшается на 1 (`n-=1;`), а в `this->sum` записывается указатель на `c.x`.

После рециклинга нужно позаботиться о том, чтобы переменные-члены `this` не использовались в текущем выполнении задачи, после того как запущена рециклированная задача. В нашем примере так и есть, потому что рециклированная задача на самом деле не запускается, а начинает работать только после возврата указателя `this`. Мы можем вместо этого запустить рециклированную задачу (т. е. выполнить `spawn (*this); return nullptr;`) при условии, что ни одна из ее переменных-членов не будет использоваться после запуска. Это ограничение распространяется даже на константные переменные-члены, поскольку не исключено, что после запуска задача может начать и закончить работу и будет уничтожена еще до того, как родитель успеет что-нибудь сделать. Похожая функция-член `task::recycle_as_continuation()` рециклирует задачу как продолжение, а не как дочернюю.

```
class FibTask: public tbb::task {
public:
   long n;    // больше не const
   long* sum; // больше не const-указатель
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // переопределяет виртуальную функцию task::execute
    if(n<cutoff) {
      *sum = fib(n);
      return nullptr;
    }
    else {
      // long x, y; больше не нужны
      FibCont& c = *new(allocate_continuation()) FibCont{sum};
      FibTask& b = *new(c.allocate_child()) FibTask{n-2, &c.y};
       recycle_as_child_of(c);
       this->n -=1;
       this->sum = &c.x;
      // Установить ref_count в состояние «два потомка»
      c.set_ref_count(2);
      tbb::task::spawn(b);
       return this; // было: return &a;
    }
  }
};
```

Рис. 10.15 ❖ Рециклинг задач при параллельном вычислении чисел Фибоначчи

На рис. 10.16 показан эффект рециклинга `FibTask(8,&sum)` в качестве потомка `FibCont`, после того как потомок обновил переменные-члены (заменил 8 на 7, а в `sum` записал указатель на `c.x`).

Примечание. ***Более экологичное (и простое) параллельное программирование*** ☺ Включение компонуемости, продолжений и рециклинга задач заметно упрощает параллельное программирование с применением TBB. Рециклинг, т. е. переработка отходов, все шире распространяется в мире, а рециклинг задач тоже помогает экономить энергию! Присоединяйтесь к движению за экологичное параллельное программирование – а тот факт, что оно к тому же проще и эффективнее, тоже не повредит!

Обход планировщика и рециклинг задач – мощные инструменты, которые помогают значительно улучшить и оптимизировать код. Они используются при реализации высокоуровневых шаблонов, с которыми мы познакомились в главах 2 и 3, и мы тоже можем применить их, когда будем проектировать шаблоны под свои нужды. В потоковых графах (главы 3 и 17) всячески рекомендуется использовать узлы типа `continue_node` (и другие узлы с возможностью обхода планировщика). В следующем разделе мы воспользуемся низкоуровневым API задач и оценим эффект этого предприятия, но сначала подготовим «контрольный список».

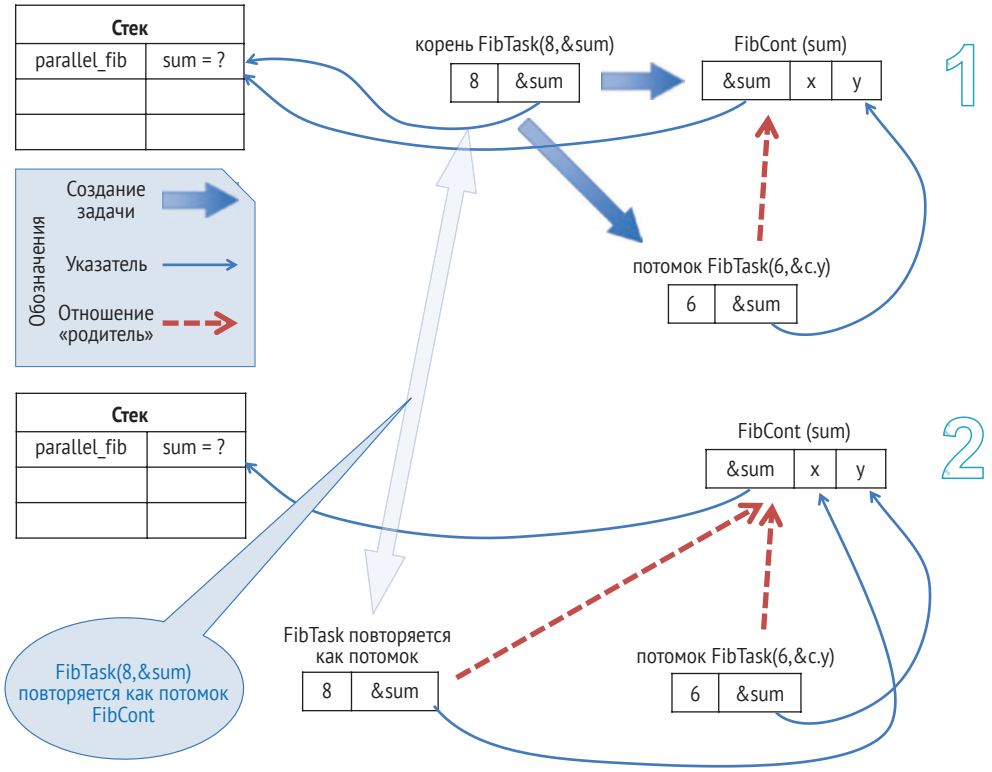


Рис. 10.16 ❖ Рециклинг `FibTask(8,&sum)` в качестве потомка `FibCont`

Контрольный список для интерфейса задач

Обращение к интерфейсу задач рекомендуется в случае параллелизма типа разветвление–соединение с большим количеством разветвлений. Тогда заимствование может привести сначала к распространению «в ширину» и занятию всех потоков, которые затем будут обходить дерево задач «в глубину», пока не возникнет необходимость в заимствовании работы. Иными словами, фундаментальная стратегия планировщика задач выражается словами «заимствуй в ширину, работай в глубину». Правило заимствования в ширину раздвигает рамки параллелизма, чтобы обеспечить занятость потоков. А правило работы в глубину гарантирует, что каждый поток будет работать эффективно, пока у него достаточно работы.

Напомним, что API не самый простой, поскольку при проектировании на первое место ставилось быстродействие. Во многих случаях мы сталкиваемся с проблемой, которую можно решить с помощью высокоуровневого интерфейса, например шаблонов `parallel_for`, `parallel_reduce` и т. д. Но если это не так и нужна дополнительная производительность, предлагаемая API задач, то стоит помнить о следующих деталях:

- всегда используйте для выделения памяти под задачу оператор `new(allocation_method) T`, где `allocation_method` – один из методов выделе-

ния класса task (см. приложение В, рис. В.76). Не создавайте экземпляров задач с локальной или файловой областью видимости;
- все «братья» (задачи одного уровня) должны быть выделены до начала работы, если только вы не собираетесь использовать функцию allocate_additional_child_of. Мы обсудим этот момент в последнем разделе главы;
- пользуйтесь передачей управления через продолжение, обходом планировщика и рециклингом задач, чтобы выжать всю производительность до последней капли;
- если задача завершилась и не была помечена для повторного выполнения (рециклинга), то она автоматически уничтожается. Кроме того, счетчик ссылок в ее преемнике уменьшается на 1, а если он обращается в ноль, то преемник автоматически запускается.

И еще одно: FIFO-задачи (типа запустил и забыл)

До сих пор мы видели, как запускаются задачи, и результат запуска задачи: поток, который поместил задачу в очередь, скорее всего, и извлечет ее обратно – в порядке LIFO (последним пришел – первым ушел), если только какой-нибудь другой поток не позаимствует ее. Как было сказано, такое поведение благотворно сказывается на локальности и снижении потребления памяти благодаря правилу «работай в глубину». Но запущенная задача может оказаться похороненной в локальной очереди потока, если после нее было запущено много других задач.

Если мы предпочитаем выполнять задачи в порядке FIFO, то следует помещать задачу в очередь функцией enqueue, а не spawn:

```
class FifoTask : public tbb::task {
public:
  tbb::task *execute() {// выполнить работу}
};

FifoTask& t = *new(tbb::task::allocate_root()) FifoTask();
tbb::task::enqueue(t);
```

Класс FifoTask в этом примере наследует классу tbb::task и переопределяет функцию-член execute(), как всякая задача. Но у этой задачи есть четыре отличия от запускаемых:
- планировщик может отложить запускаемую задачу до тех пор, пока кто-то не начнет ее ждать. Но задача, поставленная в очередь функцией enqueue, в конечном итоге будет выполнена, даже если никакой поток не ожидает ее явно. Даже если общее количество рабочих потоков равно нулю, для выполнения поставленных в очередь задач создается специальный дополнительный рабочий поток;
- запущенные задачи планируются в порядке LIFO (из очереди выбирается та, что была запущена последней), а поставленные в очередь обрабатываются приблизительно (не точно) в порядке FIFO (выбираются примерно в том порядке, в каком были помещены в очередь, – благода-

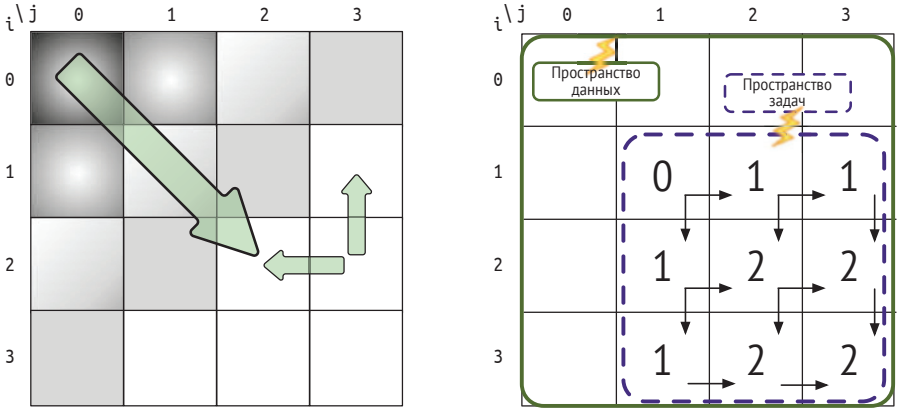
ря «приблизительности» TBB получает нужную гибкость, чтобы достичь более высокой производительности, чем при строгом следовании дисциплине);

- благодаря обходу в глубину запущенные задачи идеальны для рекурсивного параллелизма, когда нужно сэкономить память, тогда как поставленные в очередь при рекурсивном параллелизме могут потреблять недопустимо много памяти, поскольку при обходе в ширину рекурсия расширяется;
- запущенные родительские задачи должны ждать завершения своих запущенных потомков, тогда как задач, поставленных в очередь, дожидаться не надо, т. к. не исключено, что сначала нужно будет обработать прочие поставленные в очередь задачи совсем из других частей программы. При использовании поставленных в очередь задач рекомендуется, чтобы они посылали асинхронный сигнал по завершении. По существу, поставленные в очередь задачи следует выделять как корневые, а не как дочерние, которых потом придется ждать.

В главе 14 постановка задач в очередь иллюстрируется в контексте придания одним задачам более высоких приоритетов, чем другим. В руководстве по паттернам проектирования для TBB (см. раздел «Дополнительная информация» в конце главы) рассмотрено еще два сценария использования, когда поставленные в очередь задачи оказываются удобны. В первом случае поток пользовательского интерфейса (GUI) должен откликаться на действия пользователя, даже когда инициирована долго работающая задача. В предложенном решении поток GUI ставит задачу в очередь, но не ждет ее завершения. Задача трудится, а перед завершением отправляет потоку GUI сообщение. Второй паттерн также связан с назначением невытесняющих приоритетов различным задачам.

Применение низкоуровневых средств на практике

Чтобы оценить различные реализации на основе задач, рассмотрим более сложное приложение. Волновой фронт – это паттерн программирования, встречающийся в научных приложениях, например основанных на динамическом программировании или выравнивании последовательностей. Элементы данных распределяются на многомерной сетке, представляющей логическую плоскость или пространство. Элементы должны вычисляться в определенном порядке, потому что между ними имеются зависимости. В качестве примера на рис.10.7 показан двумерный волновой фронт. Здесь вычисления начинаются в одном углу матрицы и распространяются по плоскости в противоположный угол вдоль диагонали. Каждая антидиагональ представляет количество вычислений или элементов, которые можно было бы вычислить параллельно, не создавая зависимостей.



(a) Диагональ заметания (b) Матрица счетчиков

Рис. 10.17 ❖ Типичный двумерный волновой паттерн (a) и зависимости, представленные в виде матрицы атомарных счетчиков (b)

В коде на рис. 10.18 вычисляется функция в каждой ячейке двумерной сетки $n \times n$. Каждая ячейка зависит от данных в двух соседних ячейках. Так, на рис. 10.17(a) мы видим, что ячейка (2,3) зависит от ячейки (1,3), расположенной к северу от нее, и от ячейки (2,2), расположенной к западу, поскольку на каждой итерации циклов по `i` и по `j` используются ячейки, вычисленные на предыдущих итерациях: `A[i,j]` зависит от `A[i-1,j]` и от `A[i,j-1]`. На рис. 10.18 показана последовательная версия вычислений в случае, когда массив `A` линеаризован. Очевидно, что антидиагональные ячейки независимы, поэтому их можно вычислять параллельно. Чтобы задействовать этот параллелизм (циклы по `i` и по `j`), задача будет выполнять вычисления, соответствующие каждой ячейке в пространстве итераций (которое мы далее будем называть пространством задач), и независимые задачи будут выполняться параллельно.

```
for (int i=1; i<n; ++i)
  for (int j=1; j<n; ++j)
    A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
```

Рис. 10.18 ❖ Фрагмент кода, иллюстрирующий двумерный волновой фронт. Массив `A` содержит линеаризованную двумерную сетку

В нашей стратегии распараллеливания основной единицей работы является вычисление, производимое функцией `foo` в каждой ячейке матрицы `(i,j)`. Без ограничения общности можно предположить, что вычислительная нагрузка для каждой ячейки управляется параметром `gs` (степень детализации) функции `foo`. Таким образом, мы можем определить зернистость задач и, следовательно, изучить зависимость производительности от зернистости, а также ситуации с однородной и неоднородной рабочей нагрузкой в разных задачах.

На рис. 10.17(b) стрелки показывают поток зависимостей по данным для нашего волнового фронта. Например, после выполнения левой верхней задачи (1, 1), которая не зависит ни от какой другой, можно начинать две новые задачи (одну снизу, (2, 1), другую справа, (1, 2)). Эту информацию о зависимостях можно собрать в двумерной матрице счетчиков типа той, что показана на рис. 10.17(b). Значение счетчика показывает, завершения скольких задач необходимо дождаться. Начинать можно только те задачи, для которых счетчик равен нулю.

Альтернативная реализация такого волнового вычисления рассмотрена в руководстве по паттернам проектирования для Intel TBB (см. раздел «Дополнительная информация»), где реализован общий граф ациклических задач. Эта версия имеется в исходном коде к данной главе в файле `wavefront_v0_DAG.cpp`. Однако в ней требуется, чтобы все задачи были созданы заранее, а реализация, представленная ниже, более гибкая и допускает настройку с целью более эффективного использования локальности. На рис. 10.19 показан первый вариант реализации, `wavefront_v1_addchild`. Каждая готовая задача сначала выполняет свое тело, а затем уменьшает счетчики зависящих от нее задач. Если в результате операции декремента какой-то счетчик обращается в 0, то на задачу возлагается обязанность запустить новую независимую задачу. Заметим, что счетчики разделяемые и модифицируются несколькими параллельно работающими задачами, поэтому они объявлены как атомарные переменные (см. главу 5).

```
class Cell: public tbb::task {
  int i,j;
  int n;
  int gs;
  std::vector<double> &A;
  std::vector<std::atomic<int>> &counters;
public:
  Cell(int i_ ,int j_, int n_, int gs_,
       std::vector<double> &A_,
       std::vector<std::atomic<int>> &counters_) :
       i{i_},j{j_},n{n_},gs{gs_},A{A_},counters{counters_} {}
  task* execute(){
    A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
    if (j<n-1 && --counters[i*n+j+1]==0) // восточная ячейка готова
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i,j+1,n,gs,A,counters});
    if (i<n-1 && --counters[(i+1)*n+j]==0) // южная ячейка готова
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i+1,j,n,gs,A,counters});
    return nullptr;
  }
};
```

Рис. 10.19 ❖ Часть кода в файле wavefront_v1_addchild

Отметим, что на рис. 10.19 в качестве метода выделения памяти для новых задач использована функция `allocate_additional_child_of(*parent())`. Этот метод позволяет добавлять дочерние задачи, когда другие работают. Плюс в том, что таким образом мы можем не писать код, гарантирующий, что все дочерние задачи выделены, прежде чем хотя бы одна из них запущена (поскольку это зависит от того, готовы ли к работе восточная, южная или обе задачи). Минус в том, что при таком методе выделения требуется, чтобы поле `ref_count` родителя обновлялось атомарно (увеличивалось, когда выделен один «дополнительный потомок», и уменьшалось при завершении любого потомка). Поскольку мы используем `allocate_additional_child_of(*parent())`, все созданные задачи будут детьми одного и того же родителя. Первой является задача (1, 1), она запускается так:

```
tbb::task::spawn_root_and_wait(*new(tbb::task::allocate_root())
                    Cell{1,1,n,gs,A,counters});
```

и родителем этой корневой задачи является фиктивная задача, с которой мы уже познакомились на рис. 10.14. Затем все задачи, созданные в этой программе, атомарно обновляют поле `ref_count` фиктивной задачи.

Еще один подвох при использовании метода выделения `allocate_additional_child_of` состоит в том, что пользователь (это мы) должен гарантировать, что поле `ref_count` родителя не обратится в 0 слишком рано, до того как выделена дополнительная дочерняя задача. В нашем коде этот нюанс учтен, т. к. задача `t`, выделяющая память для дополнительного потомка `c`, уже гарантирует, что `ref_count` родителя `t` не меньше 1, поскольку `t` уменьшает `ref_count` своего родителя только при завершении (т. е. после выделения `c`).

В главе 2 уже был представлен шаблон `parallel_do_feeder` для иллюстрации другого волнового приложения: прямой подстановки. Этот шаблон по существу реализует алгоритм списка работ таким образом, что новые задачи можно динамически добавлять в список работ путем вызова функции-члена `parallel_do_feeder::add()`. Будем называть `wavefront_v2_feeder` версию программы волнового фронта, в которой используется `parallel_do_feeder` и, как на рис. 2.19, вызывается функция `feeder.add()` вместо обращения к функциям `spawn`, как на рис. 10.19.

Если мы не хотим, чтобы один родитель ждал все дочерние задачи, которые одновременно стремятся атомарно обновить его поле `ref_count`, то можем реализовать более хитроумную версию, имитирующую стиль блокировки, который обсуждался выше. На рис. 10.20 показана функция-член `execute()` для этого случая. В ней мы сначала помечаем, какие ячейки готовы – восточная, южная или обе, – затем выделяем память и запускаем соответствующие задачи. Отметим, что теперь используется метод `allocate_child()` и каждая задача должна ждать не более двух потомков. Атомарное обновление единственного счетчика `ref_count` уже не является узким местом, но больше задач пассивно ждут завершения своих потомков (и занимают память). Эту версию мы назовем `wavefront_v3_blockstyle`.

```
task* execute(){
  A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
  int east = 0,south = 0;
  if (j<n-1 && --counters[i*n+j+1]==0) east=1;
  if (i<n-1 && --counters[(i+1)*n+j]==0) south=1;
  set_ref_count(1+east+south);
  if(east==1 && south==0)
    spawn_and_wait_for_all(*new(allocate_child())
                           Cell{i,j+1,n,gs,A,counters});
  if(east==0 && south==1)
    spawn_and_wait_for_all(*new(allocate_child())
                           Cell{i+1,j,n,gs,A,counters});
  if(east==1 && south==1) {
    //убедиться, что все потомки выделены, до запуска любого из них
    Cell &a = *new(allocate_child()) Cell{i,j+1,n,gs,A,counters};
    Cell &b = *new(allocate_child()) Cell{i+1,j,n,gs,A,counters};
    spawn(a);
    spawn_and_wait_for_all(b);
  }
  return nullptr;
}
```

Рис. 10.20 ❖ Функция `execute()` для версии `wavefront_v3_blockstyle`

Теперь обратимся к передаче управления через продолжение и рециклингу задач. В паттерне волнового фронта у каждой задачи был шанс запустить две новые (восточного и южного соседей). От запуска одной из них можно отказаться, вернув указатель на следующую задачу, тогда вместо запуска новой задачи будут использоваться ресурсы текущей. Как было объяснено выше, тем самым мы убиваем двух зайцев: уменьшаем количество операций выделения, вызовов `spawn()`, а также экономим на времени извлечения новых задач из локальной очереди. Новую версию назовем `wavefront_v4_recycle`, ее главное преимущество в том, что количество запусков уменьшено с $n \times n - 2n$ (столько их было в предыдущих версиях) до $n - 2$ (приблизительно размер столбца). Полная реализация имеется в прилагаемом к книге коде.

Кроме того, в ходе рециклинга мы можем сообщить планировщику о желательном назначении приоритетов задачам, например чтобы гарантировать обход структуры данных с учетом кеша, что может улучшить локальность данных. На рис. 10.21 приведен фрагмент версии `wavefront_v5_locality`, в котором такая оптимизация реализована. Мы устанавливаем флаг `recycle_into_east`, если к востоку от выполняемой задачи имеется готовая к работе. В противном случае устанавливается флаг `recycle_into_south`, если к работе готова южная задача. Впоследствии, ориентируясь на эти флаги, мы заменяем текущую задачу восточной или южной. Заметим, что, поскольку в этом примере структура данных хранится по строкам, в случае когда готовы обе задачи – восточная и южная, – кеш данных будет использоваться более эффективно, если для рециклинга выбрать восточную задачу. Тогда поток (ядро), исполняющий текущую

задачу, примет к себе задачу, которая обходит соседние данные, и мы извлечем максимум пользы из пространственной локальности. Поэтому в данном случае мы заменяем себя восточной задачей, а южную запускаем для последующего выполнения.

```
task* execute(){
  A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
  bool recycle_into_east=false;
  bool recycle_into_south=false;
  if (j<n-1 && --counters[i*n+j+1]==0) recycle_into_east=true;
  if (i<n-1 && --counters[(i+1)*n+j]==0){
      if (!recycle_into_east) recycle_into_south = true;
      else
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i+1,j,n,gs,A,counters});
    }
  if (recycle_into_east) {
    recycle_as_child_of(*parent());
    j = j+1;
    return this;
  }
  else if(recycle_into_south){
    recycle_as_child_of(*parent());
    i=i+1;
    return this;
  }
  else return nullptr;
}
```

Рис. 10.21 ❖ Функция `execute()` для версии `wavefront_v5_locality`

При решении очень больших волновых задач бывает важно уменьшить объем памяти, занимаемой каждой задачей. Если вас не смущает использование глобальных переменных, то можно подумать о хранении разделяемого глобального состояния всех задач (`n`, `gs`, `A` и `counters`) в глобальных переменных. Этот вариант реализован в версии `wavefront_v6_global`, которая имеется в каталоге с исходным кодом.

Пользуясь параметром `gs`, который определяет количество операций с плавающей точкой в одной задаче, мы выяснили, что для крупнозернистых задач, выполняющих более 2000 операций с плавающей точкой (FLOP), все семь версий мало отличаются друг от друга и масштабируются почти линейно. Это объясняется тем, что накладные расходы на распараллеливание исчезающе малы по сравнению с достаточно большим временем счета во всех задачах. Но на практике найти настолько крупнозернистые волновые задачи трудно. На рис. 10.22 показано ускорение, достигаемое версиями с нулевой по пятую на 4-ядерном процессоре Core i7-6700HQ (архитектура Skylake, 6-е поколение) с частотой 2.6 ГГц, L3-кешем размера 6 МБ и оперативной памятью 16 ГБ. Сте-

пень детализации gs равна всего 200 FLOP, а n = 1024 (при таком n версия 6 показывает такую же производительность, как версия 5).

	Посл.	v0	v1	v2	v3	v4	v5
Время (мс)	240	112	129	130	96	74	69
Ускорение	1	2,14	1,86	1,85	2,50	3,24	3,48

Рис. 10.22 ❖ Ускорение различных версий на 4-ядерном процессоре

Очевидно, что версия v5 дает наилучшее решение. На самом деле мы измеряли ускорение и для других значений степени детализации и обнаружили, что чем оно меньше, тем лучше версии v4 и v5 по сравнению с v1 и v2. Кроме того, интересно было узнать, что в значительной мере улучшение вызвано рециклингом, на что указывает превосходство v4 над v1. Более полное исследование было проведено А. Диосом и описано в статьях, перечисленных в конце главы.

Поскольку производительность алгоритмов с волновым фронтом уменьшается по мере измельчения рабочей нагрузки, разработана хорошо известная техника замощения, призванная противодействовать этой тенденции (краткое определение см. в глоссарии). Замощение позволяет достичь нескольких целей: лучше задействовать локальность, поскольку каждая задача некоторое время работает в пространственно ограниченной области данных; уменьшить количество задач (а значит, и количество операций выделения и запуска); сократить накладные расходы на управление волновым фронтом (объем занимаемой памяти и время инициализации матрицы счетчиков/зависимостей, которое уменьшается, поскольку теперь один счетчик требуется на каждую плитку, а не на каждый элемент матрицы). После огрубления задач посредством замощения мы снова вольны вернуться к версиям v1 или v2, верно? Однако у замощения есть недостаток: оно уменьшает количество независимых задач (они грубее, но их меньше). Поэтому если требуется масштабировать приложение на большое число ядер, а размер проблемы не растет в той же пропорции, то, по-видимому, для получения максимальной производительности придется прибегнуть ко всем низкоуровневым средствам TBB. В таких трудных ситуациях нам предстоит продемонстрировать, что мы в совершенстве овладели TBB и успешно отточили навыки параллельного программирования.

Резюме

В этой главе мы рассмотрели возможности задач, особенно полезные при реализации рекурсивно разбиваемых и волновых приложений. В качестве сквозного примера мы выбрали вычисление чисел Фибоначчи и реализовали его сначала с помощью высокоуровневого шаблона `parallel_invoke`, который обсуждали раньше. Затем мы погрузились глубже и воспользовались API среднего уровня в виде класса `task_group`. Этот интерфейс задач предлагает большую гибкость для оптимизации под конкретные нужды. Задачи TBB лежат в основе и других высокоуровневых шаблонов, представленных в первой части книги,

но их можно использовать также для создания собственных паттернов и алгоритмов, применяя передачу управления через продолжение, обход планировщика и рециклинг задач. Для еще более требовательных разработчиков имеются дополнительные возможности: приоритеты задач, привязка задач к потокам и явная постановка задач в очередь. Их мы рассмотрим в следующей главе. Нам очень хотелось бы увидеть, как вы воспользуетесь мощными инструментами, которые получили в свое распоряжение.

Дополнительная информация

Ниже приведен перечень дополнительных материалов к этой главе.

- *A. Dios, R. Asenjo, A. Navarro, F. Corbera, E. L. Zapata.* A case study of the task-based parallel wavefront pattern // Advances in Parallel Computing: Applications, Tools and Techniques on the Road to Exascale Computing. ISBN: 978-1-61499-040-6. Vol. 22. P. 65–72. IOS Press BV, Amsterdam, 2012 (расширенная версия имеется по адресу www.ac.uma.es/~compilacion/publicaciones/UMA-DAC-11-02.pdf).
- *A. Dios, R. Asenjo, A. Navarro, F. Corbera, E. L. Zapata.* High-level template for the task-based parallel wavefront pattern // IEEE Intl. Conf. on High Performance Computing (HiPC 2011), Bengaluru (Bangalore), India, December 18–21, 2011. Построение высокоуровневого шаблона поверх задач TBB с целью упростить реализацию волновых алгоритмов.
- *González Vázquez, Carlos Hugo.* Library-based solutions for algorithms with complex patterns of parallelism // PhD report, 2015. http://hdl.handle.net/2183/14385. Описывается три сложных параллельных паттерна и их реализация в виде высокоуровневых шаблонов поверх задач TBB.
- Руководство по паттернам проектирования с помощью библиотеки Intel TBB:
  - поток GUI: http://software.intel.com/en-us/node/506119;
  - приоритеты: http://software.intel.com/en-us/node/506120;
  - волновой фронт: http://software.intel.com/en-us/node/506110.

Глава 11

Управление количеством потоков

По умолчанию библиотека TBB инициализирует планировщик, задавая количество потоков, которое обычно оказывается правильным. Она создает на один поток меньше, чем количество логических ядер на данной платформе, оставляя одно ядро для главного потока приложения. Поскольку TBB реализует распараллеливание с помощью задач, планируемых для выполнения этими потоками, то ничего плохого в таком выборе нет – существует ровно один программный поток для каждого логического ядра, и алгоритмы планирования в TBB эффективно распределяют задачи между потоками, применяя заимствование работ, описанное в главе 9.

Однако есть много ситуаций, в которых это умолчание желательно изменить – и не без причины. Быть может, мы ставим эксперименты по масштабированию и хотим узнать, насколько хорошо приложение будет работать при разном числе потоков. Или в системе параллельно работает несколько приложений, так что нашему желательно отдать лишь часть имеющихся ресурсов. Или мы знаем, что приложение само создает дополнительные платформенные потоки для рендеринга, искусственного интеллекта или еще каких-то целей, и хотим ограничить TBB, оставив в системе место для других потоков. Так или иначе, если требуется изменить режим по умолчанию, то такая возможность есть.

Существует три класса, с помощью которых можно повлиять на количество потоков, участвующих в выполнении конкретного TBB-алгоритма или потокового графа. Но взаимодействия между этими классами могут быть очень сложными! В этой главе мы рассмотрим типичные случаи и дадим рекомендации, которых, вероятно, будет достаточно для всех сценариев, кроме самых зубодробительных. Такой уровень детализации удовлетворит большинство читателей, а наши советы подходят почти для всех ситуаций. Ну а тех, кто желает разобрать TBB до последнего винтика, приглашаем отправиться в путешествие по документации, где описаны все детали всех возможных взаимодействий между этими классами. Впрочем, если следовать описанным в этой главе паттернам, то вряд ли это понадобится.

Краткий обзор архитектуры планировщика TBB

Прежде чем приступать к разговору об управлении количеством потоков, исполняющих параллельные алгоритмы, вспомним структуру планировщика TBB (рис. 11.1). Более подробное описание планировщика приведено в главе 9.

Глобальный пул потоков (рынок) – то место, откуда рабочие потоки перемещаются на арены задач. Потоки перемещаются на те арены, где имеются готовые к выполнению задачи, а если потоков недостаточно для занятия всех слотов на всех аренах, то они занимают слоты пропорционально количеству слотов на арене. Например, если на одной арене задач слотов в два раза больше, чем на другой, то она получит примерно вдвое больше рабочих потоков.

Примечание. Если используются приоритеты задач, то рабочие потоки полностью удовлетворяют запросы со стороны арен с высокоприоритетными задачами и только потом занимают слоты на аренах с низкоприоритетными задачами. Приоритеты задач будут рассматриваться в главе 14. А в этой главе будем считать, что все задачи имеют одинаковый приоритет.

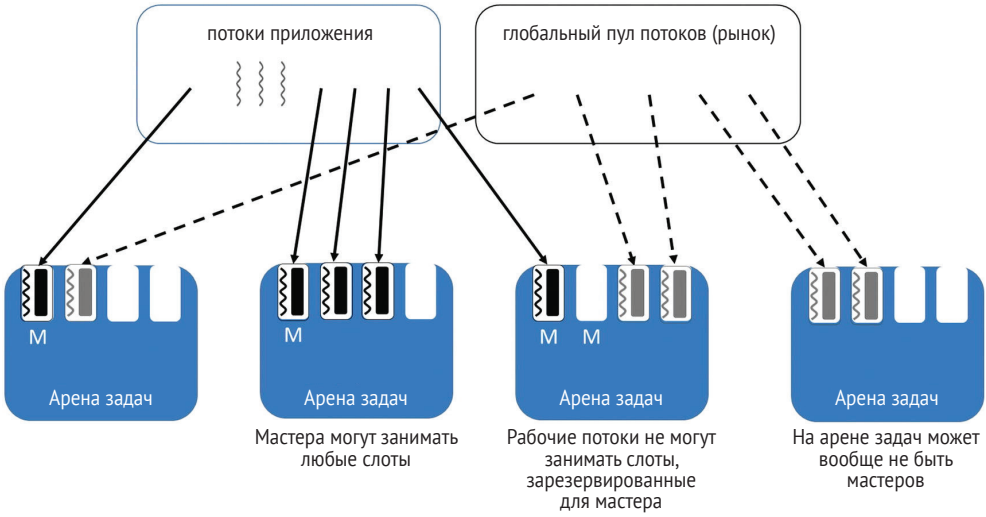


Рис. 11.1 ❖ Архитектура планировщика задач в TBB

Арены мастеров создаются одним из двух способов: (1) каждый поток-мастер по умолчанию получает собственную арену, когда выполняет TBB-алгоритм или запускает задачи, и (2) можно явно создать арену задач, воспользовавшись классом `task_arena`, который подробно описан в главе 12.

Если на арене задач нет работ, то приписанные к ней рабочие потоки возвращаются в глобальный пул потоков, чтобы поискать работу на других аренах, а если работы нет нигде, то засыпают.

Интерфейсы для управления количеством задач

Первая версия библиотеки TBB вышла более десяти лет назад и с тех пор развивалась в ногу с эволюцией платформ и рабочих нагрузок. Теперь TBB предлагает три способа управления потоками: `task_scheduler_init`, `task_arena` и `global_control`. В простых ситуациях одного из этих интерфейсов может оказаться достаточно для достижения цели, но в более сложных их приходится комбинировать.

Управление количеством потоков с помощью task_scheduler_init

В первой версии библиотеки TBB был всего один интерфейс для управления количеством потоков в приложении: класс `task_scheduler_init`. Интерфейс этого класса показан на рис. 11.2.

Объектом `task_scheduler_init` можно воспользоваться (1) для управления тем, в какой момент конструируется и уничтожается арена задач, ассоциированная с мастер-потоком; (2) для задания количества рабочих слотов на арене этого потока; (3) чтобы задать размер стека для каждого рабочего потока на арене; (4) для задания начального *мягкого* лимита (см. врезку ниже) количества потоков в глобальном пуле.

```
class task_scheduler_init {
public:
  static const int automatic = /* зависит от реализации */;
  static const int deferred = /* зависит от реализации */;

  task_scheduler_init(int number_of_threads=automatic,
                      stack_size_type thread_stack_size=0);
  ~task_scheduler_init();

  void initialize(int number_of_threads=automatic);
  void initialize(int number_of_threads,
                  stack_size_type thread_stack_size);
  void terminate();

  static int default_num_threads ();
  bool is_active() const;

  //ознакомительные возможности
  void blocking_terminate();
  bool blocking_terminate(const std::nothrow_t&)
noexcept(true);
};
```

Рис. 11.2 ❖ Интерфейс класса `task_scheduler_init`

Управление количеством потоков с помощью task_arena

Позже, когда TBB стала использоваться в крупномасштабных системах и более сложных приложениях, в нее был добавлен класс `task_arena` для создания *явных* арен задач как способа изолировать работу. Изоляция более подробно обсуждается в главе 12. А в этой главе мы рассмотрим, как класс `task_arena` позволяет задать количество доступных слотов на явной арене. Используемые в этой главе функции класса `task_arena` показаны на рис. 11.3.

Конструктор класса `task_arena` позволяет задать общее число слотов на арене в аргументе `max_concurrency` и число слотов, зарезервированных для мастер-потоков в аргументе `reserved_for_masters`. При передаче функтора методу `execute` вызывающий поток присоединяется к арене, и все задачи, запущенные внутри функтора, запускаются на этой арене.

```
class task_arena {
public:
  static const int automatic = implementation-defined;

  task_arena(int max_concurrency = automatic,
             unsigned reserved_for_masters = 1);

  template<typename F> auto execute(F& f) -> decltype(f());
  template<typename F> auto execute(const F& f) ->
decltype(f());
  template<typename F> void enqueue(F&& f);
  template<typename F> void enqueue(F&& f, priority_t p);
};
```

Рис. 11.3 ❖ Интерфейс класса `task_arena`

Мягкий и жесткий лимиты

У глобального пула потоков имеются *мягкий* и *жесткий лимиты*. Количество рабочих потоков, доступных для параллельного выполнения, равно минимальному из этих двух значений.

Мягкий лимит определяется запросами от объектов приложения `task_scheduler_init` и `global_control`. Жесткий лимит зависит от количества `P` логических ядер в системе. На момент написания этой книги жесткий лимит был равен 256 потокам на платформах, где `P <= 64`, `4P` – на платформах, где `64 < P <= 128`, и `2P` – на платформах, где `P > 128`.

Задачи TBB выполняются рабочими потоками TBB без вытеснения. Поэтому заводить больше потоков TBB, чем имеется логических ядер, не имеет особого смысла – просто операционной системе придется управлять большим количеством потоков. Если вы хотите, чтобы потоков TBB было больше, чем допускает *жесткий* лимит, то почти наверняка используете TBB неправильно или пытаетесь сделать нечто такое, для чего TBB не предназначена.

Управление количеством потоков с помощью global_control

После того как в библиотеку был включен класс `task_arena`, пользователи стали требовать интерфейса для прямого управления количеством потоков в глобальном пуле. Класс `global_control` считался *ознакомительной возможностью* до версии TBB 2019 Update 4 (теперь это полноправное средство, т. е. нет нужны активировать его с помощью макроса). Он позволяет изменять значения глобальных параметров планировщика задач, в т. ч. мягкого лимита количества потоков в пуле.

Определение класса `global_control` показано на рис. 11.4.

```
class global_control {
public:
  enum parameter {
    max_allowed_parallelism,
    thread_stack_size
  };
  global_control(parameter p, size_t value);
  ~global_control();
  static size_t active_value(parameter param);
};
```

Рис. 11.4 ❖ Интерфейс класса `global_control`

Сводка концепций и классов

В этом разделе перечислены концепции, освещаемые в данной главе, и последствия применения различных классов. Не переживайте, если что-то не понятно. В следующем разделе мы расскажем о рекомендуемых методах работы с этими классами для достижения конкретных целей. Так что хотя описанные здесь взаимодействия могут показаться сложными, типичные паттерны использования гораздо проще.

Планировщик. С планировщиком TBB связан глобальный пул потоков и по меньшей мере одна арена задач. После конструирования планировщика можно добавить дополнительные арены, что увеличит счетчик ссылок на планировщик. При уничтожении арены задач этот счетчик ссылок уменьшается. Вместе с уничтожением последней арены уничтожается и сам планировщик TBB, а также глобальный пул потоков. При любом последующем использовании задач TBB понадобится сконструировать новый планировщик. В процессе никогда не бывает более одного активного планировщика.

Жесткий лимит потоков. Существует жесткое ограничение на количество рабочих потоков, создаваемых планировщиком TBB. Оно зависит от аппаратного параллелизма данной платформы (см. врезку «Мягкий и жесткий лимиты»).

Мягкий лимит потоков. Существует мягкое ограничение на количество рабочих потоков, доступных планировщику TBB. Объект `global_control` позволяет изменить его напрямую. В противном случае оно инициализируется потоком, создавшим планировщик (см. врезку «Мягкий и жесткий лимиты»).

Мягкий лимит потоков по умолчанию. Если поток запускает задачу TBB, прямо с помощью низкоуровневого интерфейса или опосредованно, воспользовавшись TBB-алгоритмом или потоковым графом, то будет создан планировщик TBB, если его еще не существует. Если ни один объект `global_control` не задал мягкий лимит явно, то по умолчанию ему присваивается начальное значение `P-1`, где `P` – количество логических ядер на платформе.

Объект `global_control`. Объект `global_control` на протяжении срока своей жизни определяет мягкий лимит количества рабочих потоков, доступных планировщику TBB. В любой момент времени мягкий лимит равен минимальному из значений `max_concurrency_limit`, запрошенных всеми активными объектами `global_control`. Если мягкий лимит был инициализирован до конструирования всех активных объектов `global_control`, то это начальное значение также учитывается при нахождении минимума. После уничтожения объекта `global_control` мягкий лимит может увеличиться, если он был равен значению, запрошенному уничтоженным объектом. Создание объекта `global_control` не приводит ни к инициализации планировщика TBB, ни к увеличению числа ссылок на него. После уничтожения последнего объекта `global_control` восстанавливается значение мягкого лимита по умолчанию.

Объекты `task_scheduler_init`. Объект `task_scheduler_init` создает арену задач, ассоциированную с мастер-потоком, но только если для данного потока ее еще не существует. Если же арена уже существует, то увеличивается количество ссылок на нее. При уничтожении объекта `task_scheduler_init` счетчик ссылок уменьшается на единицу, и если он обращается в ноль, то арена задач уничтожается. Если в момент конструирования объекта `task_scheduler_init` планировщика TBB не существует, то он создается, и если мягкий лимит потоков не был установлен объектом `global_control`, то он инициализируется, исходя из значения аргумента `max_threads` конструктора, следующим образом:

`P-1`, где `P` – количество логических ядер	если `max_threads <= P - 1`
`max_threads`	в противном случае

Объекты `task_arena`. Объект `task_arena` создает явную арену задач, не ассоциированную ни с каким конкретным мастер-потоком. Сама арена задач инициализируется не прямо в момент выполнения конструктора, а отложенно, при первом использовании (на иллюстрациях в этой главе показано конструирование объекта, а не стоящей за ним арены задач). Если поток запускает или ставит в очередь задачу на явной арене, до того как он инициализировал собственную неявную арену задач, то это действие расценивается как первое использование планировщика TBB для данного потока, включающее все побочные эффекты: инициализацию его неявной арены задач по умолчанию и, возможно, инициализацию мягкого лимита.

Рекомендации по заданию количества потоков

Комбинация классов `task_scheduler_init`, `task_arena` и `global_control` образует мощный набор инструментов для управления количеством потоков, которые могут участвовать в выполнении параллельной работы в TBB.

Взаимодействие этих объектов может сбить с толку, если оно выходит за рамки ожидаемых паттернов. Поэтому в этом разделе мы сосредоточимся на типичных сценариях и рекомендованных подходах к работе с этими классами. Для простоты на рисунках предполагается, что в системе имеется четыре логических ядра. Тогда библиотека TBB по умолчанию создаст три рабочих потока и на любой арене задач по умолчанию будет четыре слота, один из которых зарезервирован для мастер-потока. На рисунках показано количество потоков в глобальном пуле и количество слотов на аренах задач. Чтобы не загромождать рисунки, мы не показываем назначение рабочих потоков слотам. Стрелки, направленные вниз, обозначают время жизни объектов, а большая буква X – уничтожение объекта.

Использование одного объекта task_scheduler_init в простом приложении

Простейший и, пожалуй, самый распространенный сценарий – приложение с одним главным потоком без явных арен задач. В приложении может использоваться много TBB-алгоритмов, включая вложенный параллелизм, но ему не нужно более одного созданного пользователем потока, т. е. главного потока. Если мы не предпринимаем никаких действий по управлению количеством потоков, подведомственных TBB, то для главного потока будет создана неявная арена задач в момент его первого взаимодействия с планировщиком посредством запуска задачи, выполнения TBB-алгоритма или использования потокового графа. При создании этой арены задач в глобальный пул потоков будет помещено на один поток меньше, чем количество логических ядер в системе. Этот простейший случай, когда вся инициализация производится по умолчанию, показан на рис. 11.5 (для 4-ядерной системы).

В файле `ch11/fig_11_05.cpp` на Github имеется пример программы, снабженной диагностической печатью, так чтобы было видно, сколько потоков участвует в выполнении каждого участка кода. Во многие примеры из этой главы включена аналогичная печать. Соответствующие предложения отсутствуют в исходном коде на рисунках, но есть в коде на Github. В 4-ядерной системе эта программа напечатает:

```
There are 4 logical cores.
4 threads participated in 1st pfor
4 threads participated in 2nd pfor
4 threads participated in flow graph
```

Если мы хотим получить другое поведение в этом простейшем сценарии, то для управления количеством потоков достаточно класса `task_scheduler_init`. Нужно лишь создать объект `task_scheduler_init` до первого использования задач TBB и передать ему желаемое количество потоков. Пример приведен на рис. 11.6. При конструировании этого объекта создается планировщик задач, в глобальный пул (рынок) помещается нужное число потоков (по крайней мере, достаточное для занятия слотов на арене задач[1]), конструируется одна

[1] Это некоторое упрощение реальной картины. См. врезку «Мягкий и жесткий лимиты» выше.

арена для главного потока с запрошенным числом слотов. Этот планировщик TBB уничтожается одновременно с уничтожением единственного объекта `task_scheduler_init`.

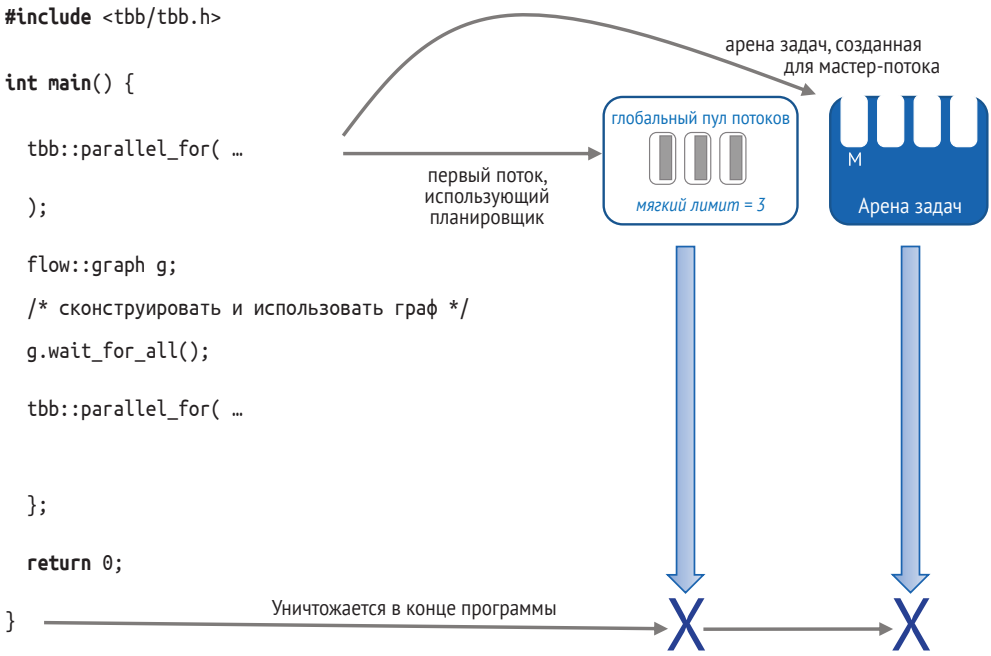


Рис. 11.5 ❖ Инициализация по умолчанию глобального пула потоков и одной арены задач для главного потока

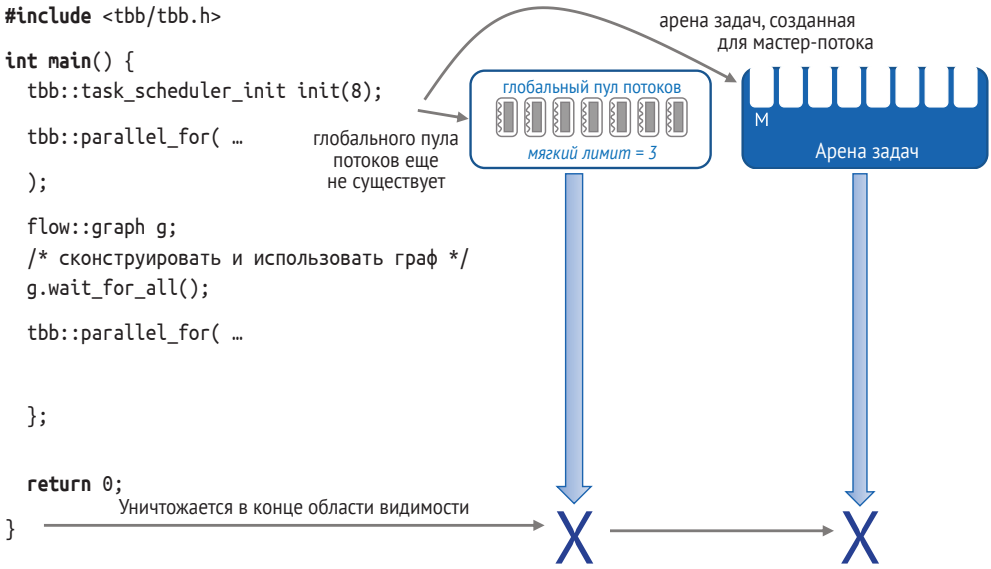


Рис. 11.6 ❖ Использование единственного объекта `task_scheduler_init` в простом приложении

При выполнении кода на рис. 11.6 печатаются такие строки:

```
There are 4 logical cores.
8 threads participated in 1st pfor
8 threads participated in 2nd pfor
8 threads participated in flow graph
```

Примечание. Конечно, статическое задание количества потоков – никуда не годная идея. Мы иллюстрируем возможности на простых примерах с конкретными числами. Но если требуется написать переносимый, неподвластный времени код, то почти никогда не следует использовать конкретные числа.

Использование нескольких объектов task_scheduler_init в простом приложении

Чуть более сложный случай возникает, когда в приложении по-прежнему один главный поток, но мы хотим, чтобы на разных этапах его работы исполнялось разное количество рабочих потоков. При условии что времена жизни объектов `task_scheduler_init` не перекрываются, количество потоков в приложении можно изменять, создавая и уничтожая объекты `task_scheduler_init` с разными значениями `max_threads`. Типичная ситуация, когда это имеет смысл, – эксперименты по масштабируемости. На рис. 11.7 показан цикл, в котором некоторый

```
#include <iostream>
#include <tbb/tbb.h>

void run_test() {
  const int N =
    10*tbb::task_scheduler_init::default_num_threads();
  tbb::parallel_for(0, N, [](int) {
    tbb::tick_count t0 = tbb::tick_count::now();
    while ((tbb::tick_count::now() - t0).seconds() < 0.01);
  });
}

void fig_11_7() {
  const int P =
    tbb::task_scheduler_init::default_num_threads();
  for (int i = 1; i <= P; ++i) {
    tbb::tick_count t0 = tbb::tick_count::now();
    tbb::task_scheduler_init init(i);
    run_test();
    auto sec = (tbb::tick_count::now() - t0).seconds();
    std::cout << "Test using " << i << " threads took "
              << sec << "seconds" << std::endl;
  }
}
```

Рис. 11.7 ❖ Простой цикл с хронометражем, в котором тест исполняется разным количеством потоков – от 1 до P

тест выполняется в разном числе потоков, от 1 до P. Здесь мы последовательно создаем и уничтожаем объекты task_scheduler_init, а стало быть, и планировщики TBB, поддерживающие разное число потоков.

На рис. 11.7 каждый раз, как мы создаем объект task_scheduler_init init, библиотека создает арену задач для главного потока, резервируя один слот для мастер-потока и i-1 дополнительных слотов. Одновременно она задает мягкий лимит и помещает в глобальный пул не менее i-1 рабочих потоков (напомним, что если max_threads < P-1, то все равно в глобальном пуле создается P-1 потоков). При уничтожении init планировщик TBB также уничтожается вместе с единственной ареной задач и глобальным пулом потоков.

Ниже приведен результат прогона примера, в котором run_test() содержит parallel_for с работой на 400 мс:

```
Test using 1 threads took 0.401094seconds
Test using 2 threads took 0.200297seconds
Test using 3 threads took 0.140212seconds
Test using 4 threads took 0.100435seconds
```

Использование нескольких арен с разным числом слотов, чтобы подсказать TBB, куда направлять рабочие потоки

Теперь рассмотрим еще более сложный сценарий, когда требуется несколько арен задач. Чаще всего эта ситуация возникает, когда в приложении имеется более одного потока. Каждый из них является мастером и получает собственную неявную арену. Несколько арен возникает и тогда, когда мы явно создаем арены с помощью класса task_arena, как описано в главе 12. Но независимо от того, как в приложении оказалось несколько арен задач, рабочие потоки распределяются по ним пропорционально количеству слотов. И при этом рассматриваются только те арены, на которых есть готовые к выполнению задачи. Как было отмечено, в этой главе предполагается, что все задачи имеют одинаковый приоритет. Приоритеты, которые могут повлиять на распределение потоков по аренам, подробно рассматриваются в главе 14.

В примере на рис. 11.8 показано три арены задач: две из них созданы для мастер-потоков (главный поток и поток t), а одна (арена a) – явно. Пример искусственный, но код достаточно содержательный, чтобы довести нашу мысль.

На рис. 11.8 мы не пытаемся управлять ни количеством потоков в приложении, ни количеством слотов на аренах задач. Поэтому каждая арена конструируется с количеством слотов по умолчанию, а при создании глобального пула потоков количество потоков в нем тоже берется по умолчанию (рис. 11.9).

Поскольку теперь потоков больше одного, на рис. 11.9 время течет сверху вниз: объекты, расположенные ниже, сконструированы позже объектов, расположенных выше. На рисунке показан один из возможных порядков выполнения, поток t первым запускает задачу, пользуясь шаблоном parallel_for, и, значит, он и создает планировщик TBB и глобальный пул потоков. Как бы сложно ни выглядел пример, поведение корректно определено.

```
void fig_11_8() {
  tbb::task_arena a;

  std::thread t([=]() {
    tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  });

  a.execute([=]() { tbb::parallel_for(0, N,
    [](int) { /* do work */}
  );

  tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  t.join();
}
```

Рис. 11.8 ❖ Приложение с тремя аренами задач: арена по умолчанию для главного потока, явно созданная task_arena a и арена по умолчанию для мастер-потока t

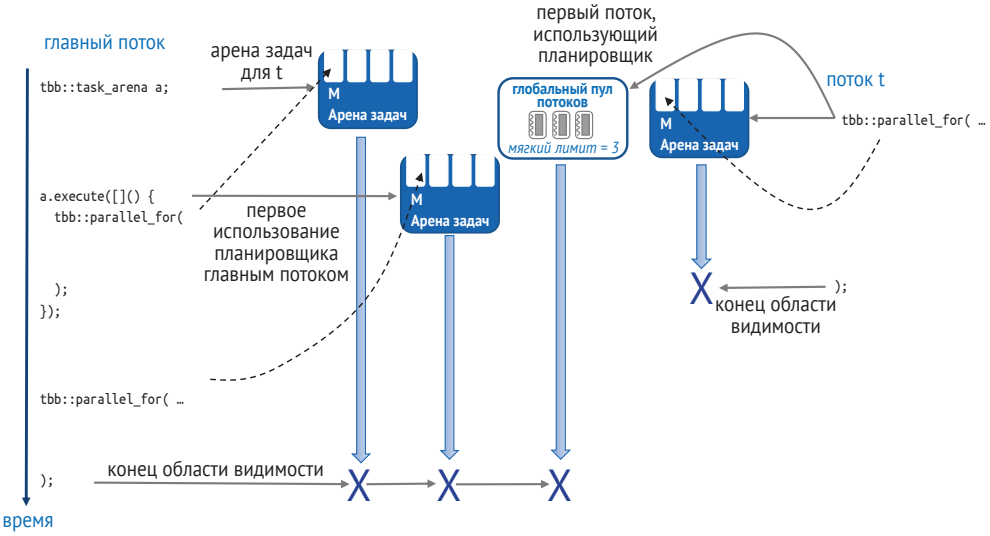


Рис. 11.9 ❖ Возможный порядок выполнения в примере с тремя аренами задач

Как показано на рис. 11.9, выполнение алгоритмов parallel_for в потоке t и на арене задач a может перекрываться. Если это так, то между ними делятся все три потока из глобального пула. Поскольку всего рабочих потоков три, одна арена первоначально получит один поток, а другая – два потока. Какая арена окажется обделенной, определяется библиотекой, но когда на любой из арен кончится работа, потоки могут переместиться на другую арену, чтобы помочь ей справиться с работой. В момент завершения вызова a.execute в главном потоке на рис. 11.9 последний parallel_for исполняется на арене по умолчанию

главного потока, и главный поток занимает ее мастер-слот. Если в этот момент parallel_for для потока t также завершен, то все три рабочих потока могут переместиться на арену главного потока и поработать над последним алгоритмом.

Поведение по умолчанию, показанное на рис. 11.9, вполне разумно. В системе всего четыре логических ядра, поэтому TBB инициализирует глобальный пул с тремя потоками. При создании каждой новой арены задач TBB не увеличивает количество потоков в глобальном пуле, потому что логических ядер осталось столько, сколько и было. Вместо этого все три потока, находящихся в глобальном пуле, динамически разделяются аренами задач.

Библиотека TBB назначает потоки аренам пропорционально количеству слотов. Но мы не обязаны ограничиваться аренами с подразумеваемым числом слотов. Их количеством можно управлять, создавая объект task_scheduler_init для каждого потока приложения и (или) задавая аргумент max_concurrency при создании явных объектов task_arena. На рис. 11.10 предыдущий пример модифицирован соответствующим образом.

```
void fig_11_10() {
  int N = 10*P;
  tbb::task_scheduler_init i4(4);

  tbb::task_arena a(3);

  std::thread t([=]() {
    tbb::task_scheduler_init i2(2);
    tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  });

  a.execute([=]() {
    tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  });

  tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  t.join();
}
```

Рис. 11.10 ❖ Приложение с тремя аренами задач: для арены по умолчанию главного потока max_concurrency равно 4, для явного объекта task_arena a max_concurrency равно 3, а для арены по умолчанию мастер-потока t max_concurrency равно 2

Теперь при выполнении приложения TBB сможет выделить не более одного рабочего потока арене, связанной с потоком t, потому что в ней всего один слот для рабочих потоков. А остальные два потока можно отдать алгоритму parallel_for на арене a. Соответствующий пример выполнения показан на рис. 11.11.

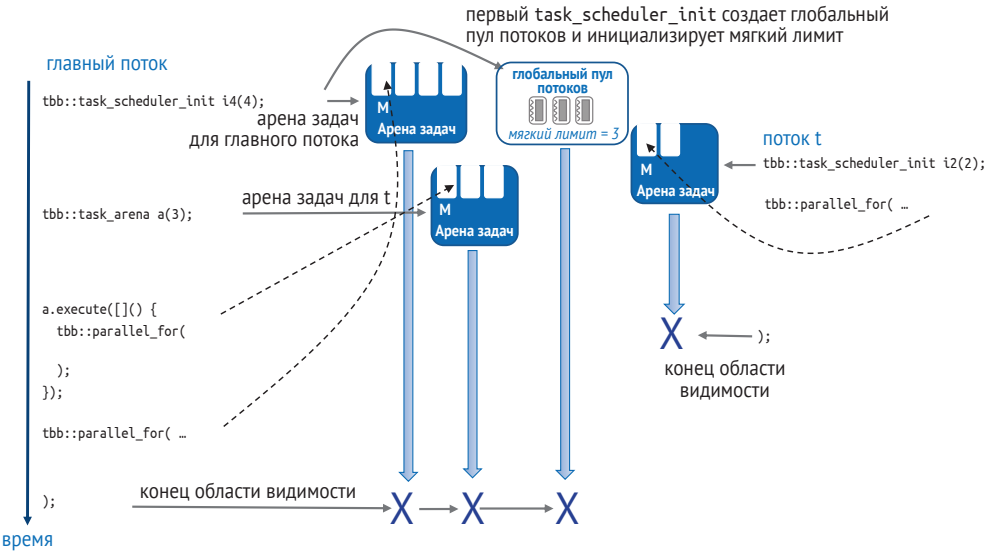


Рис. 11.11 ❖ Возможный порядок выполнения в примере с тремя аренами задач, после того как мы явно задали количество слотов на разных аренах

Если выполнить код этого примера на Github, где имеется диагностическая печать, показывающая, сколько потоков работает на каждом участке программы, то будет напечатано

```
There are 4 logical cores.
3 threads participated in arena pfor
4 threads participated in main pfor
2 threads participated in std::thread pfor
```

Поскольку мы ограничили количество слотов, доступных потоку `t`, остальные потоки не могут перемещаться с арены `task_arena a` на арену потока `t`, после того как закончат свою работу. Ограничивая количество слотов, следует проявлять благоразумие. В этом простом примере мы отдали предпочтение арене `task_arena a`, но одновременно помешали простаивающим потокам помогать потоку `t`.

Мы сумели повлиять на количество слотов, отведенных под потоки на аренах задач, но все равно полагаемся на TBB в части определения общего количества потоков в глобальном пуле, которые будут занимать эти слоты. Если мы хотим изменить и общее количество потоков, то придется обратиться к классу `global_control`.

Использование global_control для управления количеством потоков, доступных для занятия слотов на аренах

Вернемся к примеру из предыдущего раздела, но удвоим количество потоков в глобальном пуле. Новая реализация показана на рис. 11.12.

```
void fig_11_12() {
  const int P =
tbb::task_scheduler_init::default_num_threads();
  int N = 10*P;

  auto mp = tbb::global_control::max_allowed_parallelism;
  int nt = 2*P;
  tbb::global_control gc(mp, nt);

  tbb::task_arena a(3*nt/4);

  std::thread t([=]() {
    tbb::task_scheduler_init i1(nt/4);
    tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  });

  a.execute([=]() {
    tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  });

  tbb::parallel_for(0, N, [](int) { /* выполнить работу */ });
  t.join();
}
```

Рис. 11.12 ❖ Приложение с тремя аренами задач и объектом `global_control`

Теперь мы используем объект `global_control`, чтобы задать количество потоков в глобальном пуле. Напомним, что этот объект влияет на глобальные параметры планировщика; в данном случае мы изменяем параметр `max_allowed_parallelism`. Мы также используем объект `task_scheduler_init` в потоке `t` и аргумент конструктора `task_arena`, чтобы задать максимальное количество потоков, назначаемых каждой арене задач. На рис. 11.13 показан пример выполнения на 4-ядерной машине. Теперь приложение создает семь рабочих потоков (всего восемь, но один из них мастер-поток), и эти потоки неравномерно распределены между явной ареной `task_arena a` и ареной по умолчанию для потока `t`. Поскольку для главного потока мы ничего специально не делали, то последний алгоритм `parallel_for` использует его арену задач по умолчанию с P слотами.

В результате выполнения кода на рис. 11.13 печатается:

```
There are 4 logical cores.
6 threads participated in arena pfor
4 threads participated in main pfor
2 threads participated in std::thread pfor
```

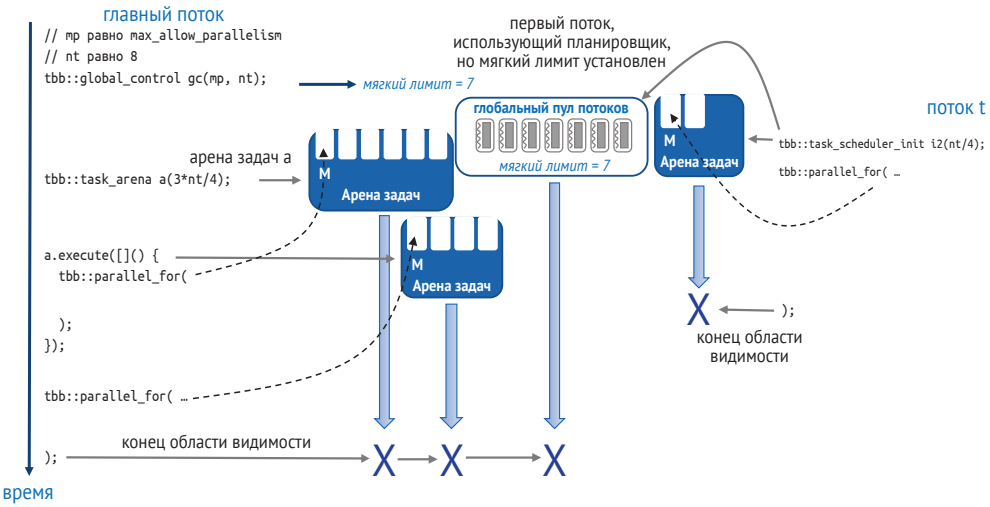


Рис. 11.13 ❖ Возможный порядок выполнения в примере с тремя аренами задач, после того как мы явно задали мягкий лимит с помощью объекта `global_control`

Использование global_control с целью временно ограничить количество доступных потоков

Еще один типичный сценарий – использование объекта `global_control` с целью временно изменить количество потоков на каком-то участке выполнения приложения, как показано на рис. 11.14. В этом примере мастер-поток создает глобальный пул и арену задач, способные поддерживать 12 рабочих потоков, конструируя объект `task_scheduler_init`. Но объект `global_control` используется так, чтобы конкретный алгоритм `parallel_for` мог использовать только семь рабочих потоков. И хотя на протяжении всего приложения арена задач содержит 12 слотов, количество потоков в пуле временно ограничено, поэтому занять эти слоты смогут не более семи рабочих потоков.

Когда какой-нибудь объект `global_control` уничтожается, мягкий лимит пересчитывается с использованием оставшихся объектов. Поскольку в данном случае таковых не осталось, то восстанавливается мягкий лимит по умолчанию. Об этом, быть может, неожиданном поведении необходимо помнить, потому что если мы хотим, чтобы в глобальном пуле осталось 11 потоков, то следует создать внешний объект `global_control`. Это показано на рис. 11.15.

На рис. 11.14 и 11.15 нельзя использовать объект `task_scheduler_init` для временного изменения количества потоков, потому что в главном потоке уже существует арена задач. Если создать еще один объект `task_scheduler_init` во внутренней области видимости, то будет лишь увеличен счетчик ссылок на эту арену, а не создана новая. Поэтому мы и используем объект `global_control` для ограничения количества доступных потоков, вместо того чтобы уменьшить количество слотов на арене.

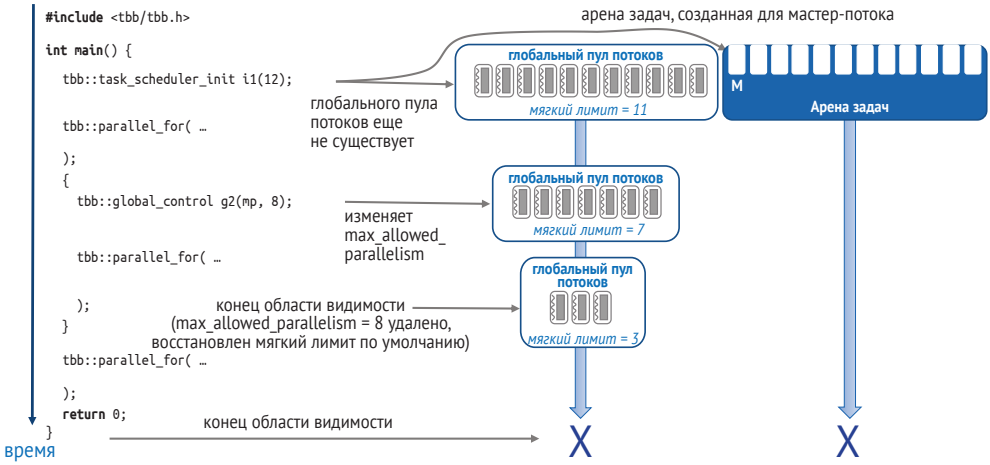


Рис. 11.14 ❖ Использование объекта `global_control` для временного изменения количества потоков, доступных конкретному экземпляру алгоритма, с последующим возвратом к значению по умолчанию

В результате выполнения кода на рис. 11.14 будет напечатано:

```
There are 4 logical cores.
12 threads participated in 1st pfor
8 threads participated in 2nd pfor
4 threads participated in 3rd pfor
```

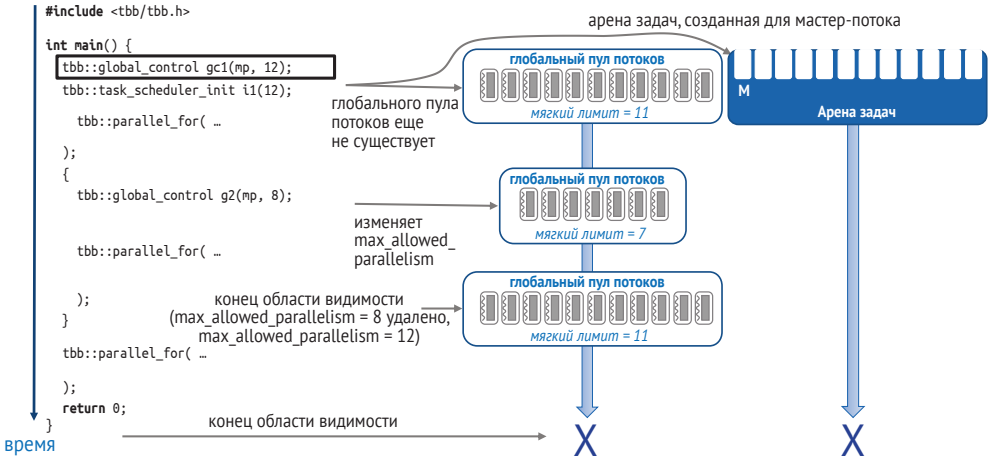


Рис. 11.15 ❖ Использование объекта `global_control` для временного изменения количества потоков, доступных конкретному экземпляру алгоритма

Когда НЕ следует управлять количеством потоков

При реализации подключаемого модуля (плагина) или библиотеки лучше отказаться от использования объектов `global_control`. Они влияют на глобальные параметры, поэтому модуль или библиотечная функция изменят количество потоков, доступных всем компонентам приложения. Принимая во внимание локальную природу модуля или библиотеки, вряд ли это будет то, чего вы хотели. На рис. 11.14 мы временно изменили количество потоков в глобальном пуле. Если бы то же самое было сделано внутри библиотечной функции, то повлияло бы на количество потоков, доступных не только данной арене задач, но и всем аренам в приложении. Откуда библиотечной функции знать, правильно ли это? Скорее всего, ниоткуда.

Мы рекомендуем не выносить в библиотеку манипуляции с глобальными параметрами, а оставить это главной программе. Разработчики приложений, допускающих подключаемые модули, должны предельно четко сообщить авторам модулей, какая стратегия параллельного выполнения применяется в приложении, чтобы те могли учитывать ее при реализации модулей.

Задание размера стека рабочих потоков

Классы `task_scheduler_init` и `global_control` можно использовать и для задания размера стека рабочих потоков. Взаимодействия между объектами такие же, как при задании числа потоков, с одним исключением. Если имеется более одного объекта `global_control`, задающего размер стека, то в качестве окончательного размера берется максимальное, а не минимальное из запрошенных значений.

Второй аргумент конструктора `task_scheduler_init` называется `thread_stack_size`. Значение 0, подразумеваемое по умолчанию, говорит планировщику, что нужно использовать умолчание, принятое на данной платформе. В противном случае используется указанное значение.

Конструктор объекта `global_control` принимает параметр и его значение. Если аргумент `parameter` содержит `thread_stack_size`, то объект изменяет глобальное значение размера стека. В отличие от значения `max_allowed_paralleism`, окончательное глобальное значение `thread_stack_size` равно максимуму из всех запрошенных значений.

Зачем изменять размер стека по умолчанию?

Стек потока должен быть достаточно большим для всего, что планируется в нем хранить, включая все локальные переменные вложенных вызовов функций. Принимая решение о размере стека, мы должны учитывать локальные переменные в телах задач, а также тот факт, что рекурсивное выполнение деревьев задач может привести к глубокой рекурсии, особенно если собираемся реализовать собственные алгоритмы с блокирующими задачами. Если не помните, почему этот подход может стать причиной взрывного роста стека, вернитесь к разделу «Низкоуровневый интерфейс задач: часть первая – блокировка задач» главы 10.

Поскольку правильный выбор размера стека зависит от приложения, хороших эвристических правил не существует. TBB берет значение по умолчанию, зависящее от ОС, и это, наверное, лучшее, что можно предположить о типичных потребностях потока.

Что не так?

Классы task_scheduler_init, task_arena и global_control постепенно вводились в библиотеку TBB для решения конкретных проблем. На ранних этапах развития TBB, когда параллельных приложений было мало, а те, что были, зачастую работали с одним потоком приложения, класса task_scheduler_init вполне хватало. Класс task_arena помогал организовать изоляцию в приложениях, становящихся все более сложными. А класс global_control дал пользователям возможность точнее контролировать глобальные параметры библиотеки, чтобы еще лучше справляться с возрастающей сложностью. К сожалению, эти средства создавались в разное время, а не как часть единого дизайна. В результате, если они используются за рамками описанных выше сценариев, поведение иногда противоречит интуиции, хотя и определено корректно.

Есть два самых частых источника недоразумений: (1) непонимание того, когда создается планировщик TBB по умолчанию, и (2) состояние гонки при задании мягкого лимита для глобального пула потоков.

При создании объекта task_scheduler_init либо создается планировщик TBB, либо увеличивается счетчик ссылок на уже существующий планировщик. Какие интерфейсы в библиотеке TBB приводят к первому использованию планировщика, запомнить трудно. Совершенно очевидно, что выполнение любого TBB-алгоритма, применение потокового графа или запуск задач – примеры использования планировщика. Но, как было отмечено выше, даже выполнение задач на явной арене рассматривается как первое использование планировщика, которое оказывает влияние не только на явную арену, но и, возможно, на неявную арену задач вызывающего потока. А как насчет использования поточно-локальной памяти или конкурентных контейнеров? Они не считаются. Что можно посоветовать, кроме как внимательно читать о последствиях используемых интерфейсов в документации? Если количество потоков в приложении расходится с ожиданиями – особенно если используется количество потоков по умолчанию, хотя вы думаете, что изменили умолчание, – поищите места, в которых мог быть непреднамеренно инициализирован планировщик TBB по умолчанию.

Вторая частая причина недоразумений – гонка за установку мягкого лимита количества доступных потоков. Например, если два потока приложения исполняются параллельно и оба создают объект task_scheduler_init, то только первый созданный установит мягкий лимит. На рис. 11.16 два потока конкурентно работают в одном приложении, и оба создают объекты task_scheduler_init – один просит установить max_threads=4, а другой – max_threads=8. Что при этом происходит с аренами задач, понятно: каждый мастер-поток получает собственную арену с запрошенным числом слотов. Но что, если мягкий лимит количества потоков в глобальном пуле к этому моменту еще не был установлен? Сколько потоков TBB поместит в глобальный пул: 3, 7, 3 + 7 = 10, P – 1 или …?

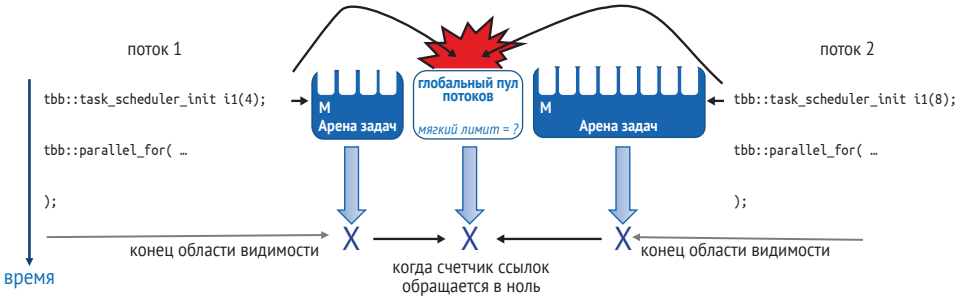


Рис. 11.16 ❖ Конкурентное использование двух объектов `task_scheduler_init`

Как следует из приведенного выше описания работы `task_scheduler_init`, ни одно из этих предположений неверно. Будет установлено значение, заданное в объекте, который создан первым. Да-да, это не опечатка! Если поток 1 создаст свой объект `task_scheduler_init` раньше, то мы получим планировщик с глобальным пулом, в котором всего три рабочих потока. А если первым окажется поток 2, то мы получим глобальный пул с семью потоками. Наши два потока могут разделять три или семь рабочих потоков – все зависит от того, кто победит в гонке за создание планировщика TBB!

Но впадать в отчаяние не стоит: почти все потенциальные проблемы, сопутствующие заданию количества потоков, можно разрешить, обратившись к типичным паттернам использования, описанным выше в этой главе. Например, если мы знаем, что в приложении возможна гонка, показанная на рис. 11.16, то можем сделать наши намерения кристально ясными, задав мягкий лимит в главном потоке с помощью объекта `global_control`.

Резюме

В этой главе мы сначала кратко повторили уже известные сведения о планировщике TBB, а затем ввели три класса, применяемых для управления количеством потоков параллельного выполнения: `task_scheduler_init`, `task_arena` и `global_control`. Далее мы описали типичные ситуации, когда требуется управлять количеством потоков в параллельных алгоритмах. Мы начали с простых приложений с единственным главным потоком, а затем рассмотрели более сложные случаи, когда имеется несколько мастер-потоков и несколько арен задач. В заключение мы отметили, что хотя при использовании этих классов можно столкнуться с подвохами, их легко избежать, если явно формулировать свои намерения, а не полагаться на поведение по умолчанию или на исход гонки.

Глава 12

Применение изоляции работы для обеспечения корректности и повышения производительности

Всякий, кому доводилось иметь дело с детьми (или кто ведет себя как ребенок), знает, что иногда единственный способ не дать детям покалечить друг друга – развести их. То же самое можно сказать о задачах и алгоритмах TBB. Если задачи или алгоритмы не могут ужиться друг с другом, мы разведем их, воспользовавшись *изоляцией работ*.

Например, при использовании вложенного параллелизма нам нужно (в некоторых редких ситуациях) организовать изоляцию работ, чтобы обеспечить корректность. В этой главе мы проработаем сценарии, в которых такая необходимость возникает, а затем предложим набор правил для определения того, требуется ли изоляция для корректности. Мы также опишем, как функция `isolate` используется для изоляции работ.

Есть и другие случаи, когда мы хотим изолировать работы, чтобы, по соображениям производительности, задачи могли исполняться не где угодно, а только на явной арене задач. В таких случаях изоляция – обоюдоострое оружие. С одной стороны, мы сможем контролировать, сколько потоков будет участвовать в исполнении задач на разных аренах, с целью отдать предпочтение одним задачам перед другими или использовать точки подключения в библиотеке TBB для привязки потоков к конкретным ядрам ради оптимизации локальности. С другой стороны, явные арены задач затрудняют потокам участие в работах вне той арены, которой они в данный момент назначены. Мы обсудим, как использовать класс `task_arena`, когда потребность в изоляции связана с производительностью. Хотим, однако, предупредить, что хотя класс `task_arena` можно использовать также для изоляции с целью гарантирования корректности, это нежелательно, т. к. сопряженные с ним накладные расходы слишком велики.

Изоляция работ – ценная возможность, когда она действительно нужна и используется должным образом, но, как мы увидим в этой главе, применять ее нужно с осторожностью.

Изоляция работ для обеспечения корректности

Планировщик TBB спроектирован так, чтобы обеспечить занятость рабочих потоков и ядер, на которых они исполняются. Если рабочий поток оказывается без работы, он заимствует работу у другого потока. В момент заимствования поток не знает, какой параллельный алгоритм, цикл или функция первоначально создали задачу, которую он забрал себе. Обычно происхождение задачи не играет роли, поэтому лучшее, что может сделать библиотека TBB, – обращаться со всеми задачами одинаково и обрабатывать их как можно быстрее.

Но если в приложении используется вложенный параллелизм, то заимствование задач может изменить порядок выполнения, ожидаемый разработчиком. Этот порядок выполнения сам по себе не таит опасностей; в большинстве случаев он именно такой, как нам нужно. Но если сделать неправильные предположения о том, как могут исполняться задачи, то можно получить паттерн, который ведет к неожиданным или даже катастрофическим последствиям.

На рис. 12.1 приведен простой пример, демонстрирующий существо проблемы. В коде имеется два цикла `parallel_for`. В теле внешнего цикла захватывается мьютекс `m`. Поток, захвативший мьютекс, вызывает вложенный цикл `parallel_for`, не освободив мьютекс. Проблема возникнет, если поток, захвативший `m`, перейдет в состояние простоя до того, как завершится внутренний цикл. Это может случиться, если рабочие потоки позаимствовали итерации, но еще не завершили их, когда у мастер-потока кончилась работа. Мастер-поток не может просто выйти из `parallel_for`, потому что цикл еще не завершен. Ради эффективности поток не может также активно ожидать, пока другие потоки завершат свою работу, ибо кто знает, сколько времени это займет. Вместо этого он сохраняет текущую задачу в своем стеке и ищет дополнительную работу, чтобы чем-то занять себя в ожидании момента, когда он сможет вернуться к прерванному занятию. Если такая ситуация возникнет в коде на рис. 12.1, то в точке, где поток ищет, какую работу позаимствовать, будет присутствовать два вида задач – из внутреннего и из внешнего циклов. Если поток позаимствует и начнет выполнять задачу из внешнего `parallel_for`, то он попытается снова захватить мьютекс `m`. Поскольку он уже захватил его, а мьютекс `tbb::spin_mutex` не рекурсивный, мы имеем взаимоблокировку. Поток загнал себя в ловушку – он ждет, когда сам же освободит мьютекс!

После знакомства с этим примером обычно возникают два вопроса: (1) что, кто-то и вправду пишет такой код? (2) действительно ли поток может позаимствовать задачу из внешнего цикла? Увы, ответ на оба вопроса утвердительный.

Такой код и в самом деле встречается – правда, почти всегда непреднамеренно. В частности, это может случиться, если удерживать блокировку на время вызова библиотечной функции. Программист полагает, будто знает, что функция делает, но, не будучи знаком с реализацией, может ошибаться. Если библиотечная функция содержит вложенный параллелизм, то может сложиться ситуация, показанная на рис. 12.1.

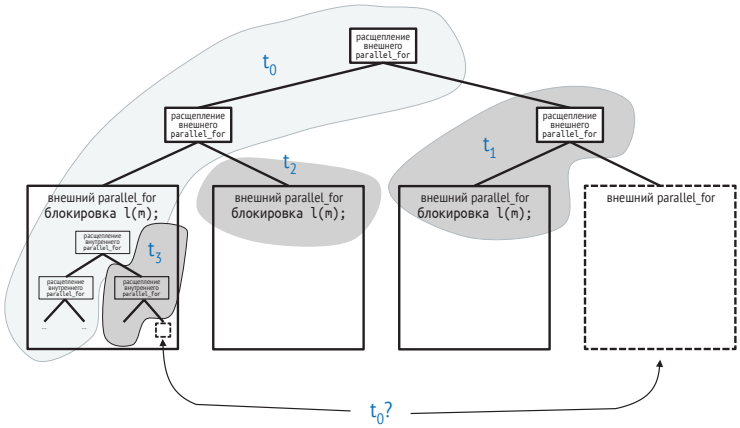
И да, заимствование работы может стать причиной взаимоблокировки в этом примере. На рис. 12.2 показано, как можно попасть в эту кошмарную ситуацию.

```
#include <tbb/tbb.h>

void doWork();

void fig_12_1() {
  tbb::spin_mutex m;
  const int P =
    tbb::task_scheduler_init::default_num_threads();

  tbb::parallel_for(0, P,
    [&m](int) {
      const int N = 1000;
      tbb::spin_mutex::scoped_lock l{m};
      tbb::parallel_for(0, N, [](int j) { doWork();});
    }
  );
}
```

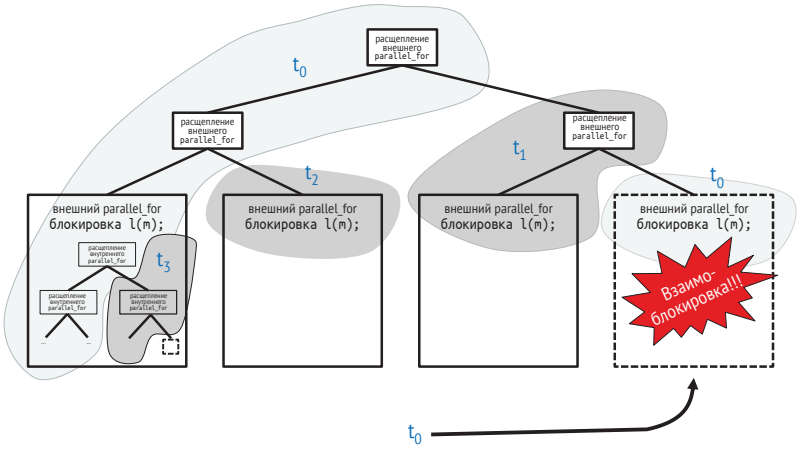
Рис. 12.1 ❖ Удержание блокировки на время выполнения вложенного `parallel_for`

На рис. 12.2(a) поток t_0 входит во внешний цикл и захватывает мьютекс `m`. Затем поток t_0 входит во вложенный `parallel_for` и выполняет левую половину пространства итераций. Пока поток t_0 занят, другие три потока, t_1, t_2 и t_3, принимают участие в выполнении задач на арене. Потоки t_1 и t_2 заимствуют итерации внешнего цикла и блокируются в ожидании возможности захватить мьютекс `m`, который удерживает t_0. Тем временем поток t_3 случайным образом выбирает t_0 в качестве жертвы и начинает выполнять правую половину его внутреннего цикла. Здесь-то и начинается интересное. Поток t_0 завершает левую половину итераций внутреннего цикла и хочет позаимствовать работу, чтобы не бездельничать. В этот момент у него есть два варианта действий: (1) если он случайным образом выберет в качестве жертвы t_3, то начнет исполнять дополнительные итерации собственного внутреннего цикла, а (2) если он случайным образом выберет в качестве жертвы t_1, то будет выполнять итерации внешнего цикла. Напомним, что по умолчанию планировщик обращается со всеми задачами одинаково, поэтому ни один вариант не является предпочтительным. На рис. 12.2(b) показан неудачный выбор, когда t_0 заимствует работу и t_1 оказывается в ситуации взаимоблокировки, пытаясь захватить мьютекс, который и так уже удерживает, поскольку внешняя задача все еще находится в его стеке.

На рис. 12.3 приведен другой пример, иллюстрирующий проблемы корректности. Мы снова видим два вложенных цикла `parallel_for`, но вместо взаимоблокировки получаем неожиданный результат из-за использования поточно-локальной памяти. Каждая задача записывает некоторое значение в ячейку поточно-локальной памяти `local_i`, затем выполняет внутренний цикл `parallel_for` и читает эту ячейку. Из-за внутреннего цикла поток может позаимствовать работу, если начнет простаивать, запишет другое значение в поточно-локальную память и вернется к внешней задаче.



(a) t_0 может заимствовать внешнюю или внутреннюю задачу



(b) t_0 заимствует внешнюю задачу – взаимоблокировка

Рис. 12.2 ❖ Потенциальный порядок выполнения дерева задач кодом на рис. 12.1

Разработчики TBB употребляют термин *шабашка* (moonlighting)[1] для описания ситуаций, когда поток имеет незаконченные дочерние задачи и при этом заимствует не связанные с ними задачи, чтобы чем-то занять себя. Обычно шабашка – это хорошо, поскольку потоки не сидят без дела. И лишь в редких ситуациях она может обернуться злом. В обоих наших примерах было сделано неправильное предположение. Мы предположили – и это неудивительно, – что поскольку в TBB планировщик невытесняющий, то не бывает так, что некоторый поток, выполняющий внутреннюю задачу, не закончив ее, приступает к выполнению внешней. Но, как мы видели, такая ситуация возможна, поскольку поток может заимствовать работу, когда чего-то ждет во внутреннем

[1] В словаре Collins Dictionary слово *moonlighting* определяется как дополнительная работа, особенно по ночам и часто незаконно.

параллельном цикле. Обычно такое поведение можно только приветствовать, но оно становится опасным, если мы по ошибке создали взаимоисключающую зависимость от потока, выполняющего задачи. В первом случае блокировка удерживалась во время выполнения вложенного параллелизма, что позволило потоку приостановить внутреннюю задачу и забрать внешнюю. Во втором случае поток обращался к поточно-локальной памяти до и после вложенного параллелизма, а мы предположили, что в промежутке он не шабашничал.

```
#include <tbb/tbb.h>

void doWork();

thread_local int local_i = -1;

void fig_12_3() {
  const int P =
tbb::task_scheduler_init::default_num_threads();

  tbb::parallel_for(0, P,
    [](int i) {
      const int N = 1000;
      local_i = i;
      tbb::parallel_for(0, N, [](int j) {doWork();});
      if (local_i != i) {
        std::cout << "Unexpected local_i!!! ";
      }
    }
  );
  std::cout << std::endl;
}
```

Рис. 12.3 ❖ Вложенный параллелизм может давать неожиданные результаты из-за использования поточно-локальной памяти

Как видим, примеры различны, но ошибка в них общая. Алексей Катранов разместил в своем блоге статью «The Work Isolation Functionality in Intel Threading Building Blocks» (она упомянута в разделе «Дополнительная информация), в которой приводит контрольный список из трех пунктов для решения о том, нужна ли изоляция работ для обеспечения корректности.

1. Используется ли вложенный параллелизм (пусть даже косвенно, вследствие использования функций из сторонней библиотеки)? Если нет, изоляция не нужна, иначе перейти к следующему шагу.
2. Будет ли безопасно, если поток повторно войдет в параллельную задачу внешнего уровня (как если бы имела место рекурсия)? Сохранение значения в поточно-локальной памяти, повторный захват мьютекса, уже захваченного этим потоком, или других ресурсов, которые не должны использоваться потоком повторно, – все это может вызывать проблемы.

Если повторное вхождение безопасно (поток реентерабелен), то изоляция не нужна, иначе перейти к следующему шагу.

3. Изоляция необходима. Вложенный параллелизм должен находиться в изолированном регионе.

Создание изолированного региона с помощью this_task_arena::isolate

Когда изоляция нужна для обеспечения корректности, можно использовать одну из функций `isolate` в пространстве имен `this_task_arena`:

```
namespace this_task_arena {
  // Поддерживались до C++11
  template<typename F> void isolate(F& f);
  template<typename F> void isolate(const F& f);

  // Поддерживаются начиная с C++11
  template<typename F> auto isolate(F& f) -> decltype(f());
  template<typename F> auto isolate(const F& f) ->
    decltype
}
```

На рис. 12.4 показано, как использовать эту функцию для добавления изолированного региона, окружающего вложенный цикл `parallel_for` на рис. 12.1.

```
#include <tbb/tbb.h>

void doWork();

void fig_12_4() {
  tbb::spin_mutex m;
  const int P =
tbb::task_scheduler_init::default_num_threads();

  tbb::parallel_for(0, P,
    [&m](int i) {
      const int N = 1000;
      tbb::spin_mutex::scoped_lock l{m};
      tbb::this_task_arena::isolate(
        [i]() {
          tbb::parallel_for(0, N, [](int j) {doWork();});
        }
      );
    }
  );
}
```

Рис. 12.4 ❖ Использование функции `isolate` для предотвращения шабашки в случае вложенного параллелизма

Если поток находится в изолированном регионе и оказывается без работы, поскольку должен чего-то ждать (например, так бывает в конце вложенного `parallel_for`), то ему разрешено заимствовать задачи только внутри своего изолированного региона. Это решает проблему взаимоблокировки, поскольку если поток хочет позаимствовать задачу, ожидая во внутреннем цикле `parallel_for` на рис. 12.4, то ему запрещено заимствовать задачу из внешнего цикла.

Если поток блокируется внутри изолированного региона, то он по-прежнему может случайным образом выбрать поток на своей арене задач в качестве жертвы, но теперь должен проверять задачи в двусторонней очереди этого потока и отбирать только те, которые происходят из его собственного изолированного региона.

Основные свойства функции `task_arena::isolate` также перечислены в блоге Алексея.

- Изоляция налагает ограничения только на потоки, которые входят или присоединяются к изолированному региону. Рабочие потоки вне изолированного региона могут брать любую задачу, в т. ч. запущенную в изолированном регионе.
- Когда поток без изоляции исполняет задачу, запущенную в изолированном регионе, он присоединяется к региону этой задачи и становится изолированным вплоть до завершения задачи.
- Потоки, ожидающие внутри изолированного региона, не могут обрабатывать задачи, запущенные в других изолированных регионах (т. е. все регионы взаимно изолированы). Кроме того, если поток внутри изолированного региона заходит во вложенный изолированный регион, то он не может обрабатывать задачи из внешнего изолированного региона.

Только не это! Изоляция работ может порождать собственные проблемы с корректностью!

К сожалению, применять изоляцию работ без разбора тоже не стоит. Имеются проблемы с производительностью, но о них мы скажем позже, а гораздо важнее то, что сама изоляция при неправильном применении может стать причиной взаимоблокировки! И мы приходим к тому, с чего начали…

В частности, нужна особая внимательность, когда изоляция работ сочетается с интерфейсами TBB, которые отделяют запуск задач от ожидания их завершения, например `task_group` и потоковые графы. Задача, вызывающая интерфейс ожидания в одном изолированном регионе, не может участвовать в задачах, запущенных в другом изолированном регионе во время ожидания. Если в таком положении оказывается достаточно много потоков, то у приложения могут кончиться потоки, и всякое движение вперед прекратится. В качестве примера рассмотрим функцию на рис. 12.5. В функции `splitRunAndWait` M задач запущены в группе `task_group tg`. Но все запуски произведены в *разных* изолированных регионах.

Если функция `fig_12_5` написана как на рис. 12.5, то никаких проблем не будет. Обращение к `tg.wait` в `splitRunAndWait` не находится внутри самого изолированного региона, поэтому мастер-поток и рабочие потоки могут прийти на помощь разным изолированным регионам и по завершении переместиться в другие.

Но что, если изменить вызывающую функцию, как показано на рис. 12.6?

```
#include <tbb/tbb.h>

void doWork();

const int M =
2*tbb::task_scheduler_init::default_num_threads();

void splitRunAndWait() {
  tbb::task_group tg;

  for ( int i=0; i<M; ++i ) {
    tbb::this_task_arena::isolate( [&tg]{
      // run в изолированном регионе
      tg.run( []{
        const int N = 10000;
        tbb::parallel_for(0,N,[](int i) {doWork();});
      });
    });
  }
  tg.wait();
}

void fig_12_5() {
  tbb::parallel_for( 0, M, []( int ) {
    splitRunAndWait();
  } );
}
```

Рис. 12.5 ❖ Функция, вызывающая `run` и `wait` для группы задач `tg`. Вызов `run` произведен внутри изолированного региона

Теперь все обращения к `splitRunAndWait` производятся из разных изолированных регионов, а затем в этих изолированных регионах производятся обращения к `tg.wait`. Каждый поток, вызывающий `tg.wait`, должен ждать завершения `tg`, но не может позаимствовать ни одну задачу, принадлежащую `tg` или любой другой группе задач, потому что все они запущены из других изолированных регионов! Если M достаточно велико, то в конечном итоге все имеющиеся потоки застрянут в вызовах `tg.wait`, и не останется ни одного потока для выполнения задач. Налицо взаимоблокировка.

```
#include <tbb/tbb.h>

void doWork();

const int M =
2*tbb::task_scheduler_init::default_num_threads();

void splitRunAndWait() {
  tbb::task_group tg;

  for ( int i=0; i<M; ++i ) {
    tbb::this_task_arena::isolate( [&tg]{
      // run в изолированном регионе
      tg.run( []{
        const int N = 10000;
        tbb::parallel_for(0,N,[](int i) {doWork();});
      });
    });
  }
  tg.wait();
}

void fig_12_6() {
  tbb::parallel_for( 0, M, []( int ) {
    tbb::this_task_arena::isolate( [] {
      splitRunAndWait();
    });
  });
}
```

Рис. 12.6 ❖ Функция, вызывающая `run` и `wait` для группы задач `tg`. Вызов `run` произведен внутри изолированного региона

Если используется интерфейс, отделяющий запуск от ожидания, то этой проблемы можно избежать, организовав программу так, чтобы ожидание всегда производилось в том же изолированном регионе, откуда запускаются задачи. Код на рис. 12.6 можно было переписать, перенеся обращение к `run` во внешний регион, как показано на рис. 12.7.

```
#include <tbb/tbb.h>

void doWork();

const int M =
2*tbb::task_scheduler_init::default_num_threads();

void splitRunAndWait() {
  tbb::task_group tg;

  for ( int i=0; i<M; ++i ) {
    tg.run( [&tg]{
      tbb::this_task_arena::isolate( [&tg]{
       // run в изолированном регионе
        const int N = 10000;
        tbb::parallel_for(0,N,[](int i) {doWork();});
      });
    });
  }
  tg.wait();
}

void fig_12_7() {
  tbb::parallel_for( 0, M, []( int ) {
    tbb::this_task_arena::isolate( [] {
      splitRunAndWait();
    });
  });
}
```

Рис. 12.7 ❖ Функция, вызывающая `run` и `wait` для группы задач `tg`. Вызовы `run` и `wait` теперь находятся вне изолированного региона

Теперь, даже если в главной функции используется параллельный цикл и изоляция, проблемы не возникнет, потому что каждый поток, вызывающий `tg.wait`, сможет исполнять задачи из своей группы `tg`.

Даже безопасная изоляция работ не бесплатна

Мало того что изоляция работ способна привести к взаимоблокировке, так она еще и обходится не бесплатно с точки зрения производительности. Поэтому даже когда ее использование безопасно, нужно дважды подумать, стоит ли это делать. Поток, который не находится в изолированном регионе, может заимствовать любую задачу, т. е. он может быстро выбрать самую старую задачу из двусторонней очереди потока-жертвы. Если у жертвы вообще нет задач, то можно сразу же выбрать другую жертву. Однако задачи, запущенные в изолированном регионе, и их потомки помечены регионом, которому принадлежат. Поток, который выполняется в изолированном регионе, обязан просмотреть

очередь жертвы и найти в ней самую старую задачу, принадлежащую своему региону, а вовсе не любую старую задачу. И о том, что у жертвы нет подходящих задач, поток узнает, только изучив все задачи в ее очереди и не найдя ни одной из своего региона. И лишь тогда можно выбрать другую жертву. Потоки, заимствующие задачи из своего изолированного региона, обречены на более высокие накладные расходы, поскольку должны быть разборчивыми!

Использование арен задач для изоляции: обоюдоострый меч

Изоляция работ ограничивает возможности потока, ищущего для себя работу. Мы можем изолировать работу с помощью функции `isolate`, как описано в предыдущем разделе, или же воспользоваться классом `task_arena`. Часть интерфейса этого класса, относящаяся к теме данной главы, показана на рис. 12.8.

```
class task_arena {
public:
    task_arena(int max_concurrency = automatic,
               unsigned reserved_for_masters = 1);
    task_arena(const task_arena &s);
    explicit task_arena(task_arena::attach);
    ~task_arena();

    int max_concurrency() const;

    // поддерживается начиная с C++11
    template<typename F> auto execute(F& f) -> decltype(f());
    template<typename F> auto execute(const F& f) ->
      decltype(f());
    template<typename F> void enqueue(F&& f);
    template<typename F> void enqueue(F&& f, priority_t p);
};
```

Рис. 12.8 ❖ Подмножество открытого интерфейса класса `task_arena`

Почти никогда не имеет смысла использовать класс `task_arena` вместо функции `isolate`, чтобы организовать изоляцию только для обеспечения корректности. И тем не менее у него есть важные применения. Мы рассмотрим основные черты класса `task_arena` и попутно расскажем о его плюсах и минусах.

С помощью конструктора `task_arena` мы можем задать общее количество слотов для потоков на арене (аргумент `max_concurrency`) и количество слотов, зарезервированных исключительно для мастер-потоков (аргумент `reserved_for_masters`). Дополнительные сведения о том, как класс `task_arena` используется для управления количеством потоков, см. в главе 11.

На рис. 12.9 приведен простой пример, в котором создается одна арена задач `ta2` с `max_concurrency=2` и задача, выполняющая `parallel_for` на этой арене.

```
void fig_12_9() {
  tbb::task_arena ta2{2};
  ta2.execute([&]() {
    const int N = 1000;
    tbb::parallel_for(0, N, [](int) {doWork();});
  });
}
```

Рис. 12.9 ❖ Арена задач с максимальной степенью конкурентности 2

Когда поток вызывает метод `execute` арены задач, он пытается присоединиться к арене в качестве мастер-потока. Если доступных слотов нет, он ставит задачу в очередь к этой арене. В противном случае поток присоединяется к арене и выполняет на ней задачу. На рис. 12.9 поток присоединяется к арене `ta2`, начинает цикл `parallel_for`, а затем принимает участие в выполнении задач, запускаемых внутри этого цикла. Поскольку параметр `max_concurrency` этой арены равен 2, к ней может присоединиться и принять участие в выполнении задач не более одного дополнительного рабочего потока. Если выполнить снабженный диагностической печатью вариант этого примера, имеющийся на Github, то будет напечатано:

```
There are 4 logical cores.
2 threads participated in ta2
```

Мы уже видим различия между `isolate` и классом `task_arena`. Действительно, только потоки на арене `ta2` смогут исполнять запущенные на ней задачи, поэтому изоляция работ присутствует, но мы также смогли задать максимальное число потоков, которые могут принимать участие в выполнении вложенного `parallel_for`.

На рис. 12.10 сделан еще один шаг: созданы две арены задач, одна с `max_concurrency=2`, другая – с `max_concurrency=6`. Затем алгоритм `parallel_invoke` используется для создания двух задач, одна из которых выполняет `parallel_for` на арене `ta2`, а другая – `parallel_for` на арене `ta6`. В обоих циклах `parallel_for` количество итераций одинаково, как и время выполнения одной итерации.

По существу, мы разделили восемь потоков на две группы, позволив двум параллельно работать на арене `ta2`, а шести – на арене `ta6`. Зачем бы это могло понадобиться? Например, потому что мы считаем работу на арене `ta6` более важной.

Выполнив код на рис. 12.10 на платформе с 8 аппаратными потоками, мы увидим примерно такой результат:

```
ta2_time == 0.500409
ta6_time == 0.169082

There are 8 logical cores.
2 threads participated in ta2
6 threads participated in ta6
```

В этом и состоит ключевое различие между организацией изоляции с помощью `isolate` и `task_arena`. При использовании арены задач нас почти всегда интересует управление потоками, участвующими в выполнении задач, а не

изоляция как таковая. Изоляция нужна не для обеспечения корректности, а ради производительности. Явная арена задач – обоюдоострый меч, она позволяет контролировать потоки, принимающие участие в выполнении работы, но возводит между ними высоченную стену. Когда поток покидает изолированный регион, созданный функцией `isolate`, он вправе участвовать в выполнении любой задачи на своей арене. Когда же остается не у дел поток, работающий на явной арене, он обязан вернуться в глобальный пул, а уже потом найти другую арену, на которой есть работа и свободные слоты.

```
void fig_12_10() {
  const int N = 1000;

  tbb::task_arena ta2{2};
  tbb::task_arena ta6{6};
  double ta2_time = 0.0, ta6_time = 0.0;

  tbb::parallel_invoke(
    [&]() {
      ta2.execute([&]() {
        tbb::tick_count t0 = tbb::tick_count::now();
        tbb::parallel_for(0, N, [](int i) {doWork();});
        ta2_time = (tbb::tick_count::now() - t0).seconds();
      });
    },
    [&]() {
      ta6.execute([&]() {
        tbb::tick_count t0 = tbb::tick_count::now();
        tbb::parallel_for(0, N, [](int i) {doWork();});
        ta6_time = (tbb::tick_count::now() - t0).seconds();
      });
    }
  );
  std::cout << "ta2_time == " << ta2_time << std::endl
            << "ta6_time == " << ta6_time << std::endl;
}
```

Рис. 12.10 ❖ Два объекта `task_arena` созданы для того, чтобы один цикл выполнять в два потока, а другой – в шесть

Примечание. Мы только что сформулировали КЛЮЧЕВОЕ эвристическое правило: использовать `isolate` в основном для обеспечения корректности, а арены задач – в интересах производительности.

Рассмотрим еще раз пример на рис. 12.10. Мы создали больше слотов на арене `ta6`. В результате цикл `parallel_for` на `ta6` завершился гораздо раньше, чем `parallel_for` на `ta2`. Но после того как вся работа на `ta6` завершена, потоки, назначенные этой арене, возвращаются в глобальный пул. Они простаивают, но

не могут помочь в работе на ta2 – на этой арене всего два слота для потоков, и оба заняты!

Абстракция класса task_arena очень мощная, но стена, которую она создает между потоками, ограничивает ее практическую применимость. В главе 11 более подробно обсуждается, как можно использовать класс task_arena вместе с классами task_scheduler_init и global_control, чтобы управлять количеством потоков, доступных конкретным параллельным алгоритмам в приложении TBB. В главе 20 показано, как использовать объекты task_arena для разбиения работы на части и планирования отдельных частей на конкретных ядрах в архитектуре неравномерного доступа к памяти (Non-Uniform Memory Access – NUMA) с целью улучшения локальности данных. В обоих случаях мы видим, что арена задач весьма полезна, но имеет свои недостатки.

Не поддавайтесь искушению использовать арены задач для изоляции ради корректности

В ситуациях, описанных в главах 11 и 20, количество потоков и даже их распределение по конкретным ядрам жестко контролируется – поэтому мы хотим иметь разные потоки на разных аренах. Но в общем случае использование объектов task_arena для управления и перемещения потоков только порождает накладные расходы.

В качестве примера снова рассмотрим вложенные параллельные циклы, но теперь не в контексте корректности. На рис. 12.11 показан код и возможное дерево задач. Если выполнить эту пару циклов, то все задачи будут запущены на одной и той же арене. Когда в предыдущем разделе мы использовали isolate, все задачи по-прежнему находились на одной арене, но потоки изолировали их – они анализировали задачу, перед тем как позаимствовать ее, и выбирали только те, которые удовлетворяют ограничениям изолированности.

Теперь модифицируем этот простой пример и организуем изоляцию с помощью явных арен задач. Если мы хотим, чтобы каждый поток, исполняющий итерацию во внешнем цикле, выполнял только задачи из своего же внутреннего цикла, чего мы с легкостью достигли, применив изоляцию на рис. 12.4, то можем создать локальные вложенные явные экземпляры task_arena внутри каждого внешнего тела, как показано на рис. 12.12(a) и 12.12(b).

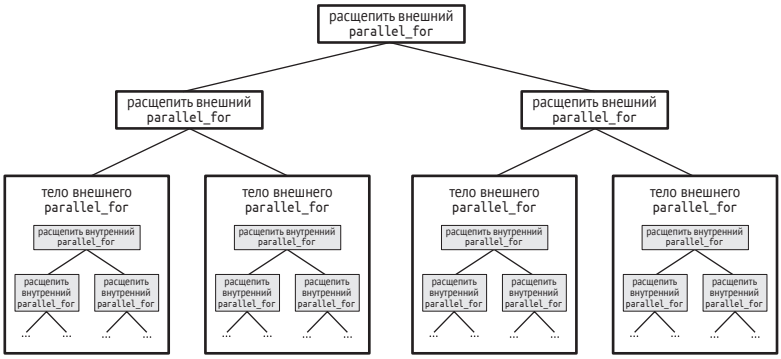
Если M == 4, то всего будет пять арен, и когда каждый поток вызывает nested.execute, он будет изолирован от задач из внешнего цикла, а также от задач из посторонних внутренних циклов. Мы придумали очень элегантное решение, правда?

Конечно, нет! Мало того что мы создаем, инициализируем и уничтожаем несколько объектов task_arena, так еще эти арены необходимо населить рабочими потоками. Как описано в главе 11, рабочие потоки распределяются по аренам

пропорционально количеству слотов. Если в системе четыре аппаратных потока, то каждая арена получит только один поток! И в чем смысл? Если потоков больше, то они равномерно распределятся между аренами задач. По завершении каждого внутреннего цикла его потоки вернутся в глобальный пул, а затем переместятся на другую арену, где еще есть задачи. Это не дешевая операция!

```
#include <tbb/tbb.h>

void doWork();

const int N = 128;

void fig_12_11(int M) {
  tbb::parallel_for(0, M,
    [](int) {
      tbb::parallel_for(0, N, [](int j) {doWork();});
    }
  );
}
```

(a) Исходный код двух вложенных циклов `parallel_for`



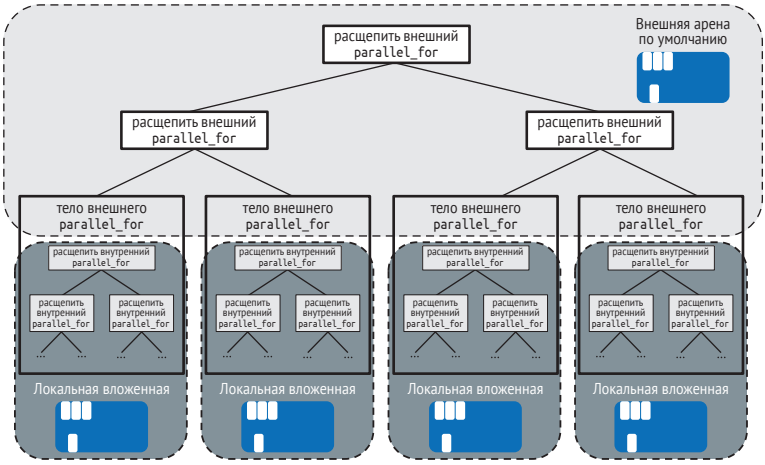
(b) Дерево задач для двух вложенных циклов `parallel_for`

Рис. 12.11 ❖ Пример двух вложенных циклов `parallel_for`: (a) исходный код; (b) дерево задач

Заводить много арен задач и перемещать между ними потоки – отнюдь не самый эффективный способ балансировки нагрузки. В нашем модельном примере на рис. 12.12(b) показано всего четыре внешние операции; а если бы их было больше, то нам пришлось бы создавать и уничтожать объекты `task_arena` в каждой внешней задаче. Наши четыре рабочих потока постоянно ковыляли бы от арены к арене в поисках работы! Для таких случаев приберегите функцию `isolate`!

```
#include <tbb/tbb.h>

void doWork();

const int N = 128;

void fig_12_12(int M) {
  tbb::parallel_for(0, M,
    [](int) {
      tbb::task_arena nested;
      nested.execute([]() {
        tbb::parallel_for(0, N, [](int j) {doWork();});
      });
    }
  );
}
```

(a) Исходный код



(b) Дерево задач с указанием арен, которым задачи принадлежат

Рис. 12.12 ❖ Создание явного объекта `task_arena` для каждого выполнения тела внешнего цикла. Теперь при выполнении на внутренней арене потоки будут изолированы от внешних работ и посторонних внутренних циклов

Резюме

Мы научились разделять задачи и алгоритмы TBB в случаях, когда они не могут ужиться. Мы видели, что вложенный параллелизм в сочетании с тем, как реализовано заимствование в TBB, может приводить к опасным ситуациям, если пренебречь осторожностью. Затем мы видели, что в таких ситуациях можно использовать функцию `this_task_arena::isolate`, но и это нужно делать осмотрительно, чтобы не создать себе новые проблемы.

Далее мы обсудили, как использовать класс `task_arena`, чтобы организовать изоляцию ради производительности. Хотя этот класс можно использовать и для изоляции в интересах корректности, накладные расходы в этом случае будут слишком велики. Но, как показано в главах 11 и 20, класс `task_arena` – важная часть нашего инструментария, необходимая, когда требуется управлять количеством потоков, доступных алгоритму, или распределением потоков по ядрам.

Дополнительная литература

Alexei Katranov. The Work Isolation Functionality in Intel Threading Building Blocks (Intel TBB) // https://software.intel.com/en-us/blogs/2018/08/16/the-work-isolation-functionality-in-intel-threading-building-blocks-intel-tbb.

Глава 13

Привязка потока к ядру и задачи к потоку

Разрабатывая параллельные приложения с помощью библиотеки TBB, мы создаем задачи, применяя высокоуровневые или низкоуровневые API. Эти задачи обрабатываются планировщиком TBB с заимствованием работ и отображаются на системные потоки. Эти потоки планируются операционной системой (ОС) и отображаются на аппаратные ядра (аппаратные потоки). В данной главе мы обсудим средства TBB, с помощью которых можно оказывать влияние на решения планировщиков ОС и TBB. *Привязка потока к ядру* используется, когда мы хотим, чтобы ОС разместила системный поток на конкретном ядре, а *привязка задачи к потоку* – когда требуется, чтобы планировщик TBB передал задачу конкретному потоку. В зависимости от цели нас может интересовать какой-то один вид привязки или оба сразу.

Для создания привязки могут быть разные причины. Одна из самых распространенных – более эффективно использовать локальность данных. Мы уже не раз говорили, что от локальности данных может очень сильно зависеть производительность параллельного приложения. Библиотека TBB, ее высокоуровневые интерфейсы, ее планировщик с заимствованием работ и конкурентные контейнеры проектировались с учетом локальности. Во многих приложениях использование этих средств дает хорошую производительность даже без ручной настройки. Но иногда стоит дать библиотеке указания или вообще взять все в свои руки, чтобы планировщики TBB и ОС планировали работу более оптимально – вблизи от обрабатываемых данных. Привязка может также представлять интерес в неоднородных системах, когда различаются возможности ядер или когда потоки обладают разными свойствами – скажем, у одних приоритет больше, чем у других.

В главе 16 представлены высокоуровневые средства обеспечения локальности данных, раскрываемые параллельными алгоритмами TBB. В главе 17 обсуждаются средства для настройки использования кеша и памяти в потоковых графах TBB. В главе 20 мы покажем, как эти возможности TBB применяются для настройки архитектур с неравномерным доступом к памяти (NUMA). Для многих читателей информации в этих главах будет достаточно, чтобы решить свои задачи по настройке приложений. В этой главе мы сосредоточимся на низкоуровневой фундаментальной поддержке со стороны планировщика и задач TBB, которая иногда абстрагируется высокоуровневыми средствами, а иногда используется непосредственно для организации привязки.

Создание привязки потока к ядру

Все основные операционные системы предлагают интерфейсы, позволяющие пользователю задать привязку к системному потоку. В Linux это функции `pthread_setaffinity_np` или `sched_setaffinity`, а в Windows – `SetThreadAffinityMask`. В главе 20 мы воспользуемся пакетом Portable Hardware Locality (hwloc) – переносимым способом задания привязки на разных платформах. В этой же главе мы опустим детали задания привязки, поскольку они зависят от системы, а акцентируем внимание на точках подключения внутри библиотеки TBB, которые позволяют применить эти интерфейсы для задания привязки рабочих и мастер-потоков.

Библиотека TBB по умолчанию создает столько рабочих потоков, сколько имеется ядер. В главе 11 мы обсуждали, как можно изменить эти параметры. Но в любом случае TBB автоматически не привязывает потоки к конкретным ядрам. TBB разрешает ОС планировать и перемещать потоки, как та считает нужным. Предоставление ОС гибкости в вопросе размещения потоков TBB – сознательное проектное решение. В многопрограммной среде, т. е. там, где TBB блистает, ОС видит все приложения и потоки. Если бы мы сами принимали решения о том, где должны исполняться потоки, руководствуясь лишь своей перспективой, ограниченной единственным приложением, то это могло бы повлечь за собой неэффективное использование системных ресурсов. Поэтому зачастую лучше не привязывать потоки к ядрам, а поручить ОС самой выбирать, где будут выполняться рабочие и мастер-потоки TBB, и, в частности, разрешить ей динамически перемещать потоки во время выполнения программы.

Однако, как показано во многих главах этой книги, TBB содержит средства, позволяющие при желании изменить это поведение. Если мы хотим привязать потоки TBB к ядрам, то можем воспользоваться классом `task_scheduler_observer` (см. врезку «Наблюдение за планировщиком с помощью класса task_scheduler_observer»). Этот класс позволяет определить обратные вызовы, которые библиотека будет вызывать всякий раз, как поток входит или покидает планировщик TBB или конкретную арену задач. С помощью этих обратных вызовов можно задать привязку. TBB не предлагает абстракцию для выполнения соответствующих системных вызовов, поэтому позаботиться об этих низкоуровневых деталях нам придется самостоятельно, т. е. воспользоваться вышеупомянутыми системно зависимыми функциями или переносимым интерфейсом.

Наблюдение за планировщиком с помощью класса task_scheduler_observer

Класс `task_scheduler_observer` позволяет наблюдать за тем, как поток начинает или заканчивает участвовать в планировании задач. Интерфейс этого класса приведен ниже.

```
class task_scheduler_observer {
public:
  task_scheduler_observer();
  virtual ~task_scheduler_observer():
  void observe( bool state=true );
```

```
    bool is_observing() const;
    virtual void on_sheduler_entry( bool is_worker ) {}
    virtual void on_sheduler_exit( bool is_worker ) {}
  };
```

Чтобы воспользоваться им, мы должны создать собственный класс, который наследует `task_scheduler_observer` и реализует обратные вызовы `on_scheduler_entry` и `on_scheduler_exit`. Если методу `observe` объекта этого класса передан параметр `state=true`, то указанные функции будут вызываться всякий раз, как мастер или рабочий поток входит в глобальный планировщик задач TBB или выходит из него.

Недавнее расширение этого класса позволяет также передавать конструктору арену задач. До выхода версии TBB 2019 Update 4 это расширение считалось ознакомительным средством, но теперь поддерживается в полном объеме. Если передана ссылка на объект `task_arena`, то наблюдатель будет получать обратные вызовы только для потоков, заходящих на указанную арену или покидающих ее:

```
explicit task_scheduler_observer(task_arena & a);
```

На рис. 13.1 приведен простой пример того, как объект `task_scheduler_observer` используется для привязки потоков к конкретным ядрам в Linux. Мы воспользовались функцией `sched_setaffinity`, чтобы задать маску процессора для каждого потока в момент его присоединения к арене по умолчанию. В главе 20 будет приведен пример задания привязки с помощью пакета hwloc. В примере на рис. 13.1 мы используем функцию `tbb::this_task_arena::max_concurrency()`, чтобы узнать число слотов на арене, и функцию `tbb::this_task_arena::current_thread_index()`, чтобы узнать, какой слот назначен вызывающему потоку. Поскольку мы знаем, что на арене по умолчанию столько слотов, сколько имеется логических ядер, то привязываем каждый поток к логическому ядру с таким же номером, как у его слота.

Конечно, мы можем создать и более хитроумные схемы назначения логических ядер потокам. Хотя в коде на рис. 13.1 этого нет, но можно было бы хранить первоначальную маску процессора для каждого потока и восстанавливать ее, после того как он покинет арену.

В главе 20 мы увидим, что класс `task_scheduler_observer` можно использовать совместно с явными экземплярами `task_arena` для создания изолированных групп потоков, которые могут исполняться только ядрами, разделяющими одни и те же банки локальной памяти в NUMA-системе, – так называемом узле NUMA. Если мы к тому же контролируем размещение данных, то можем значительно повысить производительность, запуская задачу на арене того узла NUMA, на котором находятся ее данные. Детали см. в главе 20.

Не следует забывать, что, используя привязку потока к ядру, мы не даем ОС перемещать потоки с перегруженных ядер на менее загруженные в попытке оптимизировать использование системных ресурсов. Делая такие вещи в производственном приложении, мы должны быть уверены, что общая производительность многопрограммной системы не упала! Мы еще не раз отметим, что лишь в системах, предназначенных для выполнения единственного приложения (в каждый момент времени), ограничение динамической миграции потоков может принести пользу.

```
#include <iostream>
#include <sched.h>
#define TBB_PREVIEW_LOCAL_OBSERVER 1
#include <tbb/tbb.h>

thread_local int my_cpu = -1;
void doWork();

class pinning_observer : public tbb::task_scheduler_observer {
public:
    pinning_observer() { observe(true); }

    void on_scheduler_entry( bool is_worker ) {
        cpu_set_t *mask;
        auto number_of_slots =
          tbb::this_task_arena::max_concurrency();
        mask = CPU_ALLOC(number_of_slots);
        auto mask_size = CPU_ALLOC_SIZE(number_of_slots);

        auto slot_number =
          tbb::this_task_arena::current_thread_index();
        CPU_ZERO_S(mask_size, mask);
        CPU_SET_S(slot_number, mask_size, mask);
        if (sched_setaffinity( 0, mask_size, mask )) {
           std::cout << "Error in sched_setaffinity"
                     << std::endl;
        }
        // чтобы узнать, все ли получилось
        my_cpu = sched_getcpu();
    }
};

void fig_13_1() {
  const int N = 100;

  std::cout << "Without pinning:" << std::endl;
  tbb::parallel_for(0, N, [](int) {doWork();});

  std::cout << std::endl
            << "With pinning:" << std::endl;
  pinning_observer p;
  tbb::parallel_for(0, N, [](int) {doWork();});
  std::cout << std::endl;
}
```

Рис. 13.1 ❖ Использование объекта task_scheduler_observer для привязки потоков к ядрам на платформе Linux

Создание привязки задачи к потоку

Поскольку в TBB мы выражаем параллелизм с помощью задач, привязка потока к ядру, описанная в предыдущем разделе, – лишь одна часть головоломки. Мы можем и не получить заметной выгоды от привязки потоков к ядрам, если

позволим задачам беспрепятственно менять потоки вследствие заимствования работ!

Пользуясь низкоуровневыми интерфейсами задач, описанными в главе 10, мы можем указать планировщику TBB, что некоторую задачу желательно исполнять в потоке, занимающем определенный слот на арене. Но поскольку мы стремимся всюду, где возможно, использовать высокоуровневые алгоритмы и интерфейсы задач, например `parallel_for`, `task_group` и потоковые графы, нам редко придется обращаться к этим низкоуровневым интерфейсам напрямую. В главе 16 показано, как классы `affinity_partitioner` и `static_partitioner` можно использовать совместно с циклическими алгоритмами TBB, чтобы создать привязку, не прибегая к низкоуровневым интерфейсам. А в главе 17 обсуждаются возможности потоковых графов, относящиеся к привязке.

Итак, несмотря на то что привязка раскрывается низкоуровневым классом, мы почти всегда пользуемся этим средством опосредованно, через высокоуровневые абстракции. Поэтому использование интерфейсов, описанных в этом разделе, лучше оставить экспертам по TBB, которые пишут алгоритмы, применяя средства самого низкого уровня. Впрочем, если вы из их числа или хотите глубже разобраться в том, как привязка реализуется на более высоком уровне, читайте дальше.

На рис. 13.2 показаны функции и типы из класса TBB `task`, которые относятся к привязке задач.

```
typedef implementation-defined-unsigned-type affinity_id;
virtual void note_affinity( affinity_id id );
void set_affinity( affinity_id id );
affinity_id affinity() const;
```

Рис. 13.2 ❖ Функции в классе `tbb::task`, предназначенные для привязки задачи к потоку

Тип `affinity_id` используется для представления того слота на арене, к которому привязана задача. Значение 0 означает, что у задачи нет привязки. Мы можем установить привязку задачи к слоту до ее запуска, передав `affinity_id` функции-члену `set_affinity`. Но поскольку значение `affinity_id` зависит от реализации, мы не передаем конкретное значение, скажем 2 для второго слота. Вместо этого мы запоминаем `affinity_id` при предыдущем выполнении задачи, переопределив функцию обратного вызова `note_affinity`.

Эта функция вызывается библиотекой TBB перед вызовом функции задачи `execute`, если (1) задача не имеет привязки, но будет выполнена в потоке, отличающемся от того, который ее запустил, или (2) у задачи есть привязка, но она будет выполнена в потоке, отличном от того, который задан в привязке. Переопределив этот обратный вызов, мы можем следить за заимствованием и давать библиотеке указания о том, чтобы она воспроизвела точно такое же поведение заимствования при последующем выполнении алгоритма (это будет продемонстрировано в следующем примере).

Наконец, функция `affinity` позволяет узнать текущую привязку задачи.

На рис. 13.3 показан класс, который наследует `tbb::task` и использует функции привязки задачи для запоминания значений `affinity_id` в глобальном мас-

```
#include <iostream>
#define TBB_PREVIEW_LOCAL_OBSERVER 1
#include <tbb/tbb.h>

const int numTasks = 8;
tbb::task::affinity_id a[numTasks];
void doWork();
void printExclaim(const std::string &str, int id,
                  int slot, int note);

class MyTask : public tbb::task {
  int taskId;
  bool doMakeNotes;
public:
  MyTask(int id, bool do_make_notes = true)
    : taskId(id), doMakeNotes(do_make_notes) { }

  void note_affinity(tbb::task::affinity_id id) override {
    if (doMakeNotes) a[taskId] = id;
  }

  tbb::task *execute() override {
    auto slot_number = tbb::this_task_arena::current_thread_index();
    tbb::task::affinity_id a_id = a[taskId];
    std::string exclaim = "yay!";
    if (a_id != 0 && slot_number != a_id-1) exclaim = "boo!";
    if (doMakeNotes) exclaim = "hmm.";
    printExclaim(exclaim, taskId, slot_number, a_id);
    doWork();
    return NULL;
  }
};

void executeTaskTree(const std::string &name, bool note_affinity,
                     bool set_affinity) {
  std::cout << name << std::endl << "id:slot:a[i]" << std::endl;
  tbb::task *root = new(tbb::task::allocate_root()) tbb::empty_task;
  root->set_ref_count(numTasks+1);
  for (int i = 0; i < numTasks; ++i) {
    tbb::task *t = new (root->allocate_child())
                     MyTask(i,note_affinity);
    if (set_affinity && a[i]) t->set_affinity(a[i]);
    tbb::task::spawn(*t);
  }
  root->wait_for_all();
}

void fig_13_3() {
  std::fill(a, a+numTasks, 0);
  executeTaskTree("note_affinity", true, false);
  executeTaskTree("without set_affinity", false, false);
  executeTaskTree("with set_affinity", false, true);
}
```

Рис. 13.3 ❖ Использование функций для привязки задачи

сиве a. Значение запоминается, только когда переменная doMakeNotes равна true. Функция execute печатает идентификатор задачи, слот потока, исполняющего задачу, и значение, которое было записано в позицию массива, соответствующую идентификатору этой задачи. Сообщению предшествует строка «hmm», если doMakeNotes равно true (после этого значение запоминается), «yay!» – если задача исполняется в слоте арены, идентификатор которого был запомнен в массиве a (снова запланирована тому же потоку), и «boo!» – если она исполняется в другом слоте. Логика печати инкапсулирована в функции printExclaim.

Хотя значение affinity_id считается деталью реализации, исходный код TBB открыт, и мы заглянули в него. Поэтому мы знаем, что affinity_id равно 0, когда привязки нет, а в противном случае равно индексу слота плюс 1. Опираться на эти знания при написании производственного кода не следует, но в коде функции execute выше мы ими воспользовались, когда выбирали между печатью «yay!» или «boo!».

Функция fig_13_3 на рис. 13.3 строит и выполняет три дерева задач, по восемь задач в каждом, и назначает им идентификаторы от 0 до 7. В этом примере используются низкоуровневые интерфейсы задач, описанные в главе 10. В первом дереве задач функция note_affinity применяется, чтобы следить за тем, когда задача была заимствована потоком, отличным от мастера. Второе дерево задач исполняется без слежения и задания привязок. И наконец, в последнем дереве задач используется функция set_affinity для воссоздания того же порядка планирования, что при первом прогоне.

Когда эта программа была запущена на 8-ядерной платформе, она напечатала:

```
note_affinity
id:slot:a[i]
hmm. 7:0:-1
hmm. 0:1:1
hmm. 1:6:6
hmm. 2:3:3
hmm. 3:2:2
hmm. 4:4:4
hmm. 5:7:7
hmm. 6:5:5

without set_affinity
id:slot:a[i]
yay! 7:0:-1
boo! 0:4:1
boo! 1:3:6
boo! 4:5:4
boo! 3:7:2
boo! 2:2:3
boo! 5:6:7
boo! 6:1:5

with set_affinity
id:slot:a[i]
yay! 7:0:-1
yay! 0:1:1
```

```
yay! 4:4:4
yay! 5:7:7
yay! 2:3:3
yay! 3:2:2
yay! 6:5:5
yay! 1:6:6
```

Отсюда видно, что задачи в первом дереве распределены по всем восьми имеющимся потокам, а `affinity_id` для каждой задачи записано в массив `a`. При выполнении следующего набора задач записанные значения `affinity_id` не используются для задания привязки, и задачи случайным образом заимствуются другими потоками. Так и должно быть! Но когда в последнем прогоне мы пользуемся функцией `set_affinity`, повторяются те же назначения потоков задачам, что при первом прогоне. Замечательно, именно этого мы и добивались!

Однако `set_affinity` лишь дает *указание*, которое библиотека TBB вправе игнорировать. Когда мы задаем привязку с помощью этих интерфейсов, ссылка на задачу с привязкой помещается в буфер привязки целевого потока (см. рис. 13.4). Но сама задача остается в локальной двусторонней очереди запустившего ее потока. Диспетчер задач проверяет буфер привязки, только когда у него кончились работы в локальной очереди, как было показано в цикле диспетчеризации задач в главе 9. Поэтому, если поток не проверит свой буфер привязки достаточно быстро, другой поток может первым позаимствовать и выполнить его задачи.

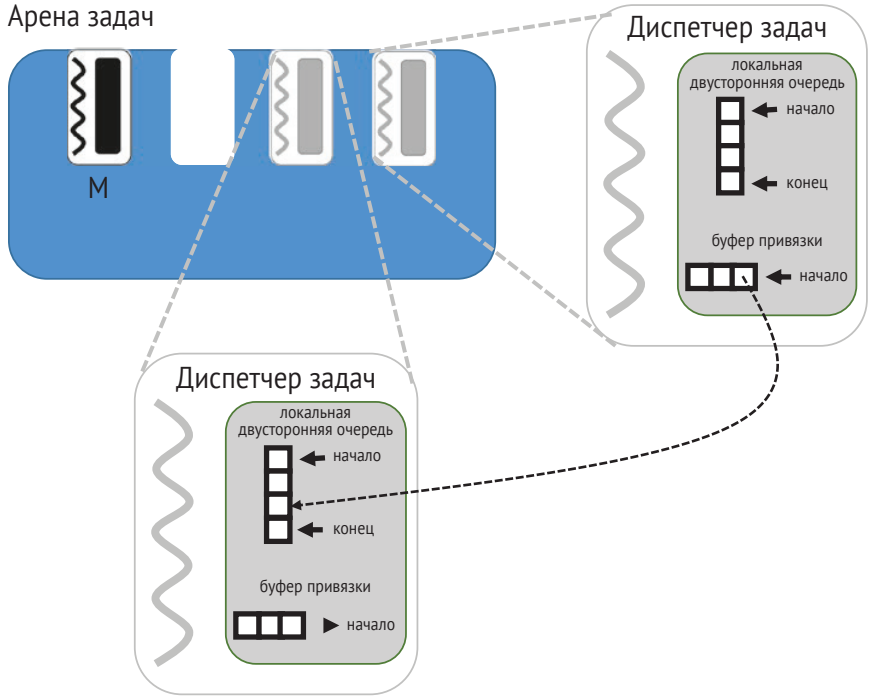


Рис. 13.4 ❖ В буфере привязки хранится ссылка на задачу, а сама задача остается в локальной двусторонней очереди запустившего ее потока

Для демонстрации всего сказанного выше мы можем изменить порядок привязки в нашем примере, как показано на рис. 13.5. На этот раз – как это ни глупо – мы установим привязки всех задач к одному и тому же слоту, номер которого хранится в элементе a[2].

```
void executeTaskTree(const std::string &name, bool
note_affinity,
                     bool set_affinity) {
  std::cout << name << std::endl << "id:slot" << std::endl;
  tbb::task *root = new(tbb::task::allocate_root())
tbb::empty_task;
  root->set_ref_count(numTasks+1);
  for (int i = 0; i < numTasks; ++i) {
    tbb::task *t = new ( root->allocate_child() )
                     MyTask(i,note_affinity);
    if (set_affinity) t->set_affinity(a[2]);
    tbb::task::spawn(*t);
  }
  root->wait_for_all();
}

void fig_13_5() {
  std::fill(a, a+numTasks, 0);
  executeTaskTree("note_affinity", true, false);
  executeTaskTree("with set_affinity to a[2]", false, true);
}
```

Рис. 13.5 ❖ Функция, которая выполняет разные группы задач, иногда запоминая привязки, а иногда устанавливая их

Если бы планировщик TBB удовлетворял все наши запросы на привязку, то нагрузка оказалась бы сильно разбалансирована, потому что мы просим привязать все задачи к одному и тому же рабочему потоку. Но, выполнив новый вариант примера, мы увидим такую картину:

```
id:slot
7:0
0:1
1:2
3:4
2:3
5:5
4:6
6:7
with set_affinity to a[2]
id:slot
7:0
0:3
2:4
1:6
```

```
5:1
6:5
3:7
4:2
```

Поскольку привязка – это всего лишь указание, остальные простаивающие потоки по-прежнему находят задачи и заимствуют их из локальной двусторонней очереди мастер-потока, до того как поток в слоте `a[2]` успевает опустошить свой буфер привязки. На самом деле только первая запущенная задача с `id==0` исполняется потоком, находящимся в том слоте, который запомнен в `a[2]`. Таким образом, наши задачи, как и раньше, распределены между всеми восьмью потоками.

Библиотека TBB проигнорировала наш запрос и тем самым сумела избежать несбалансированной нагрузки в случае, если бы все эти задачи были направлены одному потоку. Такая слабая привязка полезна на практике, потому что позволяет указать, какая привязка, по нашему мнению, могла бы улучшить производительность, но оставляет библиотеке возможность внести корректировки, чтобы случайно не возник серьезный дисбаланс.

Хотя эти интерфейсы можно использовать непосредственно, в главе 16 мы увидим, что циклические алгоритмы предлагают упрощенную абстракцию, класс `affinity_partitioner`, которая успешно скрывает от нас большую часть низкоуровневых деталей.

Когда и как следует использовать средства привязки в TBB?

Объекты `task_scheduler_observer` следует использовать для привязки потока к ядру, только если мы стремимся максимизировать производительность в специально выделенной системе. В противном случае лучше дать ОС возможность заниматься своим делом и планировать потоки, как она считает разумным с учетом всей доступной ей информации. Решив привязывать потоки к ядрам, мы должны тщательно оценить, к чему приведет отъем этих функций у ОС, особенно если наше приложение выполняется в многопрограммной среде.

Что касается привязки задач к потокам, то обычно лучше использовать высокоуровневые интерфейсы, например класс `affinity_partitioner`, описанный в главе 16. Этот класс пользуется средствами, упомянутыми в данной главе, чтобы следить, где исполнялись задачи, и давать планировщику TBB указания воспроизвести то же разбиение при следующих выполнениях цикла. Он также отслеживает изменения, чтобы поддерживать актуальность указаний.

Поскольку привязки задач в TBB являются лишь указаниями планировщику, потенциальный вред от неправильного использования этих интерфейсов снижается, так что очень уж осторожничать при задании привязок необязательно. На самом деле эксперименты с привязкой задач, особенно при посредстве высокоуровневых интерфейсов, даже приветствуются и являются обычным элементом настройки приложений.

Резюме

В этой главе мы обсудили, как создать привязки потоков к ядрам и задач к потокам в приложении TBB. Хотя TBB не предоставляет интерфейса, содержащего конкретные детали задания привязки потока к ядру, класс `task_scheduler_observer` предлагает механизм обратных вызовов, позволяющий включить необходимые обращения к системно зависимым функциям или переносимым библиотекам. Поскольку планировщик TBB с заимствованием работ случайным образом назначает задачи системным потокам, одной лишь привязки потоков к ядрам может оказаться недостаточно. Поэтому мы также обсудили интерфейсы класса TBB `task`, которые позволяют передавать планировщику указания о том, к каким потокам было бы желательно привязать задачи. Мы отметили, что чаще всего эти интерфейсы используются не напрямую, а при посредстве интерфейсов более высокого уровня, описанных в главах 16 и 17. Однако для читателей, которым интересно узнать о низкоуровневых интерфейсах, мы включили примеры, демонстрирующие использование функций `note_affinity` и `set_affinity` для реализации привязки задач к потокам.

Как и ко многим другим средствам оптимизации, имеющимся в библиотеке TBB, к привязкам следует подходить с осторожностью. Неправильное использование привязки потоков к ядрам может значительно снизить производительность, лишив ОС возможности балансировать нагрузку. Злоупотребление привязкой задач к потокам тоже может принести вред, но, поскольку это всего лишь указания, которые планировщик TBB вправе игнорировать, риски гораздо меньше.

Дополнительная информация

- Определенные в Posix функции получения и задания привязки потока к ядрам ЦП описаны на странице руководства http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html.
- Функция `SetThreadAffinityMask` описана по адресу https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-setthreadaffinitymask.
- Portable Hardware Locality (hwloc). URL: www.open-mpi.org/projects/hwloc/.

Глава 14

Приоритеты задач

Планировщик TBB не является планировщиком реального времени, поэтому непригоден для использования в системах *жесткого* реального времени. В системе реального времени для задачи может быть задан предельный срок, к которому она должна быть завершена, и если задача не укладывается в этот срок, то ее полезность снижается. В системах жесткого реального времени пропуск срока может стать причиной полного отказа системы. В системах мягкого реального времени пропуск срока – не катастрофа, но влечет за собой ухудшение качества обслуживания. В библиотеке TBB назначение предельных сроков выполнения задачам не поддерживается, зато поддерживаются приоритеты задач. Приоритеты могут быть полезны в приложениях, к которым предъявляются мягкие требования в части реального времени. Чтобы понять, достаточно ли этого, нужно знать как требования к приложению, так и свойства задач и их приоритетов в TBB.

Помимо использования в системах мягкого реального времени, у приоритетов задач есть и другие применения. Например, иногда мы хотим повысить приоритет некоторых задач в ущерб другим, чтобы улучшить производительность или отзывчивость приложения. Или, быть может, чтобы отдать предпочтение задачам, освобождающим память, перед задачами, которые ее выделяют, и тем самым снизить потребление памяти приложением. Или повысить приоритет задач, работающих с уже кешированными данными, по сравнению с задачами, загружающими новые данные в кеш.

В этой главе мы опишем, как приоритеты задач поддерживаются планировщиком TBB. Читатели, рассматривающие применение TBB в приложениях мягкого реального времени, могут воспользоваться этой информацией, чтобы решить, удовлетворяет ли TBB их требованиям. Остальным читателям она может пригодиться для оптимизации производительности, возможной с помощью приоритетов задач.

Поддержка невытесняющих приоритетов в классе задач TBB

Как и поддержка привязки задач, описанная в главе 13, поддержка приоритетов обеспечивается функциями в низкоуровневом классе `task`. В TBB определено три приоритета: `priority_normal`, `priority_low` и `priority_high` (рис. 14.1).

```
namespace tbb {
  enum priority_t {
    priority_normal = implementation-defined,
    priority_low = implementation-defined,
    priority_high = implementation-defined
  };

  class task {
    // . . .
    static void enqueue( task&, priority_t );
    void set_group_priority ( priority_t );
    priority_t group_priority () const;
    // . . .
  };
}
```

Рис. 14.1 ❖ Типы и функции для поддержки приоритетов в классе `task`

Вообще говоря, задачи с более высоким приоритетом выполняются быстрее задач с низким приоритетом. Но есть нюансы.

Самый важный подводный камень заключается в том, что задачи TBB исполняются потоками TBB без вытеснения. Коль скоро выполнение задачи началось, она будет выполнена до конца – даже если тем временем запущена или поставлена в очередь задача с более высоким приоритетом. Такое поведение может показаться недостатком, потому что задерживается переключение на высокоприоритетные задачи, но в нем есть и свои плюсы, т. к. оно предотвращает некоторые опасные ситуации. Представим, к примеру, что задача t_0 захватила некоторый разделяемый ресурс, а затем запущена задача с более высоким приоритетом. Если TBB не позволит задаче t_0 завершиться и освободить ресурс, то в случае, когда более приоритетная задача попытается захватить тот же ресурс, возникнет взаимоблокировка. Более сложная проблема той же природы, *инверсия приоритетов*, известна тем, что вызвала проблемы в марсоходе Mars Pathfinder в конце 1990-х годов. В статье «What Happened on Mars?» (Что произошло на Марсе?) Майк Джонс предлагает разрешать такие проблемы с помощью наследования приоритетов. В этом случае задача, блокирующая более приоритетные задачи, наследует приоритет самой приоритетной из блокируемых задач. В библиотеке TBB не реализовано ни наследование приоритетов, ни другие сложные подходы, потому что в силу невытесняющей многозадачности многие из этих проблем просто не могут возникнуть.

Библиотека TBB ***не*** предоставляет высокоуровневой абстракции для задания ***приоритета потока***. Поэтому если хотим это сделать, то должны прибегнуть к системно зависимому коду – так же, как при задании привязки потока к ядру в главе 13. И точно так же мы можем пользоваться объектами `task_scheduler_observer` и вызывать системно зависимые функции в обратных вызовах, когда поток входит в планировщик задач TBB или на конкретную арену задач либо покидает их. Однако предупреждаем, что ***при использовании приоритетов потоков нужна предельная осторожность***. Входя в мир приоритетных потоков, для которых поддерживается вытеснение, мы распахиваем дверь для всех известных патологий, в т. ч. инверсии приоритетов.

Важнейшее правило. Не задавайте разные приоритеты для потоков, работающих на одной и той же арене. Из-за этого могут и обязательно будут происходить странные вещи, потому что TBB рассматривает потоки на одной арене как равноправные.

Помимо невытесняющей природы многозадачности в TBB, стоит отметить и другие важные ограничения, относящиеся к поддержке приоритетов задач. Во-первых, изменения не всегда вступают в силу сразу для всех потоков. Это возможно, потому что некоторые низкоприоритетные потоки могут начать выполнение даже при наличии более приоритетных. Во-вторых, рабочим потокам, возможно, придется переместиться на другую арену, чтобы получить доступ к самым высокоприоритетным задачам, а для этого, как мы отмечали в главе 12, может потребоваться время. Коль скоро рабочие потоки мигрировали, некоторые арены (не имеющие высокоприоритетных задач) могут остаться вообще без рабочих потоков. Но поскольку мастер-потоки мигрировать не могут, они остаются и при этом не приостановлены, а значит, могут продолжать выполнение задач на своих аренах, пусть даже их приоритет ниже.

Приоритеты задач не являются необязательными к исполнению указаниями, как в случае привязки задач к потокам (см. главу 13). И тем не менее существует достаточно подводных камней, из-за которых на практике приоритеты оказываются не столь мощным орудием, как нам хотелось бы. Кроме того, наличия всего трех уровней приоритета, низкого, обычного и высокого, может оказаться недостаточно в сложных приложениях. Но, несмотря на все это, мы продолжим рассказ о механизмах приоритетов в TBB.

Задание статических и динамических приоритетов

Статические приоритеты можно назначать отдельным задачам, которые ставятся в разделяемую очередь (о постановке задач в очередь см. главу 10). А *динамические приоритеты* назначаются группам задач с помощью функции `set_group_priority` или функции-члена `set_priority` объекта `task_group_context` (см. врезку ниже).

TASK_GROUP_CONTEXT: любая задача принадлежит какой-то группе

Класс `task_group_context` представляет группу задач, которые можно отменить разом и для которых можно задать общий приоритет. Любая задача принадлежит какой-то группе, но только одной в каждый момент времени.

В главе 10 мы выделяли память для задач TBB с помощью таких специальных функций, как `allocate_root()`. Существует перегруженный вариант этой функции, который позволяет назначить вновь созданной корневой задаче объект `task_group_context`:

```
static proxy2 allocate_root( task_group_context& );
```

Объект типа `task_group_context` можно также передать в качестве необязательного аргумента высокоуровневым TBB-алгоритмам и потоковым графам, например:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index_type last, const Func& f,
```

```
                    partitioner[, task_group_context& group]] );
  graph([task_group_context& context]);
```

Мы можем назначать группы на уровне задач во время создания или с помощью высокоуровневых интерфейсов – TBB-алгоритмов и потоковых графов. Существуют и другие абстракции, например `task_group`, позволяющая группировать задачи в целях выполнения. Класс `task_group_context` предназначен для поддержки отмены, обработки исключений и приоритетов.

Если для задания приоритета используется функция `task::enqueue`, то приоритет распространяется только на одну задачу и не может быть изменен впоследствии. Если же приоритет назначается группе задач, то он распространяется на все задачи в этой группе и может быть в любой момент изменен с помощью функции `task::set_group_priority` или `task_group_context::set_priority`.

Планировщик TBB запоминает самый высокий приоритет готовых к выполнению задач, запущенных и поставленных в очередь, и задерживает (с учетом ранее описанных нюансов) выполнение низкоприоритетных задач до тех пор, пока не будут выполнены все более приоритетные. По умолчанию всем задачам и группам задач назначается приоритет `priority_normal` в момент создания.

Два простых примера

На рис. 14.2 приведен пример программы, которая ставит в очередь задачи на платформе с P логическими ядрами. Каждая задача активно ожидает в течение заданного времени. Первая задача в функции `fig_14_02` создается с обычным приоритетом и работает приблизительно 500 мс. Затем в цикле for создается по P задач с низким, обычным и высоким приоритетами, каждая из которых работает приблизительно 10 мс. Во время выполнения задача помещает сообщение в буфер в поточно-локальной памяти. Сообщения высокоприоритетных задач начинаются буквой H, обычных – буквой N, а низкоприоритетных – буквой L. В конце печатается содержимое всех буферов, так что мы видим, в каком порядке задачи выполнялись потоками. Полный код примера имеется на Github.

При выполнении этого примера в 8-ядерной системе результат получается таким:

```
N:0              ← thread 1
H:7 H:5 N:3 L:7  ← thread 2
H:2 H:1 N:8 L:5  ← thread 3
H:6 N:1 L:3 L:2  ← thread 4
H:0 N:2 L:6 L:4  ← thread 5
H:3 N:4 N:5 L:0  ← thread 6
H:4 N:7 N:6 L:1  ← thread 8
```

Здесь каждая строка представляет отдельный рабочий поток. Для каждого потока выполненные им задачи упорядочены слева направо. Мастер-поток не участвует в выполнении этих задач, поскольку он не вызывает `wait_for_all`, так что мы видим всего семь строк. Первый поток исполняет только первую

```
#include <iostream>
#include <string>

#include <tbb/tbb.h>

void doWork(double sec);

class Spinner : public tbb::task {
public:
  Spinner(const char *m, int id, double len);
  tbb::task *execute() override;

private:
  std::string msg;
  int messageId;
  double length;
};

void enqueueTask(const char *msg,
                 int id,
                 double len,
                 tbb::priority_t
priority=tbb::priority_normal) {
  tbb::task::enqueue(*new( tbb::task::allocate_root())
                       Spinner(msg, id, len),
                     priority);
}

void fig_14_02() {
  int P = tbb::task_scheduler_init::default_num_threads();

  enqueueTask("N", 0, 0.5);
  doWork(0.01);

  for (int i = 0; i < P; ++i) {
    enqueueTask("L", i, 0.01, tbb::priority_low);
    enqueueTask("N", i+1, 0.01, tbb::priority_normal);
    enqueueTask("H", i, 0.01, tbb::priority_high);
  }
  doWork(1.0);
}
```

Рис. 14.2 ❖ Постановка в очередь задач с различными приоритетами

долгую задачу с обычным приоритетом, которая работает 500 мс. Поскольку многозадачность в TBB невытесняющая, этот поток не может отказаться от своей задачи, раз уж он взялся за нее, поэтому продолжает ее выполнять, хотя имеются и более приоритетные задачи. Но в остальных строчках мы видим, что хотя в цикле for попеременно ставились в очередь задачи с разными прио-

ритетами, рабочие потоки исполняют сначала задачи с высоким приоритетом, потом с обычным и в самом конце с низким.

В коде на рис. 14.3 параллельно выполняются два алгоритма `parallel_for` в двух платформенных потоках t_0 и t_1. В каждом цикле `parallel_for` имеется 16 итераций и используется объект `simple_partitioner`. Как описано в главе 16, `simple_partitioner` разбивает пространство итераций, пока не будет достигнута заданная зернистость, по умолчанию 1. В нашем примере каждый `parallel_for` будет выполнять 16 задач, каждая из которых работает в течение 10 мс. Код, исполняемый потоком t_0, создает объект `task_group_context` и назначает ему приоритет `priority_high`. В потоке t_1 используется `task_group_context` по умолчанию с приоритетом `priority_normal`.

```
#include <iostream>
#include <thread>

#include <tbb/tbb.h>

void doWork(double sec);

void fig_14_03() {
  std::thread t0([]() {
    tbb::task_group_context tcg;
    tcg.set_priority(tbb::priority_high);
    tbb::parallel_for( 0, 16, [] (int) {
      // выполняется высокоприоритетная работа
      doWork(0.01);
      std::cout << "High\n";
    }, tbb::simple_partitioner(), tcg );
  });

  std::thread t1( []() {
    tbb::parallel_for( 0, 16, [] (int) {
      // выполняется работа с нормальным приоритетом
      doWork(0.01);
      std::cout << "Normal\n";
    }, tbb::simple_partitioner());
  });

  t0.join();
  t1.join();
}
```

Рис. 14.3 ❖ Выполнение алгоритмов с разными приоритетами

При запуске на 8-ядерной платформе получается такой результат:

```
Normal
High
High
High
```

```
High
High
High
Normal
High
High
High
High
High
High
High
High
Normal
High
High
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
Normal
```

Сначала мы видим семь задач «High», выполненных для каждой задачи «Normal». Это объясняется тем, что поток t_1, запустивший цикл `parallel_for` с обычным приоритетом, не может покинуть свою неявную арену задач. Он может выполнять только задачи «Normal». Однако остальные семь потоков выполняют только задачи «High», пока они не кончатся. А после того как высокоприоритетные задачи закончились, все рабочие потоки могут переместиться на арену потока t_1 и помочь ему.

Реализация приоритетов без поддержки со стороны задач TBB

Что, если трех уровней приоритета недостаточно? Одно из возможных обходных решений – запускать обобщенные задачи-обертки, которые используют очереди с приоритетами или другую структуру данных, чтобы найти для себя работу. При таком подходе планировщик TBB распределяет эти задачи-обертки между ядрами, а сами задачи реализуют стратегию приоритетного выполнения с помощью разделяемой структуры данных.

В примере на рис. 14.4 используются классы `task_group` и `concurrent_priority_queue`. Когда нужно выполнить некоторую работу, предпринимается два действия: (1) описание работы помещается в разделяемую очередь и (2) за-

дача-обертка запускается в составе группы task_group, которая извлекает и выполняет работу из очереди. В результате запускается ровно одна задача для каждой работы, но какую именно работу будет обрабатывать эта задача, неизвестно до начала выполнения.

```
#include <iostream>
#include <tbb/tbb.h>

class WorkItem {
public:
  WorkItem() { }
  WorkItem(int p) : priority(p) { }

  bool operator<(const WorkItem &b) const {
    if (priority < b.priority) return true;
    else return false;
  }

  void doWork();

private:
  int priority;
};

void fig_14_4() {
  tbb::concurrent_priority_queue<WorkItem> q;
  tbb::task_group g;

  const std::string prefix = "w";
  for (int i = 0; i < 16; ++i) {
    q.push(WorkItem(i));
    g.run([&q]() {
      WorkItem w;
      if (q.try_pop(w))
        w.doWork();
    });
  }
  g.wait();
}
```

Рис. 14.4 ❖ Использование конкурентной очереди с приоритетами для подачи работы задачам-оберткам

В конкурентной очереди с приоритетами порядок по умолчанию задается с помощью оператора operator<, поэтому если work_item::operator< определен, как на рис. 14.4, то работы будут выполняться в порядке убывания приоритета от 15 до 0, как показано ниже:

```
WorkItem: 15
WorkItem: 14
WorkItem: 13
WorkItem: 12
WorkItem: 11
WorkItem: 10
WorkItem: 9
WorkItem: 8
WorkItem: 7
WorkItem: 6
WorkItem: 5
WorkItem: 4
WorkItem: 3
WorkItem: 2
WorkItem: 1
WorkItem: 0
```

Если изменить оператор, так чтобы он возвращал `true`, когда (`priority > b.priority`), то задачи будут выполняться в порядке возрастания приоритета от 0 до 15.

Применение обобщенных задач-оберток повышает гибкость, потому что мы сами управляем определением приоритетов. Но, по крайней мере на рис. 14.4, это создает узкое место – потоки конкурентно обращаются к разделяемой структуре данных. Тем не менее если приоритетов задач, определенных TBB, недостаточно, мы можем прибегнуть и к такому резервному плану.

Резюме

В этой главе мы дали краткий обзор поддержки приоритетов задач в TBB. Механизмы, предоставляемые классом `task`, позволяют назначать задаче один из трех приоритетов: низкий, обычный или высокий. Мы показали, как можно назначить статические приоритеты задачам в момент постановки в очереди и динамические приоритеты группам задач с помощью объектов `task_group_context`. Поскольку многозадачность в TBB невытесняющая, приоритеты также невытесняющие. Мы вкратце описали достоинства и недостатки невытесняющих приоритетов, а также некоторые подводные камни, о которых нужно помнить, пользуясь поддержкой со стороны библиотеки. Затем мы привели несколько примеров, на которых продемонстрировали назначение приоритетов задачам и алгоритмам TBB.

Поскольку поддержка приоритетов в TBB обременена многими ограничениями, мы завершили обсуждение демонстрацией альтернативного подхода с применением задач-оберток и очереди с приоритетами.

Планировщик TBB не является планировщиком реального времени. Но тем не менее в этой главе мы видели, что ограниченная поддержка приоритетов задач и алгоритмов все же имеется. Достаточно ли ее для реализации приложения мягкого реального времени или для оптимизации производительности, предстоит решать разработчику в каждом конкретном случае.

Дополнительная информация

- *Mike Jones*. What Happened on Mars? // Заметка, отправленная 4 декабря 1997 года. URL: www.cs.cmu.edu/afs/cs/user/raj/www/mars.html.
- *L. Sha, R. Rajkumar, and J. P. Lehoczky*. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers. Vol. 39. P. 1175–1185. Sep. 1990.

Глава 15

Отмена и обработка исключений

Время от времени все мы сталкиваемся с ошибками во время выполнения, будь то последовательная или параллельная программа. Чтобы утишить боль, мы научились оповещать о них с помощью кодов ошибок или более высокоуровневых альтернатив, в частности исключений. Язык C++, как и большинство объектно-ориентированных языков, поддерживает обработку исключений, и при грамотном использовании это позволяет создавать надежные приложения. Принимая во внимание, что библиотека TBB добавляет параллелизм на основе задач поверх C++, следует ожидать, что обработка исключений в ней хорошо поддержана. И как мы увидим ниже, это действительно так. Это означает, что в случае ошибки (боже упаси!) наш код сможет прибегнуть к обработчику исключений, если таковой существует, или завершить всю программу. Поддержка исключений в TBB далеко не тривиальное дело, поскольку:

1) исключения могут быть возбуждены в задачах, исполняемых несколькими потоками;
2) чтобы завершить работу, в которой возникло исключение, необходимо реализовать отмену задач;
3) необходимо сохранить компонуемость TBB;
4) управление исключениями не должно сказываться на производительности в случае отсутствия исключений.

В TBB обработка исключений удовлетворяет всем этим требованиям, включая и отмену задач. Как было сказано, отмена необходима, потому что в случае возникновения исключения может возникнуть необходимость отменить выполнение параллельного алгоритма, в котором оно возникло. Например, если в алгоритме `parallel_for` произойдет выход за границы массива или деление на ноль, то библиотека должна будет отменить весь `parallel_for`. Для этого TBB предстоит отменить все задачи, занятые параллельной обработкой частей пространства итераций, а затем перейти к обработчику исключений. Реализация отмены задач в TBB устроена так, что отменяются только задачи, участвующие в сбойном алгоритме `parallel_for`, а все остальные никак не затрагиваются.

Отмена задач нужна не только для обработки исключений, у нее есть и другие применения. Поэтому мы начнем эту главу с демонстрации того, как можно использовать отмену для ускорения некоторых параллельных алгоритмов. Хотя отмена TBB-алгоритмов «просто работает», продвинутым разработчи-

кам может быть интересно узнать, как получить полный контроль над отменой и как она реализована в TBB. Мы попытаемся удовлетворить это желание (напомним, что эта часть книги посвящена как раз таким дополнительным темам). Во второй части главы мы перейдем к обработке исключений. И снова – обработка исключений «просто работает», без каких бы то ни было добавлений: для перехвата исключений, определенных как в стандарте C++, так и в самой библиотеке TBB, достаточно всем нам хорошо известной конструкции try-catch (как в последовательном коде). Но и в этом вопросе необязательно ограничиваться базовыми вещами. В заключение этой главы мы опишем, как реализовать собственные исключения, дополняющие TBB, и разберемся, как механизмы отмены и обработки исключений взаимодействуют на внутреннем уровне.

Даже если вы скептически относитесь к исключениям, поскольку принадлежите к школе, предпочитающей «коды ошибок», не бросайте эту главу – быть может, мы сумеем убедить вас в преимуществах обработки исключений для разработки надежных отказоустойчивых параллельных приложений с применением TBB.

Как отменить коллективную работу

Существуют ситуации, когда нужно отменить некоторую работу. Причины могут быть как внешними (пользователь нажал кнопку отмены в графическом интерфейсе), так и внутренними (искомый элемент уже найден, так что продолжать поиск бессмысленно). Все мы видели такие ситуации в последовательном коде, но они встречаются и в параллельных приложениях. Например, в некоторых алгоритмах дорогостоящей глобальной оптимизации применяется параллельный паттерн ветвей и границ, когда пространство поиска организовано в виде дерева и требуется отменить задачи, обходящие определенные ветви, если решение, скорее всего, находится в другой ветви.

Посмотрим, как воспользоваться отменой в несколько искусственном примере: требуется найти позиции единственного элемента –2 в векторе целых чисел. Пример искусственный, потому что мы положили `data[500]=-2`, т. е. наперед знаем результат. На рис. 15.1 показана реализация на основе алгоритма `parallel_for`.

Идея заключается в том, чтобы отменить все прочие конкурентные задачи, участвующие в выполнении `parallel_for`, как только какая-то нашла, что `data[500]==-2`. Так что же делает вызов `task::self().cancel_group_execution()`? Функция `task::self()` возвращает ссылку на самую внутреннюю задачу из тех, что выполняет вызывающий поток. Задачи рассматриваются в нескольких главах, но детали были раскрыты в главах 10–14. Там мы видели некоторые функции-члены класса `task`, и `cancel_group_execution()` – еще одна из них. Как следует из названия, эта функция отменяет не только вызывающую задачу, но и **все** задачи, принадлежащие той же группе.

В данном случае группа задач состоит из всех задач, участвующих в выполнении алгоритма `parallel_for`. Отменив эту группу, мы останавливаем все такие задачи и по существу прерываем параллельный поиск. Представьте себе, что

задача, которая нашла, что data[500]==-2, кричит всем своим братьям: «Ребята, есть! Прекращайте поиск!» Вообще говоря, каждый TBB-алгоритм создает собственную группу задач, и каждая задача, участвующая в его выполнении, принадлежит этой группе. Таким образом, любая задача из группы (алгоритма) может отменить весь алгоритм.

```
std::vector<int> data(n);
data[500] = -2;
int index = -1;
auto t1 = tbb::tick_count::now();
tbb::parallel_for(tbb::blocked_range<int>{0, n},
  [&](const tbb::blocked_range<int>& r){
      for(int i=r.begin(); i!=r.end(); ++i){
        if(data[i] == -2) {
          index = i;
          tbb::task::self().cancel_group_execution();
          break;
        }
      }
});
auto t2 = tbb::tick_count::now();
std::cout << "Index "     << index;
std::cout << " found in " << (t2-t1).seconds() << " seconds!\n";
```

Рис. 15.1 ❖ Нахождение индекса элемента, равного –2

Для вектора длиной n = 1 000 000 000 этот цикл занимает 0.01 с, и печатается примерно такое сообщение:

```
Index 500 found in 0.01368 seconds!
```

Однако если вызов task::self().cancel_group_execution() закомментировать, то выполнение на том же ноутбуке занимает уже 1.56 с.

Вот и все дела. Это все, что нужно знать об отмене (базовой) TBB-алгоритма. Но теперь, когда у нас появилась ясная мотивация для отмены задач (ускорение более чем в 100 раз в предыдущем примере!), возникает желание более основательно разобраться в том, как работает отмена и можно ли точнее управлять тем, какие задачи будут отменены.

Отмена задач в деталях

В главе 14 мы познакомились с классом task_group_context. Каждая задача принадлежит одному и только одному экземпляру task_group_context, который мы для краткости будем обозначать TGC. TGC представляет группу задач, которую можно отменить и для которой можно задать приоритет. В главе 14 мы на примерах показали, как изменить приоритет TGC. Мы также сказали, что объект TGC можно передавать высокоуровневым алгоритмам, например parallel_for или потоковому графу, в качестве необязательного аргумента. Так, на рис. 15.2 показан другой способ написать код рис. 15.1.

```
tbb::task_group_context tg;
…
tbb::parallel_for(tbb::blocked_range<int>{0, n},
  [&](const tbb::blocked_range<int>& r){
      for(int i=r.begin(); i!=r.end(); ++i){
        if(data[i] == -2){
          index = i;
          tg.cancel_group_execution();
          break;
        }
      }
}, tg);
```

Рис. 15.2 ❖ Альтернативная реализация кода рис. 15.1

Здесь мы видим, что TGC `tg` создается и передается последним аргументом алгоритму `parallel_for`, а затем используется при вызове `tg.cancel_group_execution()` (теперь с помощью функции-члена класса `task_group_context`).

Заметим, что код на рис. 15.1 и 15.2 в точности эквивалентен. Необязательный параметр `tg`, передаваемый в качестве последнего аргумента `parallel_for`, просто открывает возможность для более тонкой разработки. Предположим, к примеру, что мы передаем ту же самую переменную `tg` алгоритму `parallel_pipeline`, запущенному в параллельном потоке. Теперь любая задача, участвующая в выполнении `parallel_for` или `parallel_pipeline`, может вызвать `tg.cancel_group_execution()`, чтобы отменить оба параллельных алгоритма.

Задача может также узнать, какому экземпляру TGC она принадлежит, вызвав функцию-член `group()`, которая возвращает указатель на TGC. Поэтому мы можем безопасно добавить следующую строку в лямбда-выражение, переданное `parallel_for` на рис. 15.2: `assert(task::self().group()==&tg);`. Это означает, что следующие три строки в точности эквивалентны, и любую из них можно включить в код на рис. 15.2:

```
tg.cancel_group_execution();
tbb::task::self().group()->cancel_group_execution();
tbb::task::self().cancel_group_execution();
```

Когда некоторая задача запрашивает отмену всего TGC, запущенные задачи, ожидающие в очередях, завершаются, не начав выполнения, но те, что уже работают, не прерываются планировщиком, потому что, как мы помним, планировщик в TBB невытесняющий. Таким образом, прежде чем передать управление функции `task::execute()`, планировщик проверяет флаг отмены в TGC этой задачи и решает, нужно ли ее выполнять или весь TGC отменен. Но если уж задача получила управление, то она будет сохранять его, пока не соизволит вернуть его планировщику. Если же мы все-таки хотим расправиться и с работающими задачами, то каждая задача может опрашивать состояние отмены одним из двух способов:

```
if (task::self().group()->is_group_execution_cancelled()) return;
if (task::self().is_cancelled()) return;
```

Следующий вопрос: к какому TGC приписываются новые задачи? Разумеется, у нас есть средства для управления этим отображением, но определено также поведение по умолчанию, о котором стоит знать. Сначала рассмотрим, как назначать TGC задачам вручную.

Явное назначение TGC

Мы уже видели, как можно создать объект TGC и передать его высокоуровневому параллельному алгоритму (`parallel_for`,...) и высокоуровневому API задач (`allocate_root()`). Напомним, что в главе 10 мы представили также класс `task_group` в качестве API среднего уровня, который позволяет создавать задачи с общим TGC, допускающие отмену или задание приоритета одним движением. Все задачи, запускаемые с помощью одной и той же функции-члена `task_group::run()`, принадлежат одному TGC, и потому любая задача из группы может отменить всю банду.

В качестве примера рассмотрим код на рис. 15.3, где переписан параллельный поиск заданного значения, «скрытого» в векторе, с возвратом индекса найденного элемента. На этот раз мы вручную реализуем метод «разделяй и властвуй», применив возможности `task_group` (алгоритм `parallel_for` под капотом делает то же самое, хотя мы этого и не видим).

Для простоты вектор `data`, результирующий индекс `myindex`, и группа задач `g` сделаны глобальными переменными. Этот код рекурсивно разбивает пространство поиска, пока не будет достигнута некоторая степень детализации (величина отсечения `cutoff`, которую мы видели в главе 10). Это параллельное разбиение реализует функция `ParallelSearch(begin,end)`. Когда степень детализации оказывается достаточно малой (в этом примере – 100 итераций), вызывается функция `SequentialSearch(begin,end)`. Если искомое значение –2 найдено в одном из диапазонов, которые обходит `SequentialSearch`, все запущенные задачи отменяются с помощью вызова `g.cancel()`. На нашем 4-ядерном ноутбуке при N, равном 10 млн, программа напечатала:

```
SerialSearch: 5000000 Time: 0.012667
ParallelSearch: 5000000 Time: 0.000152 Speedup: 83.3355
```

Значение –2 было найдено в элементе с индексом 5 000 000. Полученное ускорение по сравнению с последовательным кодом – в 83 раза – может ошеломить. Но в данной ситуации параллельной реализации пришлось выполнять меньше работы, чем последовательной: после того как ключ найден, продолжать обход вектора `data` уже не нужно. В рассмотренном примере искомое значение находилось в середине вектора, в позиции N/2, и последовательная версия должна была дойти до этой точки, тогда как параллельная начинает поиск в разных позициях: 0, N/4, N/2, N·3/4 и т. д.

```
int grainsize = 100;
std::vector<int> data;
int myindex=-1;
tbb::task_group g;

void SerialSearch(long begin, long end){
  for(int i=begin; i<end; ++i){
    if(data[i] == -2){
      myindex = i;
      g.cancel();
      break;
    }
  }
}

void ParallelSearch(long begin, long end){
  if((end-begin) < grainsize){ // эквивалент отсечения
    return SerialSearch(begin, end);
  }
  else{
    long mid = begin + (end-begin)/2;
    g.run([&]{ParallelSearch(begin, mid);}); // запустить задачу
    g.run([&]{ParallelSearch(mid, end);});   // запустить еще одну задачу
  }
}

int main(int argc, char** argv)
{
  int n = (argc>1) ? atoi(argv[1]) : 1000;
  data.resize(n);
  data[n/2] = -2;

  auto t0 = tbb::tick_count::now();
  SerialSearch(0, n);
  auto t1 = tbb::tick_count::now();
  ParallelSearch(0, n);
  g.wait();      // ждать завершения всех запущенных задач
  auto t2 = tbb::tick_count::now();
  double t_s = (t1 - t0).seconds();
  double t_p = (t2 - t1).seconds();

  cout << "SerialSearch:   " << myindex << " Time: " << t_s << endl;
  cout << "ParallelSearch: " << myindex << " Time: " << t_p
       << " Speedup: "       << t_s/t_p << endl;
  return 0;
}
```

Рис. 15.3 ❖ Ручная реализация параллельного поиска с использованием класса `task_group`

Если вы возрадовались сверх меры, глядя на достигнутое ускорение, успокойтесь и давайте посмотрим, нельзя ли еще улучшить результат. Напомним, что `cancel()` не умеет завершать уже работающие задачи. Однако работающая

задача может узнать, не отменила ли выполнение какая-то другая задача в том же TGC. Для этого нужно лишь добавить строчку

```
if(g.is_canceling()) return;
```

в начало функции `ParallelSearch()`. Этот вроде бы совсем незначительный ход приводит к таким результатам:

```
SerialSearch: 5000000 Time: 0.012634
ParallelSearch: 5000000 Time: 2e-06 Speedup: 6317
```

Эх, если бы всегда можно было получить такой выигрыш от распараллеливания на 2-ядерной машине!

Примечание. Дополнительная редко используемая возможность: помимо явного создания `task_group`, задания TGC для параллельного TBB-алгоритма и задания TGC для корневой задачи с помощью функции `allocate_root`, можно также изменить TGC любой задачи, воспользовавшись ее функцией-членом

```
void task::change_group(task_group_context& ctx);
```

а поскольку мы также можем узнать TGC любой задачи с помощью функции `task::group()`, то получаем возможность переместить любую задачу в TGC любой другой задачи. Например, если две задачи имеют доступ к переменной `TGC_X` (скажем, имеется глобальная переменная `task_group_context *TGC_X`) и первая задача выполнила код

```
TGC_X=task::self().group();
```

то вторая задача может выполнить такой код:

```
task::self().change_group(*TGC_X);
```

Назначение TGC по умолчанию

А что, если TGC явно не задан? Тогда вступают в силу правила по умолчанию.

- Поток, создавший объект `task_scheduler_init` (явно или неявно в результате использования TBB-алгоритма), создает свой собственный TGC, помеченный как «**изолированный**». Первая задача, выполненная этим потоком, принадлежит этому TGC, а последующие дочерние задачи наследуют тот же TGC от родителя.
- Если любая из этих задач вызывает параллельный алгоритм, не передав TGC в виде необязательного аргумента (например, `parallel_for`, `parallel_reduce`, `parallel_do`, `pipeline`, потоковый граф и т. д.), то для новых задач, участвующих в выполнении этого вложенного алгоритма, неявно создается новый TGC, помеченный как «**связанный**». Таким образом, этот TGC является потомком, связанным с изолированным родительским TGC.
- Если задача, участвующая в выполнении параллельного алгоритма, вызывает вложенный параллельный алгоритм, то для последнего создается новый связанный дочерний TGC, родителем которого является TGC вызвавшей задачи.

На рис. 15.4 изображен лес деревьев TGC, автоматически построенный гипотетической программой.

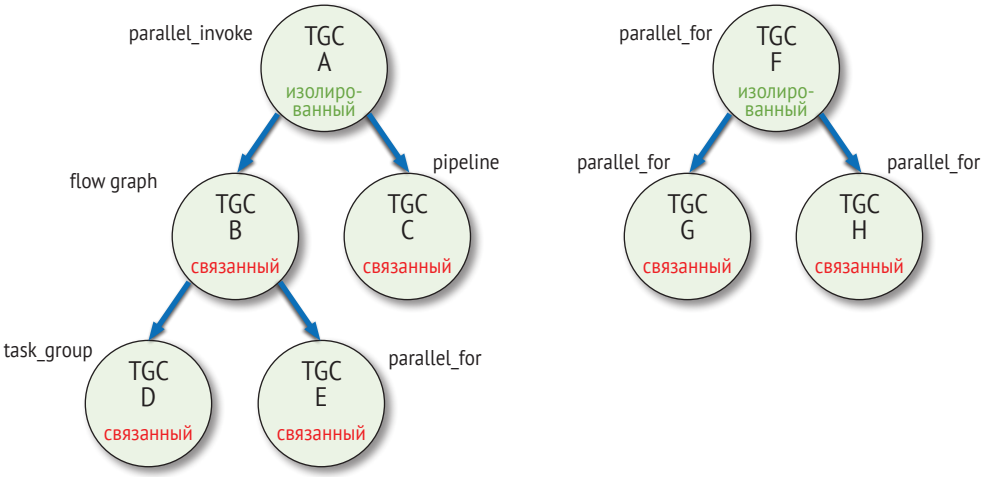


Рис. 15.4 ❖ Лес деревьев TGC, автоматически созданный в процессе выполнения гипотетического кода с использованием TBB

В этом гипотетическом коде пользователь хочет выполнить несколько вложенных TBB-алгоритмов, но ничего не знает о TGC, поэтому просто вызывает их, явно не передавая необязательный объект TGC. В одном мастер-потоке имеется обращение к алгоритму `parallel_invoke`, который автоматически инициализирует планировщик, создающий одну арену и первый изолированный TGC `A`. Затем внутри `parallel_invoke` создается два TBB-алгоритма: потоковый граф и конвейер. Для каждого из них автоматически создается и связывается с `A` по одному новому TGC – в данном случае `B` и `C`. В одном из узлов потокового графа создается группа `task_group`, а в другом – алгоритм `parallel_for`. Это приводит к созданию еще двух TGC, `D` и `E`, связанных с `B`. На этом закончено формирование первого дерева TGC – его корень изолирован, а все остальные TGC связанные, т. е. имеют родителя. Второе дерево строится другим мастер-потоком – он создает алгоритм `parallel_for` всего с двумя параллельными диапазонами, для каждого из которых вызывается вложенный `parallel_for`. И на этот раз корнем дерева является изолированный TGC `F`, а два других TGC, `G` и `H`, связанные. Заметим, что пользователь только писал код, в котором имеются вложенные TBB-алгоритмы, а лес TGC был автоматически создан самой библиотекой TBB. И не забывайте о задачах: каждый TGC разделяется несколькими задачами.

А что происходит, когда задача отменяется? Тут всё просто. Отменяется весь TGC, содержащий эту задачу, и отмена распространяется вниз по дереву. Например, если отменить задачу потокового графа (TGC `B`), то будет отменена также группа `task_group` (TGC `D`) и `parallel_for` (TGC `E`), как показано на рис. 15.5. Это разумно: мы отменяем потоковый граф и все, созданное в нем. Пример несколько надуманный, потому что трудно найти реальное приложение с такой вложенностью алгоритмов. Однако он показывает, как связываются различные TGC, чтобы не нарушить столь высоко ценимую компонуемость TBB.

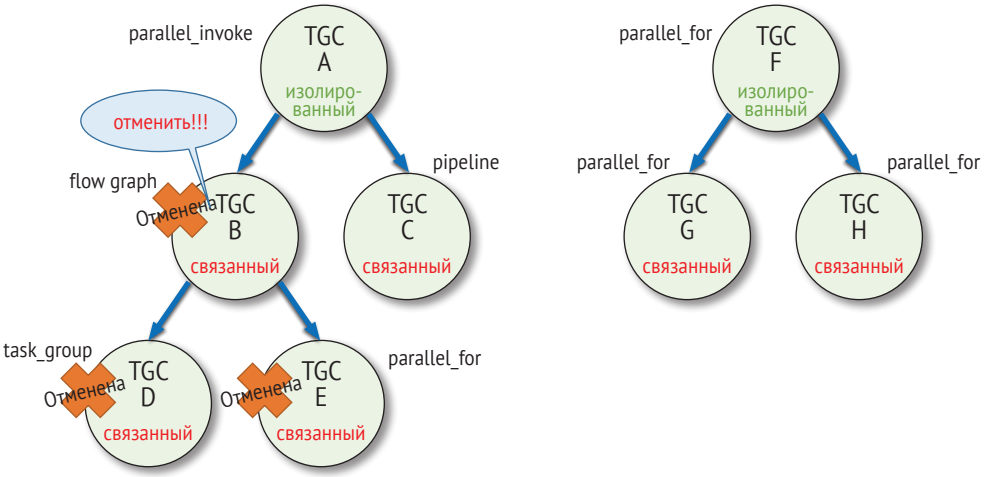


Рис. 15.5 ❖ Отмена вызывается из задачи, принадлежащей TGC B

Однако секундочку – а что, если мы захотим отменить потоковый граф и `task_group`, но оставить `parallel_for` (TGC `E`) в полном здравии? Что ж, это тоже возможно, нужно только вручную создать изолированный объект TGC и передать его в последнем аргументе алгоритму `parallel for`. То есть написать код типа показанного на рис. 15.6, где эта возможность реализована в узле `function_node` потокового графа `g`.

```
tbb::flow::function_node<float,float> node{g,…,[&](float a){
  tbb::task_group_context TGC_E(task_group_context::isolated);
  // вложенный parallel_for
  tbb::parallel_for(0, N, 1, [&](…){/*тело цикла*/}, TGC_E);
  return a;
}};
```

Рис. 15.6 ❖ Как отсоединить вложенный алгоритм от дерева TGC

Изолированный объект TGC `E` создается в стеке и передается последним аргументом алгоритму `parallel_for`. На рис. 15.7 показано, что теперь, даже если задача из потокового графа отменит свой TGC `B`, это действие распространится только до TGC `D` и не достигнет TGC `E`, потому что он при создании не был включен в дерево.

Точнее, изолированный TGC `E` теперь может стать корнем другого дерева в нашем лесу, т. е. родителем новых TGC, созданных более глубоко вложенными алгоритмами. Пример будет представлен в следующем разделе.

Подведем итоги. Если при создании TBB-алгоритма объект TGC не передается явно, то созданный по умолчанию лес TGC приводит к ожидаемому поведению в случае отмены. Однако этим поведением можно управлять самостоятельно, если создать нужное число объектов TGC и передать их алгоритмам. Например, мы можем создать один TGC, `A`, и передать его всем параллельным

алгоритмам, вызываемым в первом потоке гипотетической программы. В таком случае все задачи, принимающие участие в выполнении всех алгоритмов, будут принадлежать этому TGC A, как показано на рис. 15.8. Если теперь отменить любую задачу потокового графа, то будут отменены не только вложенные `task_group` и алгоритмы `parallel_for`, но все вообще алгоритмы, разделяющие TGC A.

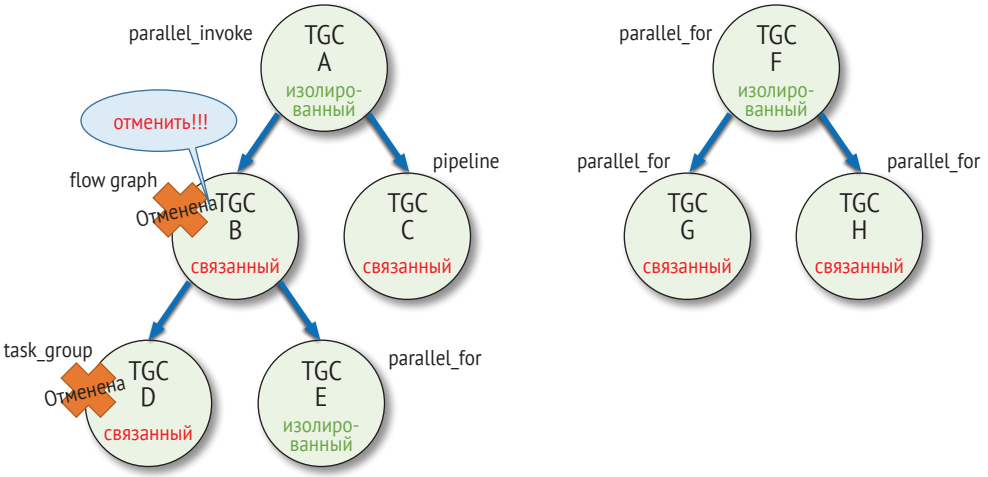


Рис. 15.7 ❖ Теперь TGC E изолирован и не подлежит отмене

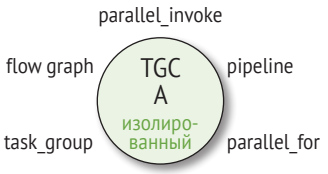


Рис. 15.8 ❖ После модификации гипотетического кода, при которой один и тот же TGC A передается всем параллельным алгоритмам

И напоследок мы хотим подчеркнуть, что эффективное отслеживание связей в лесу объектов TGC – далеко не тривиальная задача. Интересующийся читатель может обратиться к статье Андрея Марочко и Алексея Куканова (см. раздел «Дополнительная информация»), где подробно описаны проектные решения и внутренние детали реализации. А нам важно понимать, что были предприняты неординарные усилия, чтобы внутренняя кухня TGC не влияла на производительность в случае, когда отмена не нужна.

Обработка исключений в TBB

Примечание. Если вы ничего не знаете об исключениях в C++, то следующий пример иллюстрирует основные идеи:

```
int main(){
  try{
    try{
      throw 5;
    }
    catch(int& n){
      cout << "Re-throwing value: "<< n << andl;
      throw;
    }
  }
  catch(int& e){
    cout<< "Value caught: "<< e << endl;
  }
  catch (...){
    cout << "Exception occurred\n";
  }
}
```

В результате выполнения этого кода печатается

```
Re-throwing value: 5
Value caught: 5
```

Как видим, в первом блоке try имеется вложенный блок try-catch. В нем возбуждается исключение – целое число 5. Поскольку тип в блоке catch совпадает с типом исключения, этот блок становится обработчиком исключения. В нем мы печатаем полученное значение и повторно возбуждаем исключение, которое распространяется дальше. На внешнем уровне имеется два блока catch, но выполняется только первый, потому что тип аргумента в нем совпадает с типом исключения. Во втором блоке catch на внешнем уровне в качестве аргумента указано многоточие (...), поэтому он обрабатывает любое исключение, не обработанное ни одним из предшествующих блоков catch в цепочке. Например, если бы вместо 5 мы возбудили исключение 5.0, то было бы напечатано сообщение «Exception occurred».

Теперь, когда мы понимаем, что такое отмена – основной механизм поддержки исключений в TBB, – перейдем к существу вопроса. Наша цель – научиться разрабатывать пуленепробиваемый код с исключениями, как показано на рис. 15.9.

```
int main() {
  std::vector<int> data(1000);
  try{
    tbb::parallel_for(0, 2000, [&](int i) {data.at(i)++;});
  }
  catch(std::out_of_range& ex){
    std::cout << "Out_of_range: " << ex.what() << std::endl;
  }
  return 0;
}
```

Рис. 15.9 ❖ Простой пример обработки исключений в TBB

Ладно, быть может, этот код еще не совсем пуленепробиваемый, но в качестве первого примера сойдет. Суть в том, что вектор `data` содержит всего 1000 элементов, а алгоритм `parallel_for` настаивает на переборе элементов вплоть до 2000 – 1. А чтобы еще усугубить положение, доступ к `data` осуществляется не в виде `data[i]`, а с помощью метода `data.at(i)`, который, в отличие от первого способа, добавляет проверку выхода за границы и возбуждает исключение `std::out_of_range`, если мы нарушим правила. Поэтому после компиляции и выполнения кода на рис. 15.9 мы увидим сообщение

```
Out_of_range: vector
```

Как мы знаем, для параллельного инкремента элементов `data` запускается несколько задач. Некоторые из них попытаются увеличить элементы с индексами больше 999. Задача, которая первой обратится к несуществующему элементу, например `data.at(1003)++`, очевидно, должна быть отменена. Функция-член `std::vector::at()` вместо инкремента несуществующего элемента с индексом 1003 возбуждает исключение `std::out_of_range`. Поскольку объект исключения не перехвачен самой задачей, он распространяется вверх по стеку вызовов и достигает планировщика TBB. Планировщик перехватывает исключение и отменяет все конкурентные задачи с таким же TGC (мы уже знаем, как отменяется TGC целиком). Кроме того, копия объекта исключения сохраняется в структуре данных TGC. Когда все задачи с данным TGC отменены, приходит конец и самому TGC, в результате чего исключение повторно возбуждается в потоке, где этот TGC был создан. В нашем примере это поток, который вызвал `parallel_for`. Но `parallel_for` находится внутри блока try-catch, который перехватывает объект `out_of_range`. Это означает, что этот блок становится обработчиком исключения и печатает сообщение. Функция-член `ex.what()` возвращает строку, содержащую описание исключения.

Примечание. Деталь реализации. Компилятор ничего не знает о многопоточной природе параллельного алгоритма TBB. Следовательно, помещение такого алгоритма внутрь блока try защищает только вызывающий поток (мастер-поток), но рабочие потоки продолжат исполнять задачи, которые тоже могут возбуждать исключения. Чтобы решить эту проблему, планировщик уже включает блоки try-catch, чтобы любой рабочий поток мог перехватить исключения, вышедшие за пределы его задач.

Функции catch() аргумент следует передавать по ссылке. В таком случае единственная функция catch, перехватывающая тип базового класса, сможет перехватывать и объекты всех производных от него классов. Например, на рис. 15.9 мы могли бы написать **catch**(std::exception& ex) вместо **catch**(std::out_of_range& ex), потому что класс std::out_of_range наследует классу std::logic_failure, который, в свою очередь, наследует std::exception, поэтому перехват по ссылке распространяется на всю цепочку наследования.

Не все компиляторы C++ поддерживают механизм распространения исключений, определенный в стандарте C++11. Точнее, если компилятор не поддерживает тип std::exception_ptr (как все компиляторы, вышедшие до выхода C++11), то TBB не сможет повторно возбудить точную копию объекта исключения. В таких случаях TBB сохраняет сводную информацию об исключении в объекте tbb::captured_exception, который уже можно возбудить повторно. Есть еще кое-какие детали, относящиеся к формированию сводной информации для различных видов исключений (std::exception, tbb::tbb_exception и прочих). Но, поскольку в наши дни трудно найти компилятор, не поддерживающий C++11, мы не станем уделять много внимания этому аспекту обратной совместимости TBB.

Написание собственных классов исключений TBB

В библиотеке TBB уже определен ряд классов исключений, все они перечислены в таблице на рис. B.77.

Но в некоторых случаях рекомендуется создавать собственные производные классы исключений для работы с TBB. Для этого можно воспользоваться абстрактным классом tbb::tbb_exception, показанным на рис. 15.10. На самом деле это интерфейс, поскольку в нем определено пять чисто виртуальных функций, которые мы обязаны переопределить в производном классе.

```
class tbb_exception: public std::exception{
            virtual tbb_exception* move() throw() = 0;
            virtual void destroy() throw() = 0;
            virtual void throw_self() = 0;
            virtual const char* name() throw() = 0;
            virtual const char* what() throw() = 0;
        };
```

Рис. 15.10 ❖ Определение собственного класса исключения, производного от tbb::tbb_exception

Опишем подробнее назначение чисто виртуальных функций в интерфейсе tbb_exception.

- move() должна создать указатель на копию объекта исключения, которая сможет пережить оригинал. Рекомендуется перемещать содержимое оригинала, особенно если тот подлежит уничтожению. Спецификация throw() после move() (как и после destroy(), what() и name()) информирует

компилятор о том, что эта функция не должна возбуждать никаких исключений.

- `destroy()` должна уничтожать копию, созданную `move()`.
- `throw_self()` должна возбуждать `*this`.
- `name()` обычно возвращает имя первоначально перехваченного исключения в составе информации времени выполнения о типе RTTI. Его можно получить с помощью оператора `typeid` и класса `std::type_info`. Например, можно было бы вернуть `typeid(*this).name()`.
- `what()` возвращает завершающуюся нулем строку с описанием исключения.

Но вместо того чтобы реализовывать все виртуальные функции, необходимые для наследования `tbb_exception`, проще и даже рекомендуется создавать собственные исключения с помощью шаблона класса `tbb::movable_exception`. Этот шаблон уже реализует все требуемые функции, так что пять описанных выше виртуальных функций теперь являются обычными функциями-членами, которые мы можем переопределить или оставить как есть. При этом в шаблоне имеются и другие функции, показанные в приведенном ниже фрагменте объявления:

```
template<typename ExceptionData>
class movable_exception: public tbb_exception{
public:
  movable_exception(const ExceptionData& src);
  ExceptionData& data() throw();
  ...
}
```

Конструктор `movable_exception` и функция-член `data()` будут объяснены ниже на примере. Предположим, что деление на 0 является тем исключительным событием, которое мы хотим перехватывать явно. На рис. 15.11 показано, как создать собственный класс исключения на основе шаблона класса `tbb::movable_exception`.

Мы создаем свой класс `div_ex`, содержащий данные, которые хотим передавать вместе с исключением. В этом случае полезной нагрузкой является целое число, в котором хранится место, где произошло деление на 0. Теперь мы можем создать объект `de` класса `movable_exception`, конкретизированного аргументом шаблона `div_ex`:

```
tbb::movable_exception<div_ex> de{div_ex{i}};
```

Как видим, в качестве аргумента конструктору `movable_exception<div_ex>` передается конструктор `div_ex` – `div_ex{i}`.

Позже, в блоке `catch`, мы перехватываем объект исключения как `ex`, и с помощью функции-члена `ex.data()` получаем ссылку на объект `div_ex`. Таким образом, мы имеем доступ к переменным-членам и функциям-членам, определенным в `div_ex`: `name()`, `what()` и `it`. Для входного параметра `n=1000000` этот пример напечатает:

```
Exception name: div_ex
Exception: Division by 0! at position: 500000
```

```
class div_ex
{
  public:
    int it;
    explicit div_ex(int it_) : it{it_} {}
    const char* what() const throw(){
      return "Division by 0!";
    }
    const char* name() const throw(){
      return typeid(*this).name();
    }
};

int main(int argc, char** argv){
  int n = (argc>1) ? atoi(argv[1]): 1000;

  std::vector<float> data(n, 1.0);
  data[n/2] = 0.0;
  try {
    tbb::parallel_for(0, n, [&](int i){
      if (data[i]) data[i] = 1/data[i];
      else{
        tbb::movable_exception<div_ex> de{div_ex{i}};
        throw de;
      }
    });
  }
  catch(tbb::movable_exception<div_ex>& ex){
    std::cout << "Exception name: " << ex.data().name() << endl;
    std::cout << "Exception: "      << ex.data().what();
    std::cout << " at position: "   << ex.data().it << endl;
  }
  return 0;
}
```

Рис. 15.11 ❖ Удобный способ конфигурирования перемещаемого исключения

Мы добавили функции-члены `what()` и `name()` в класс `div_ex`, но они необязательны и от них можно отказаться, если они не нужны. В таком случае блок catch можно изменить следующим образом:

```
catch(tbb::movable_exception<div_ex>& ex){
  cout << "Division by 0 at pos: " << ex.data().it <<endl;
}
```

поскольку этот обработчик исключений будет выполняться только при получении исключения типа `movable_exception<div_ex>`, а оно возникает лишь при делении на 0.

Соберем все вместе: компонуемость, отмена и обработка исключений

Прежде чем закончить эту главу, приведем последний пример, демонстрирующий аспекты компонуемости TBB. На рис. 15.12 приведен фрагмент кода, где в цикле `parallel_for` должен был бы производиться обход строк матрицы `data`, если бы не тот факт, что на первой же итерации возбуждается исключение (содержащее строку «oops»)! Для каждой строки вложенный алгоритм `parallel_for` должен был бы параллельно обходить столбцы `data`.

```
bool data[N][N];
int main(){
  try{
    tbb::parallel_for(0, N, 1,
      [&](int i) {
        tbb::task_group_context root{task_group_context::isolated};
        tbb::parallel_for(0, N, 1,
          [&](int j){
            data[i][j] = true;
          }
          //, root // раскомментируйте и посмотрите, что будет!
        );
        throw "oops";
      });
  }

  catch(...){
    std::cout << "An exception captured " << std::endl;
  }

  return 0;
}
```

Рис. 15.12 ❖ Цикл `parallel_for`, вложенный во внешний `parallel_for`, который возбуждает исключение

Предположим, что четыре разные задачи выполняют четыре итерации внешнего цикла по `i` и вызывают внутренний `parallel_for`. В результате может образоваться дерево TGC, как на рис. 15.13.

Это означает, что когда на первой итерации внешнего цикла мы доходим до предложения `throw`, уже выполняется несколько внутренних циклов. Однако исключение на внешнем уровне распространяется вниз, отменяя по пути внешние параллельные циклы, что бы они ни делали. Видимый результат этой глобальной отмены заключается в том, что некоторые строки, находившиеся в процессе изменения значения с `false` на `true`, оказываются обработанными не до конца, т. е. часть значений в них уже равна `true`, а часть осталась равна `false`.

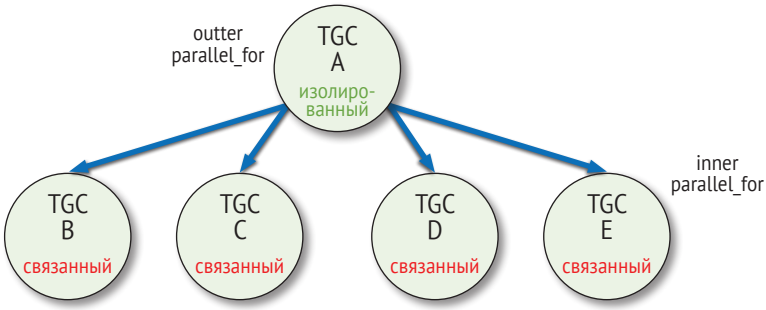


Рис. 15.13 ❖ Возможное дерево TGC для кода на рис. 15.12

Но взгляните, для каждой строки существует изолированный объект `task_group_context` с именем `root` – он создается в строчке

```
tbb::task_group_context root(task_group_context::isolated);
```

Если передать этот TGC в качестве последнего аргумента внутреннему `parallel_for`, раскомментировав строчку

```
          , root  // раскомментируйте и посмотрите, что будет!
```

то мы получим другую конфигурацию TGC, показанную на рис. 15.14.

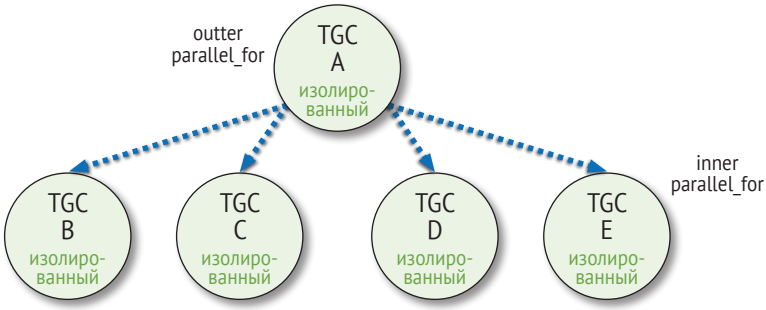


Рис. 15.14 ❖ Другая конфигурация TGC

В этой новой ситуации исключение вызывает отмену TGC `A`, в котором было возбуждено, но у этого TGC нет ни одного потомка, который можно было бы отменить. Если теперь проверить значения в массиве `data`, то мы увидим, что в каждой строке все элементы равны либо `true`, либо `false`, но никогда не встречаются комбинации того и другого, как в предыдущем случае. Это объясняется тем, что после того как внутренний цикл начнет записывать значения `true` в некоторую строку, он обработает ее до конца, а не будет прерван посередине.

В более общем случае, если так можно выразиться о лесе деревьев TGC на рис. 15.4, что произойдет, когда вложенный алгоритм возбуждает исключение, не перехваченное ни на каком уровне? Например, предположим, что в дереве TGC на рис. 15.15 исключение возбуждено внутри потокового графа (TGC `B`).

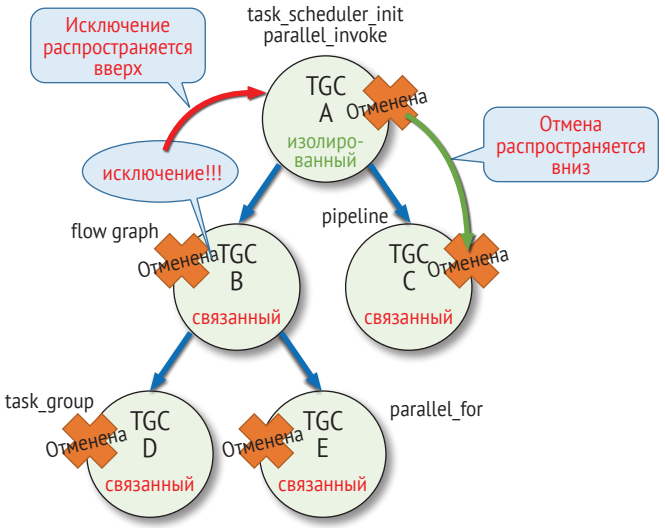


Рис. 15.15 ❖ Результат исключения, возбужденного во вложенном TBB-алгоритме

Разумеется, TGC `B` и его потомки `D` и `E` также отменяются. Это мы знаем. Но исключение распространяется вверх, и если на этом уровне оно тоже не перехвачено, то вызывает также отмену задач в TGC `A`, а т. к. отмена распространяется вниз, то умирает заодно и TGC `C`. Отлично! Это ожидаемое поведение: единственное исключение, на каком бы уровне оно ни было возбуждено, отправляет в могилу весь параллельный алгоритм (как то было бы в последовательном случае). Предотвратить такую цепочку отмен можно, либо перехватив исключение на желаемом уровне, либо сконфигурировав вложенный алгоритм в изолированном TGC. Ну разве не прелесть?

Резюме

В этой главе мы видели, как просто отменить параллельный TBB-алгоритм и воспользоваться механизмом исключений для обработки ошибок во время выполнения. Оба средства правильно работают без какой-либо настройки, даже если мы ограничимся поведением по умолчанию. Мы также обсудили важное средство TBB, контекст группы задач – TGC. Это ключевой элемент в реализации отмены и обработки исключений в TBB, и им можно воспользоваться самостоятельно, чтобы получить больший контроль над обоими механизмами. Мы начали с рассмотрения операции отмены и объяснили, как одна задача может отменить весь TGC, которому принадлежит. Затем мы рассмотрели, как вручную задать TGC, на который отображается задача, а также правила, действующие, когда такое отображение не определено разработчиком. Правила, подразумеваемые по умолчанию, приводят к ожидаемому поведению: если параллельный алгоритм отменяется, то отменяются и все вложенные в него параллельные алгоритмы. Затем мы перешли к обработке исключений. И здесь

поведение исключений в TBB похоже на исключения в последовательном коде, хотя внутренняя реализация гораздо сложнее, т. к. исключение, возбужденное в одной задаче, исполняемой одним потоком, может быть перехвачено в другом потоке. Если компилятор поддерживает стандарт C++11, то между потоками можно переместить точную копию исключения, в противном случае сводная информация об исключении запоминается в объекте `tbb::captured_exception`, чтобы исключение можно было повторно возбудить в параллельном контексте. Мы также описали, как написать собственный класс исключения, воспользовавшись шаблоном класса `tbb::movable_exception`. Наконец, в конце главы мы продемонстрировали, как взаимодействуют компонуемость, отмена и обработка исключений.

Дополнительная информация

Ниже приведены ссылки на дополнительные материалы, относящиеся к теме этой главы.

- *A. Marochko, and A. Kukanov.* Composable Parallelism Foundations in the Intel Threading Building Blocks Task Scheduler // Advances in Parallel Computing. Vol 22. 2012.
- *Deb Haldar.* Top 15 C++ Exception handling mistakes and how to avoid them. URL: www.acodersjourney.com/2016/08/top-15-c-exception-handling-mistakes-avoid/.

Глава 16

Настройка TBB-алгоритмов: зернистость, локальность, параллелизм и детерминированность

В главе 2 мы описали обобщенные параллельные алгоритмы, имеющиеся в библиотеке TBB, и привели несколько примеров их использования. При этом мы отметили, что алгоритмы и по умолчанию ведут себя достаточно хорошо, но сказали, что на случай необходимости имеются способы настройки производительности. В данной главе мы вернемся к этому обещанию и поговорим о некоторых важных средствах, позволяющих изменить поведение алгоритмов по умолчанию.

На первом месте в нашем обсуждении будет стоять три вопроса. Первый – зернистость, т. е. объем выполняемой задачей работы. TBB умеет эффективно планировать задачи, но мы должны продумывать размер задач, создаваемых алгоритмами, потому что он может оказывать сильное влияние на производительность, особенно если задачи очень малы или велики. Второй вопрос – локальность данных. В предисловии мы рассказали о том, что способ использования кешей и памяти приложением может решить его судьбу. И последний вопрос – располагаемый параллелизм. Разумеется, при использовании TBB наша цель – распараллелить программу, но это нельзя делать слепо, без учета зернистости и локальности. Настройка производительности приложения зачастую становится упражнением на нахождение баланса между этими тремя аспектами.

Одно из ключевых различий между TBB-алгоритмами и другими интерфейсами, к примеру Parallel STL, заключается в том, что первые предлагают точки подключения и средства, позволяющие оказывать влияние на их поведение в трех вышеупомянутых отношениях. TBB-алгоритмы – не черные ящики, которые мы никак не можем контролировать!

В этой главе мы сначала обсудим зернистость задач и сформулируем эвристическое правило о том, какой размер задачи можно считать достаточным. Затем мы перейдем к простым циклическим алгоритмам и использованию

диапазонов и разбивателей для управления зернистостью задач и локальностью данных. Мы также кратко обсудим детерминированность и ее влияние на гибкость при настройке производительности. И в заключение обратимся к конвейерному TBB-алгоритму и посмотрим, какие в нем имеются средства для задания зернистости, локальности данных и максимальной степени параллелизма.

Зернистость задач: какой размер достаточен?

Чтобы предоставить библиотеке TBB максимальную гибкость при балансировке нагрузки между потоками, мы хотим разделить работу алгоритма на как можно большее количество частей. В то же время, чтобы минимизировать накладные расходы на заимствование работ и планирование задач, мы хотим, чтобы задачи были как можно больше. Поскольку эти цели прямо противоположны, оптимальной производительности алгоритм достигает где-то посередине.

Но это еще не все – точный оптимальный размер задачи зависит от платформы и приложения, поэтому невозможно предложить универсальную инструкцию. И тем не менее полезно иметь приблизительную оценку, которой можно было бы хоть как-то руководствоваться. Понимая все привходящие обстоятельства, мы все же рискуем предложить такое эвристическое правило:

Эвристическое правило. В среднем задачи TBB должны выполняться дольше 1 мкс, чтобы затраты на заимствование работ были незаметны. Это несколько тысяч тактов ЦП, и если вы предпочитаете выражение в тактах, то мы предлагаем взять за основу 10 000 тактов.

Важно помнить, что не каждая задача обязана выполняться дольше 1 мкс, да зачастую это и невозможно. Например, в алгоритмах типа «разделяй и властвуй» можно было бы использовать малые задачи для разбиения работы и более крупные в листовых узлах. Именно так работает алгоритм `parallel_for`. Задачи TBB используются как для разбиения диапазона, так и для применения тела к конечным поддиапазонам. У задач разбиения обычно очень мало работы, тогда как задачи тела цикла гораздо объемнее. В таком случае мы не можем гарантировать, что все задачи будут выполняться дольше 1 мкс, но можем стремиться к тому, чтобы средний размер был больше этой величины.

Используя алгоритмы типа `parallel_invoke` или напрямую задачи TBB, мы полностью контролируем размер задач. Например, в главе 2 мы реализовали параллельную версию быстрой сортировки с помощью `parallel_invoke` и переходили от рекурсивной параллельной реализации к последовательной, когда размер массива (а значит, и время выполнения задачи) оказывался ниже порога отсечения:

```
if (end - begin < cutoff) {
  serialQuicksort(begin, end);
}
```

При работе с простыми циклическими алгоритмами – `parallel_for`, `parallel_reduce` и `parallel_scan` – передаваемые в качестве аргументов диапазон и разбиватель дают нам искомые средства управления. Подробнее мы поговорим об этом в следующем разделе.

Выбор диапазонов и разбивателей для циклов

Как было сказано в главе 2, диапазон представляет собой рекурсивно разбиваемое множество значений, обычно пространство итераций цикла. Диапазоны используются совместно с простыми циклическими алгоритмами: `parallel_for`, `parallel_reduce`, `parallel_deterministic_reduce` и `parallel_scan`. TBB-алгоритм разбивает свой диапазон и применяет тело алгоритма к поддиапазонам с помощью задач TBB. В сочетании с разбивателями диапазоны дают простой, но действенный способ представить пространство итераций и управлять его разбиением на задачи и назначением задач рабочим потокам. Это разбиение можно использовать для настройки зернистости задач и локальности данных.

Чтобы считаться диапазоном, класс должен моделировать концепцию диапазона, показанную на рис. 16.1. Диапазон можно копировать, его можно расщепить с помощью *расщепляющего конструктора* и факультативно предоставить *пропорционально расщепляющий конструктор*. Кроме того, диапазон должен содержать методы для проверки на пустоту и расщепляемость, а также булеву константу, равную `true`, если определен пропорционально расщепляющий конструктор.

Псевдосигнатура	Семантика
`R::R(const R&)`	Копирующий конструктор
`R::~R()`	Деструктор
`bool R::empty() const`	`true`, если диапазон пуст
`bool R::is_divisible() const`	`true`, если диапазон можно разбить на два поддиапазона
`R::R(R& r, split)`	Базовый расщепляющий конструктор. Разбивает `r` на два поддиапазона
`R::R(R& r, proportional_split_proportion)`	**Факультативно.** Пропорционально расщепляющий конструктор. Разбивает `r` на два поддиапазона в соответствии с заданной пропорцией
`static const bool R::is_splittable_in_proportion`	**Факультативно.** Если `true`, то пропорционально расщепляющий конструктор определен для данного диапазона и может использоваться в параллельных алгоритмах

Рис. 16.1 ❖ Концепция диапазона

Мы можем определить собственные типы диапазонов, но библиотека TBB уже предоставляет блочные диапазоны, приведенные на рис. 16.2, которых хватает в большинстве ситуаций. Например, пространство итераций показанного ниже вложенного цикла можно представить диапазоном `blocked_range2d<int, int> r(i_begin, i_end, j_begin, j_end )`:

```
for (int i = i_begin; i < i_end, ++i )
  for (int j = j_begin; j < j_end; ++j )
    /* тело цикла */
```

Тип диапазона	Аргументы конструктора	Описание

`blocked_range`	`Value begin,`
	`Value end,`
	`[size_t grainsize]`
		Моделирует одномерный диапазон

`blocked_range2d`	`RowValue row_begin,`
	`RowValue row_end,`
	`[size_type row_grainsize],`
	`ColValue col_begin,`
	`ColValue col_end,`
	`[size_type col_grainsize]`
		Моделирует двумерный диапазон. После многократного расщепления поддиапазоны стремятся к отношению степеней детализации по строкам и по столбцам

`blocked_range3d`	`PageValue page_begin,`
	`PageValue page_end,`
	`[size_type page_grainsize],`
	`RowValue row_begin,`
	`RowValue row_end,`
	`[size_type row_grainsize],`
	`ColValue col_begin,`
	`ColValue col_end,`
	`[size_type col_grainsize]`
		Моделирует трехмерный диапазон. После многократного расщепления поддиапазоны стремятся к отношению степеней детализации по страницам, строкам и столбцам

`blocked_rangeNd`	`const blocked_range<Value>& dim`$_0$`,`
	`…`
	`const blocked_range<Value>& dim`$_{N-1}$`,`
		Ознакомительная возможность – моделирование N-мерного диапазона. Необходима поддержка C++11. После многократного расщепления поддиапазоны стремятся к отношению степеней детализации по N диапазонам `blocked_range`

Рис. 16.2 ❖ Блочные диапазоны, предоставляемые библиотекой TBB

Для интересующихся читателей мы опишем, как определить собственный тип диапазона, в разделе «В глубоких водах» ниже.

Обзор разбивателей

Помимо диапазонов, TBB-алгоритмы поддерживают разбивателей, описывающих, как алгоритм должен разбивать свой диапазон. На рис. 16.3 перечислены различные типы разбивателей.

Разбиватель	Описание	Когда используется совместно с blocked_range(i,j,g)
`simple_partitioner`	Размер порции ограничен степенью детализации	$g/2 \leqslant$ *размер_порции* $\leqslant g$
`auto_partitioner` (по умолчанию)	Размер порции выбирается автоматически	$g/2 \leqslant$ *размер_порции*
`affinity_partitioner`	Автоматически выбираются размер порции, привязка к кешу и начальное равномерное распределение итераций	
`static_partitioner`	Детерминированный размер порции, привязка к кешу и начальное равномерное распределение итераций без балансировки нагрузки. Равномерное распределение создается, если степень детализации не запрещает создать P порций	$\max(g/3, N/P) \leqslant$ *размер_порции*, где: N – размер проблемы; P – количество ресурсов

Рис. 16.3 ❖ Разбиватели, предоставляемые библиотекой TBB

Класс simple_partitioner используется для рекурсивного разбиения диапазона до тех пор, пока функция is_divisible не вернет false. Для диапазонов блочного типа это означает, что разбиение производится, пока размер диапазона не окажется меньше или равен степени детализации. Если степень детализации тщательно настроена (мы поговорим об этом в следующем разделе), то лучше выбирать simple_partitioner, поскольку он гарантирует, что конечные поддиапазоны будут учитывать заданные степени детализации.

В классе auto_partitioner применяется динамический алгоритм, который расщепляет диапазон достаточно мелко с точки зрения балансировки нагрузки, но необязательно настолько мелко, насколько позволяет is_divisible. При использовании совместно с блочными диапазонами степень детализации попрежнему задает нижнюю границу размера конечных порций, но гораздо менее важна, потому что auto_partitioner может остановиться на большей степени детализации. Поэтому общепринято указывать степень детализации 1 и дать возможность auto_partitioner самому определить оптимальную величину. В TBB 2019 в алгоритмах parallel_for, parallel_reduce и parallel_scan по умолчанию используется разбиватель auto_partitioner со степенью детализации 1.

Класс static_partitioner распределяет диапазон по рабочим потокам настолько равномерно, насколько возможно, и больше никакой балансировки нагрузки не производит. Распределение работ и отображение на потоки детерминированы и зависят только от количества итераций, степени детализации и количества потоков. Из всех разбивателей static_partitioner имеет самые низкие накладные расходы, поскольку не принимает никаких динамических решений. Его использование может также улучшить поведение кеша, потому что при исполнении одного цикла паттерны планирования повторяются. Однако static_partitioner резко ограничивает возможности балансировки нагрузки, так что использовать его следует осмотрительно. В разделе «Использование static_partitioner» мы подробнее остановимся на достоинствах и недостатках static_partitioner.

Класс affinity_partitioner объединяет лучшие черты auto_partitioner и static_partitioner и улучшает привязку к кешу, если один и тот же объект разбивателя повторно используется при повторном выполнении цикла для одного и того же набора данных. Как и static_partitioner, affinity_partitioner сначала создает равномерное распределение, но впоследствии допускает дополнительную балансировку нагрузки. Кроме того, он хранит историю того, какой поток какую порцию диапазона обрабатывал, и пытается воспроизвести этот порядок при последующих выполнениях. Если набор данных целиком помещается в кеш процессора, то такое повторение может привести к значительному повышению производительности.

Выбирать ли степень детализации для управления зернистостью задач

В начале этой главы мы говорили о том, насколько важна может быть зернистость задач. Значит, при работе с диапазонами блочного типа мы всегда должны тщательно настраивать степень детализации, верно? Необязательно. Вы-

бор правильной степени детализации может быть очень важен или почти не важен – все зависит от используемого разбивателя.

Если используется `simple_partitioner`, то степень детализации, и только она, определяет размеры диапазонов, передаваемых телу. В этом случае диапазон рекурсивно разбивается до тех пор, пока `is_divisible` не вернет `false`. Все остальные разбиватели применяют внутренние алгоритмы для решения вопроса о том, когда прекратить разбиение. Для разбивателей, использующих `is_divisible` только как нижнюю границу, обычно достаточно задать степень детализации 1.

Чтобы продемонстрировать влияние степени детализации на различные разбиватели, мы напишем микротест производительности для `parallel_for`, в котором будем изменять количество итераций цикла (N), степень детализации, время выполнения одной итерации и тип разбивателя.

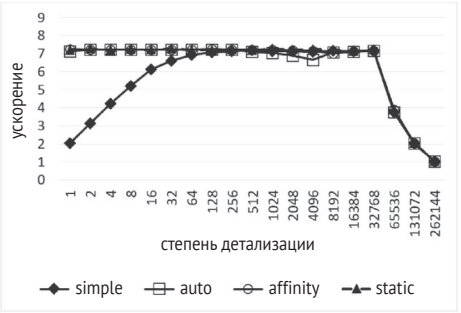
```
template< typename P >
static inline double executePfor(int num_trials, int N,
                                 int gs, P &p, double tpi) {
  tbb::tick_count t0;
  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    tbb::parallel_for (
      tbb::blocked_range<int>{0, N, gs},
      [tpi](const tbb::blocked_range<int> &r) {
        int e = r.end();
        for (int i = r.begin(); i < e; ++i) {
          spinWaitForAtLeast(tpi);
        }
      },
      p
    );
  }
  tbb::tick_count t1 = tbb::tick_count::now();
  return (t1 - t0).seconds()/num_trials;
}
```

Рис. 16.4 ❖ Функция для измерения времени выполнения `parallel_for`, принимающая количество итераций N, разбиватель (p), степень детализации (gs) и время работы одной итерации (tpi)

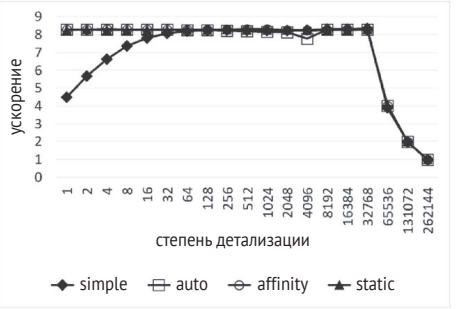
Все данные о производительности, представленные в этой главе, были получены на односокетном сервере с процессором Intel Xeon E3-1230 с четырьмя ядрами, поддерживающими по два аппаратных потока. Процессор имел базовую тактовую частоту 3,4 ГГц, разделяемый L3-кеш 8 МБ и L2-кеш размером 256 КБ для каждого ядра. Система работала под управлением SUSE Linux Enterprise Server 12. Все примеры компилировались компилятором Intel C++ Compiler 19.0, входящим в комплект поставки TBB 2019, с флагами «–std=c++11 –O2 –tbb».

На рис. 16.5 приведены результаты работы программы рис. 16.4 при $N = 2^{18}$ для каждого из имеющихся в TBB типов разбивателей с различными значе-

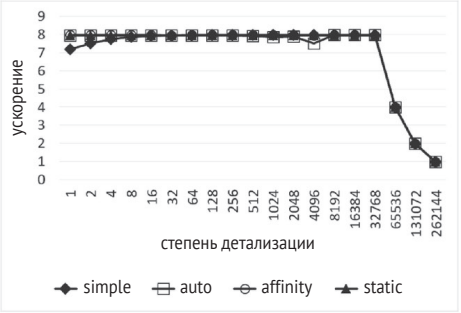
ниями степени детализации. Как видим, при очень малом времени работы одной итерации (10 нс) `simple_partitioner` близок к максимальной производительности других разбивателей, когда степень детализации >= 128. С увеличением времени одной итерации `simple_partitioner` очень быстро приближается к максимальной производительности, поскольку для компенсации накладных расходов на планирование нужно меньше итераций.



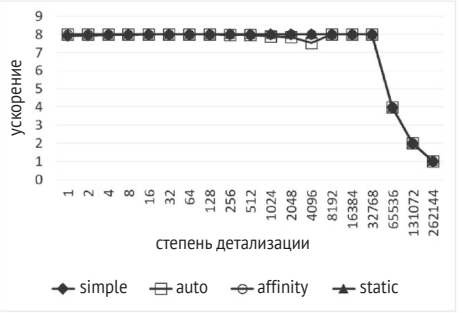
(a) Время одной итерации = 10 нс



(b) Время одной итерации = 100 нс



(c) Время одной итерации = 1 мкс



(d) Время одной итерации = 10 мкс

Рис. 16.5 ❖ Ускорение для различных типов разбивателей и увеличивающейся степени детализации. Общее количество итераций цикла равно 2^{18} = 262 144

Для всех типов разбивателей на рис. 16.5, кроме `simple_partitioner`, мы видим максимальную производительность при изменении степени детализации от 1 до 4096. На нашей платформе было 8 логических ядер, поэтому степень детализации должна быть меньше или равна $2^{18}/8$ == 32 768, чтобы каждому потоку досталась хотя бы одна порция, поэтому для всех производителей наблюдается резкий спад, когда степень детализации больше 32 768. Отметим также, что при степени детализации 4096 `auto_partitioner` и `affinity_partitioner` демонстрируют падение производительности на всех рисунках. Это объясняется тем, что выбор большой степени детализации ограничивает возможности этих алгоритмов, препятствуя доведению автоматического разбиения до конца.

Этот небольшой эксперимент подтверждает, что степень детализации критически важна для `simple_partitioner`. Мы можем использовать `simple_parti-`

tioner, чтобы вручную задать размер задач, но при этом должны подходить к выбору ответственно.

Второй урок заключается в том, что эффективное выполнение с ускорением, близким к линейной верхней границе, наблюдается, когда время выполнения тела близко к 1 мкс (10 нс × 128 = 1,28 мкс). Этот результат подтверждает эвристическое правило, сформулированное в начале главы! И это не должно вызывать удивления, потому что само правило было выведено на основе подобных экспериментов.

Диапазоны, разбиватели и производительность кеша данных

Диапазоны и разбиватели могут повысить производительность кеша данных, разрешив кеш-независимые алгоритмы или алгоритмы с привязкой к кешу. Кеш-независимые алгоритмы полезны, когда набор данных слишком велик для размещения в кешах данных, но можно обратить себе на пользу повторное использование данных в алгоритме, если он организован по принципу «разделяй и властвуй». Наоборот, привязка к кешу полезна, когда набор данных полностью помещается в кеше. Привязка к кешу используется, чтобы повторная обработка одних и тех же частей диапазона планировалась на тех же процессорах, что и раньше, а это значит, что данные, помещающиеся в кеш, можно будет снова получить из того же кеша.

Кеш-независимые алгоритмы

Кеш-независимые алгоритмы достигают хорошего (или даже оптимального) использования кешей данных, ничего не зная о параметрах аппаратного кеша. Концепция похожа на замощение или разбиение цикла на блоки, но не требует задания точного размера плитки или блока. Кеш-независимые алгоритмы часто рекурсивно разбивают проблему на все меньшие и меньшие части. В какой-то момент эти мелкие подпроблемы начинают помещаться в машинный кеш. Рекурсивное разбиение может либо продолжаться до достижения наименьшего возможного размера, либо ради эффективности его можно прекратить в какой-то точке отсечения, но эта точка отсечения ***не*** связана с размером кеша. Как правило, при этом размер порции данных оказывается существенно меньше любого разумного размера кеша.

Поскольку про *кеш-независимые алгоритмы* (cache-oblivious) вовсе нельзя сказать, что производительность кеша им безразлична, мы встречали много других названий, например *нейтральные относительно кеша* (cache agnostic), поскольку такие алгоритмы оптимизируются для любого встретившегося им кеша, а также *параноидальные относительно кеша* (cache paranoid), поскольку в них неявно предполагается наличие бесконечного количества уровней кеша. Но термин «cache oblivious» уже устоялся в литературе.

Ниже мы возьмем транспонирование матрицы в качестве примера алгоритма, который может выиграть от реализации, не учитывающей кеш. На рис. 16.6 приведена последовательная реализация, в которой кеш учитывается.

```
void fig_16_6(int N, double *a, double *b) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      b[j*N+i] = a[i*N+j];
    }
  }
}
```

Рис. 16.6 ❖ Последовательная реализация транспонирования матрицы

Для простоты предположим, что в строке кеша помещается четыре элемента. На рис. 16.7 показаны строки кеша, к которым производятся обращения в процессе транспонирования первых двух строк матрицы `a` размера N×N. Если кеш достаточно большой, то в нем могут остаться все строки, хранящие элементы `b`, к которым были обращения во время транспонирования первой строки `a`, и тогда не нужно будет перезагружать их во время транспонирования второй строки `a`. Но если кеш недостаточно велик, то эти строки придется перезагрузить, что приведет к непопаданию в кеш при каждом доступе к матрице `b`. На рисунке показан массив 16×16, но представьте себе, что он гораздо больше.

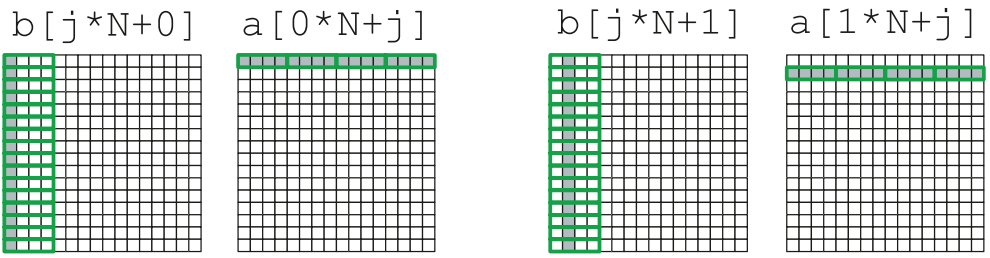


Рис. 16.7 ❖ Строки кеша, к которым производились обращения при транспонировании первых двух строк матрицы `a`. Для простоты показано четыре элемента в каждой строке кеша

Реализация этого кеш-независимого алгоритма сокращает количество обращений к данным между повторными использованиями одной и той же строки кеша или элемента данных. Как показано на рис. 16.8, если сосредоточить все усилия на транспонировании маленького блока матрицы `a` и только потом переходить к другим ее блокам, то можно будет уменьшить количество строк кеша, содержащих элементы `b`, которые необходимо оставить в кеше, чтобы получить прирост производительности благодаря повторному использованию строк кеша.

Последовательная кеш-независимая реализация транспонирования матрицы показана на рис. 16.9. Мы рекурсивно разбиваем проблему по осям `i` и `j` и используем последовательный цикл for, когда размер диапазона оказывается ниже порогового.

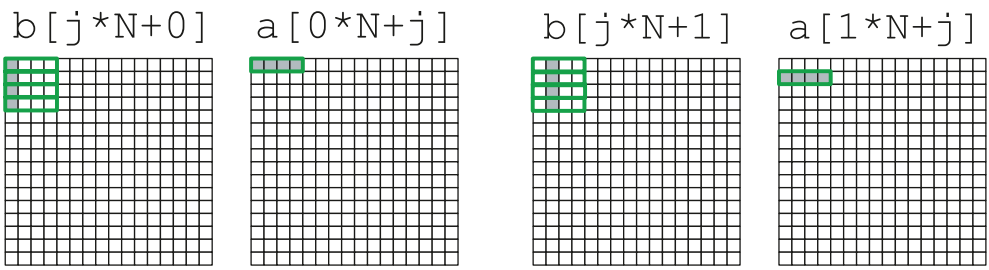


Рис. 16.8 ❖ Поблочное транспонирование сокращает количество строк кеша, которые нужно оставить, чтобы получить выигрыш от повторного использования

```
void obliviousTranspose(int N, int ib, int ie, int jb, int je,
                        double *a, double *b, int gs) {
  int ilen = ie-ib;
  int jlen = je-jb;
  if (ilen > gs ||  jlen > gs) {
     if ( ilen > jlen ) {
       int imid = (ib+ie)/2;
       obliviousTranspose(N, ib, imid, jb, je, a, b, gs);
       obliviousTranspose(N, imid, ie, jb, je, a, b, gs);
     } else {
       int jmid = (jb+je)/2;
       obliviousTranspose(N, ib, ie, jb, jmid, a, b, gs);
       obliviousTranspose(N, ib, ie, jmid, je, a, b, gs);
     }
  } else {
    for (int i = ib; i < ie; ++i) {
      for (int j = jb; j < je; ++j) {
        b[j*N+i] = a[i*N+j];
      }
    }
  }
}
```

Рис. 16.9 ❖ Последовательная кеш-независимая реализация транспонирования матрицы

Поскольку в этой реализации разбиения в направлениях `i` и `j` чередуются, обход матрицы в процессе транспонирования происходит, как показано на рис. 16.10: сначала блок 1, потом 2, 3 и т. д. Если `gs` равно 4 и размер строки кеша равен 4, то мы получаем повторное использование внутри каждого блока, показанного на рис. 16.8. Но если в строке кеша помещается не 4, а 8 элементов (что гораздо вероятнее в реальных системах), то повторное использование будет иметь место не только внутри мельчайших блоков, но и между блоками. Например, если в кеше данных можно оставить все строки, загруженные при обработке блоков 1 и 2, то они будут повторно использованы и при транспонировании блоков 3 и 4.

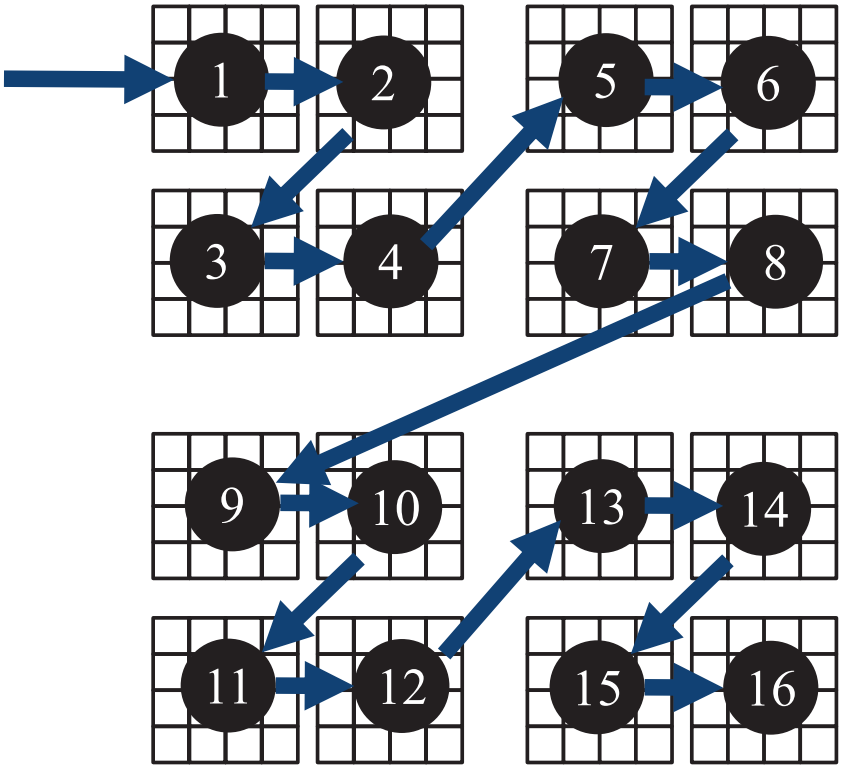


Рис. 16.10 ❖ Порядок обхода,
при котором сначала транспонируются подблоки,
а затем производится переход к другим блокам

В этом и состоит истинная мощь кеш-независимых алгоритмов – нам не нужно точно знать размеры уровней в иерархии памяти. По мере того как подпроблемы становятся меньше, они помещаются во все меньшие компоненты иерархии памяти, и при этом улучшается повторное использование на каждом уровне.

Циклические TBB-алгоритмы и планировщик TBB проектировались с намерением специально поддержать кеш-независимые алгоритмы. Поэтому мы можем быстро написать параллельную кеш-независимую реализацию транспонирования матрицы, применив `parallel_for`, `blocked_range2d` и `simple_partitioner`, как показано на рис. 16.11. Мы воспользовались классом `blocked_range2d`, потому что хотим, чтобы пространство итераций разбивалось на двумерные блоки. А `simple_partitioner` выбрали, поскольку выигрыш от повторного использования можно получить, только если при разбиении на блоки мы доходим до размера, меньшего, чем размер кеша; другие типы разбивателей стараются оптимально сбалансировать нагрузку и потому могут выбрать диапазон большего размера, если его достаточно для балансировки.

На рис. 16.12 показано, что при том способе, которым `parallel_for` рекурсивно разбивает диапазоны, создаются такие же блоки, какие нужны в нашей кеш-независимой реализации. Принципы планировщика – работа в глубину

и заимствование в ширину – означают также, что блоки будут обрабатываться примерно в таком порядке, как на рис. 16.10.

```
double fig_16_11(int N, double *a, double *b, int gs) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::parallel_for(
    tbb::blocked_range2d<int,int>{0, N, gs, 0, N, gs},
    [N, a, b](const tbb::blocked_range2d<int,int> &r) {
      int ie = r.rows().end();
      int je = r.cols().end();
      for (int i = r.rows().begin(); i < ie; ++i) {
        for (int j = r.cols().begin(); j < je; ++j) {
          b[j*N+i] = a[i*N+j];
        }
      }
    }, simple_partitioner()
  );
  tbb::tick_count t1 = tbb::tick_count::now();
  return (t1-t0).seconds();
}
```

Рис. 16.11 ❖ Параллельная кеш-независимая реализация транспонирования матрицы с использованием классов `simple_partitioner` и `blocked_range2d`

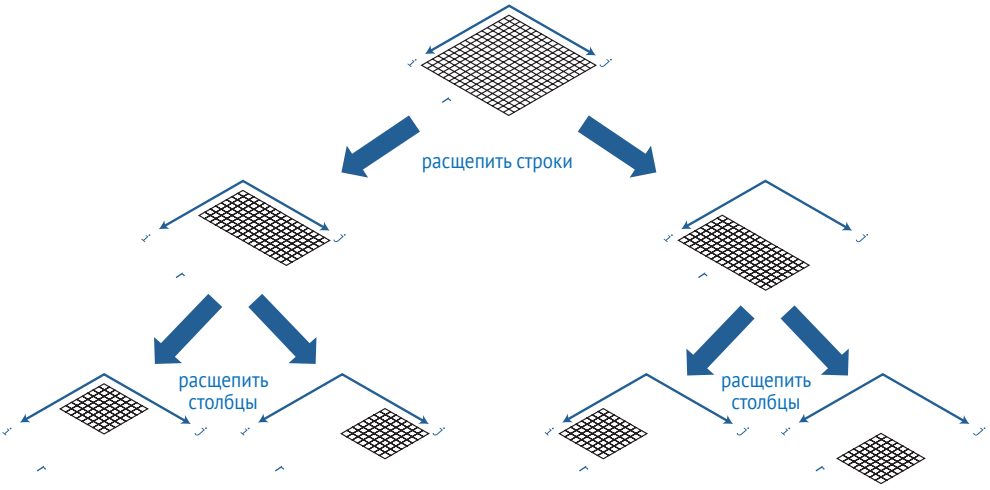


Рис. 16.12 ❖ Рекурсивное разбиение `blocked2d_range` дает те же блоки, что нужны нам в параллельной кеш-независимой реализации

На рис. 16.13 представлены данные о производительности последовательной кеш-независимой реализации (рис.16.9), реализации с использованием одномерного диапазона `blocked_range` и реализации с использованием `blocked_range2d` (рис. 16.11). Мы реализовали параллельные версии, так чтобы можно было легко изменить степень детализации и разбиватель. Код всех версий имеется в файле `fig_16_11.cpp`.

На рис. 16.13 показано ускорение, достигаемое нашей реализацией для матрицы размера 8192×8192 по сравнению с последовательной реализацией на рис. 16.6.

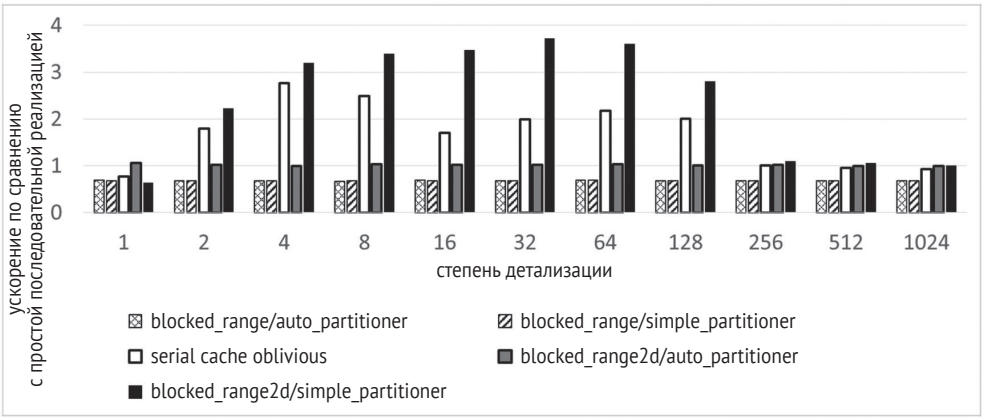


Рис. 16.13 ❖ Ускорение на тестовой машине для N = 8192 при различных значениях степени детализации и разбивателях

Транспонирование матрицы ограничено скоростью чтения и записи данных, никаких вычислений тут нет. По рис. 16.13 видно, что параллельные реализации с одномерным диапазоном `blocked_range` ведут себя хуже, чем простая последовательная реализация при любой степени детализации. Последовательная реализация уже ограничена пропускной способностью памяти – добавление потоков только усиливает нагрузку на и без того нагруженную подсистему памяти, ничего не давая взамен.

В последовательном кеш-независимом алгоритме операции доступа к памяти переупорядочены, в результате чего количество непопаданий в кеш уменьшилось. Производительность по сравнению с простой версией значительно возросла. Если в параллельной реализации используется класс `blocked_range2d`, то мы получаем аналогичные двумерные разбиения. Но, как видно на рис. 16.13, лишь при использовании `simple_partitioner` мы получаем настоящее поведение кеш-независимого алгоритма. На самом деле наш параллельный кеш-независимый алгоритм с применением `blocked_range2d` и `simple_partitioner` уменьшает нагрузку на подсистему памяти до такой степени, что теперь работа в несколько потоков может повысить производительность по сравнению с последовательной кеш-независимой реализацией!

Не у всех проблем есть кеш-независимое решение, но у многих часто встречающихся оно имеется. Имеет смысл потратить время на исследование, чтобы понять, возможно ли кеш-независимое решение и стоит ли оно свеч. Если да, то типы блочных диапазонов и разбиватель `simple_partitioner` позволят очень легко реализовать его с помощью TBB-алгоритмов.

Привязка к кешу

Кеш-независимые алгоритмы улучшают производительность кеша за счет того, что разбивают проблемы, в которых данные обладают локальностью, но

не помещаются в кеш, на меньшие проблемы, которые в кеш уже помещаются. С другой стороны, привязка к кешу означает повторную обработку диапазонов, которые уже находятся в кеше. Поскольку данные размещены в кеше, то назначение диапазонов тем процессорам, которые их уже обрабатывали раньше, позволяет быстрее обращаться к кешированным данным. Для задействования привязки к кешу в циклических TBB-алгоритмах можно использовать разбиватели `affinity_partitioner` или `static_partitioner`. На рис. 16.14 приведен код простого микротеста производительности, который прибавляет значение к каждому элементу одномерного массива. Функция получает ссылку на разбиватель; передавать его по ссылке необходимо, чтобы можно было записывать историю в объект `affinity_partitioner`.

```
template <typename Partitioner>
double fig_16_14(double v, int N, double *a, Partitioner &p) {
  tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
    [v, a](const tbb::blocked_range<int> &r) {
      int ie = r.end();
      for (int i = r.begin(); i < ie; ++i) {
        a[i] += v;
      }
    }, p
  );
}
```

Рис. 16.14 ❖ Функция, в которой алгоритм `parallel_for` используется для прибавления значения ко всем элементам одномерного массива

Чтобы оценить влияние привязки к кешу, мы можем выполнить эту функцию несколько раз с одним и тем же значением `N` и массивом `a`. При использовании `auto_partitioner` планировщик расписывает поддиапазоны потокам поразному при каждом вызове. Даже если массив целиком помещается в кеши процессора, один и тот же участок `a` может не попасть прежнему процессору при следующем выполнении.

```
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, tbb::auto_partitioner{});
}
```

Если же используется `affinity_partitioner`, то библиотека может сохранить результаты планирования задач и с помощью указаний о привязке воспроизвести их при следующем выполнении (об указаниях см. главу 13). Поскольку история хранится в самом разбивателе, мы должны передавать один и тот же объект разбивателя при всех выполнениях, а не создавать временные объекты, как в случае `auto_partitioner`:

```
tbb::affinity_partitioner aff_p;
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, aff_p);
}
```

Наконец, для создания привязки к кешу можно использовать также static_partitioner. Поскольку в этом случае планирование детерминировано, необязательно передавать один и тот же объект разбивателя при каждом выполнении:

```
for (int i = 0; i < M; ++i) {
  fig_16_14(v[i], N, a, tbb::static_partitioner{});
}
```

Мы воспользовались этим микротестом производительности на своей тестовой машине, задав N = 100 000 и M = 10 000. Массив чисел типа double будет иметь размер 100 000 × 8 = 800 K. На нашей машине имелось четыре L2-кеша размером 256 K каждый, по одному на ядро. При использовании affinity_partitioner тест завершался в 1,4 раза быстрее, чем при применении auto_partitioner, а при использовании static_partitioner – в 2,4 раза быстрее! Поскольку данные помещались в агрегированный L2-кеш (4 × 256 K = 1 МБ), воспроизведение одного и того же способа планирования сильно повлияло на время выполнения. В следующем разделе мы обсудим, почему static_partitioner превзошел affinity_partitioner в этом случае и почему это не должно вызывать ни удивления, ни восторга. Если увеличить N до 1 000 000 элементов, то разницы во времени выполнения уже не будет, поскольку массив не помещается в кеши тестовой системы. В таком случае следует переделать алгоритм, реализовав замощение или разбиение на блоки, чтобы воспользоваться локальностью кешей.

Использование static_partitioner

Разбиватель static_partitioner имеет самые низкие накладные расходы и быстро обеспечивает равномерное распределение блочного диапазона между потоками на арене. Поскольку разбиение детерминированное, это может также улучшить поведение кеша в случае, когда цикл или несколько циклов повторно выполняются для одного и того же диапазона. В предыдущем разделе мы видели, что этот разбиватель значительно превосходит affinity_partitioner в нашем микротесте. Однако поскольку он создает лишь столько порций, сколько необходимо для обеспечения работой каждого потока на арене, отсутствует возможность заимствования работ с целью динамической балансировки нагрузки. По существу, static_partitioner отключает принятый в TBB подход к планированию на основе заимствования.

Но для включения static_partitioner в TBB есть основательные причины. С увеличением количества ядер случайное заимствование работ становится все дороже, особенно при переходе от последовательной части приложения к параллельной. Когда мастер-поток впервые запускает новую задачу на арене, все рабочие потоки просыпаются и *несметной ордой* устремляются на поиск работы. Хуже того, они не знают, где искать, и начинают случайно тыкаться не только в двустороннюю очередь мастер-потока, но и в локальные очереди друг друга. В конечном итоге какой-то рабочий поток найдет работу у мастера и разобьет ее на части, а другой рабочий поток рано или поздно найдет одну из частей и разобьет ее. И так далее. Спустя некоторое время буря успокоится, все потоки найдут себе работу и будут радостно трудиться, выбирая задачи из собственных локальных очередей.

Но если мы уже знаем, что рабочая нагрузка хорошо сбалансирована, что система не перегружена и что все ядра одинаково мощные, нужны ли нам все эти накладные расходы на заимствование работ, когда всего-то и надо обеспечить равномерное распределение между рабочими потоками? Не нужны – и при использовании `static_partitioner` их и не будет! Он проектировался специально для этого случая. Он равномерно распределяет диапазон между рабочими потоками, так что им вообще не нужно прибегать к заимствованию задач. В ситуации, когда `static_partitioner` применим, он дает самый эффективный способ разбиения цикла.

Но умерьте восторги! Если рабочая нагрузка неравномерна или какое-то ядро перегружено дополнительными потоками, то применение `static_partitioner` может погубить производительность. Так, на рис. 16.15 представлена та же конфигурация микротеста, что на рис. 16.5(c), – мы пользовались ей, чтобы оценить влияние степени детализации на производительность. Но на рис. 16.15 показано, что происходит, если добавить всего один поток, работающий на одном из ядер. Для всех разбивателей, кроме `static_partitioner`, влияние этого дополнительного потока очень мало. Но `static_partitioner` предполагает, что возможности всех ядер одинаковы, и распределяет работу между ними равномерно. В результате перегруженное ядро становится узким местом и от ускорения ничего не остается.

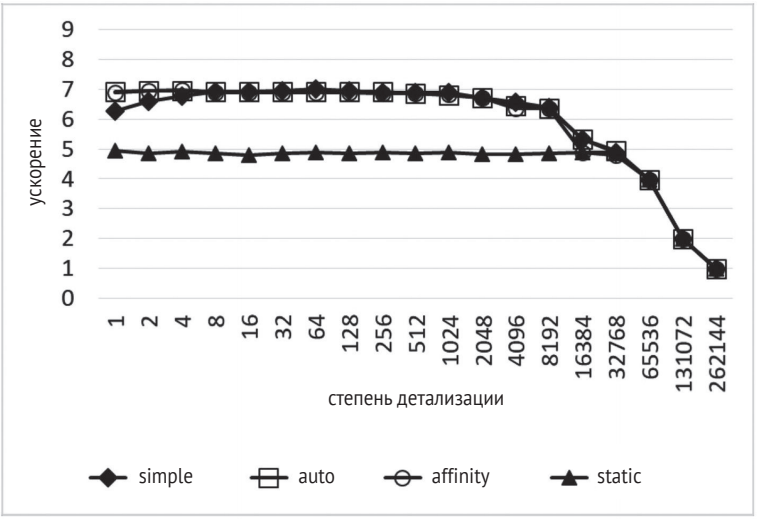


Рис. 16.15 ❖ Ускорение для различных типов разбивателей и увеличивающихся значений степени детализации в случае, когда дополнительный поток крутится в цикле в фоновом режиме. Время выполнения одной итерации равно 1 мкс

На рис. 16.16 показан цикл, в котором объем работы на каждой итерации увеличивается. Если используется `static_partitioner`, то у потока, получившего набор итераций с наименьшими номерами, будет гораздо больше работы, чем у невезунчика, которому достался последний набор итераций.

```
template <typename Partitioner>
double fig_16_16(int N, Partitioner &p) {
  tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
    [](const tbb::blocked_range<int> &r) {
      int ie = r.end();
      for (int i = r.begin(); i < ie; ++i) {
        spinForMicroseconds(i);
      }
    }, p
  );
}
```

Рис. 16.16 ❖ Цикл, в котором объем работы на каждой итерации увеличивается

Если мы прогоним цикл на рис. 16.16 десять раз с каждым типом разбивателя и N = 1000, то получим следующие результаты:

```
auto_partitioner = 0.629974 seconds
affinity_partitioner = 0.630518 seconds
static_partitioner = 1.18314 seconds
```

Разбиватели `auto_partitioner` и `affinity_partitioner` могут перераспределить нагрузку между потоками, добившись лучшего баланса, тогда как `static_partitioner` так и будет использовать распределение, казавшееся в начале равномерным, но потом ставшее несправедливым.

Поэтому `static_partitioner` полезен разве что в приложениях высокопроизводительных вычислений (High Performance Computing – HPC). Такие приложения работают в системах с большим количеством ядер, зачастую в пакетном режиме, когда в каждый момент времени выполняется только одно приложение. Если рабочая нагрузка не нуждается ***ни в какой*** динамической балансировке, то `static_partitioner` почти всегда будет превосходить другие разбиватели по производительности. К сожалению, такие хорошо сбалансированные нагрузки и однопользовательские пакетные системы – скорее исключение, чем правило.

Ограничение планировщика ради детерминированности

В главе 2 мы обсуждали связь между ассоциативностью и типами с плавающей точкой. Мы отметили, что любая реализация чисел с плавающей точкой приближенная, поэтому распараллеливание может приводить к различным результатам, если зависит от таких свойств, как ассоциативность и коммутативность. Эти результаты необязательно неправильные, просто они разные. Но, по крайней мере для редукции, TBB предлагает алгоритм `parallel_deterministic_reduce` на случай, если мы хотим получать одинаковые результаты при каждом выполнении алгоритма на одной и той же машине с одними и теми же данными.

Как легко догадаться, `parallel_deterministic_reduce` принимает только `simple_partitioner` или `static_partitioner`, поскольку для того и другого разбивателя количество диапазонов детерминировано. Кроме того, алгоритм `parallel_de-`

terministic_reduce всегда выполняет один и тот же набор операций расщепления и соединения на данной машине вне зависимости от того, сколько потоков динамически участвует в выполнении, и от того, как задачи отображены на потоки, – для алгоритма parallel_reduce это необязательно. В результате parallel_deterministic_reduce всегда возвращает один и тот же результат при выполнении на одной и той же машине – но при этом утрачивает часть гибкости.

На рис. 16.17 показано ускорение в примере вычисления числа π из главы 2 при реализации с помощью parallel_reduce (r-auto, r-simple и r-static) и parallel_deterministic_reduce (d-simple и d-static). Максимальное ускорение в обоих случаях похоже, однако auto_partitioner performs показывает очень хорошие результаты для parallel_reduce, тогда как для parallel_deterministic_reduce его использование попросту невозможно. При необходимости мы можем реализовать детерминированную версию нашего теста производительности, но тогда придется иметь дело с осложнениями при выборе подходящей степени детализации.

Хотя у parallel_deterministic_reduce имеются кое-какие дополнительные накладные расходы, поскольку он обязан выполнить все расщепления и соединения, в типичном случае эти расходы малы. Более серьезное ограничение заключается в том, что мы не можем воспользоваться разбивателями, которые автоматически находят размер порции.

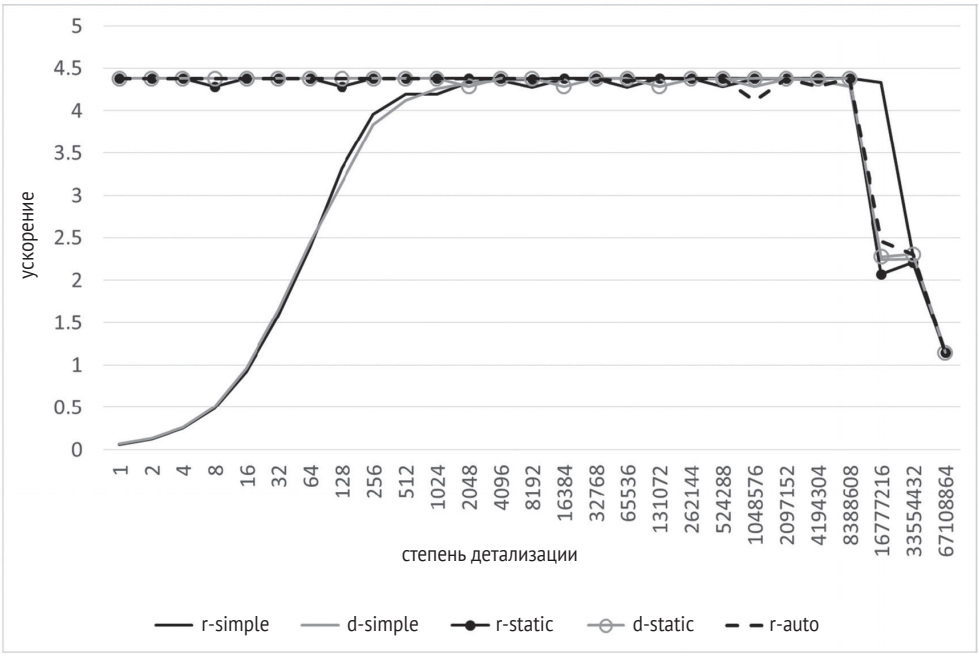


Рис. 16.17 ❖ Ускорение в примере вычисления числа π из главы 2 при использовании parallel_reduce с auto_partitioner (r-auto), simple_partitioner (r-simple) и static_partitioner (r-static), а также при использовании parallel_deterministic_reduce с simple_partitioner (d-simple) и static_partitioner (d-static). Результаты приведены для различных размеров ядер от 1 до N

Настройка конвейеров в TBB: количество фильтров, режимы и маркеры

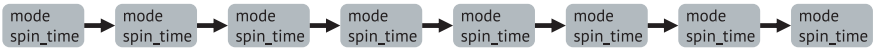
Как и в случае циклических алгоритмов, производительность конвейеров TBB зависит от зернистости, локальности и располагаемого параллелизма. Но, в отличие от циклических алгоритмов, конвейеры не поддерживают диапазонов и разбивателей. Вместо них для настройки используются такие параметры, как количество фильтров, режимы выполнения фильтров и количество маркеров, передаваемых конвейеру во время выполнения.

Конвейерные фильтры запускаются как задачи и планируются библиотекой TBB, следовательно, мы хотим, чтобы тела фильтров, как и поддиапазоны, создаваемые циклическими алгоритмами, выполнялись достаточно долго, чтобы сгладить накладные расходы, но в то же время хочется задействовать как можно больше параллелизма. Компромисс достигается благодаря способу разбиения работы на фильтры. Кроме того, время выполнения фильтров должно быть примерно одинаково, поскольку самый медленный последовательный этап станет бутылочным горлышком.

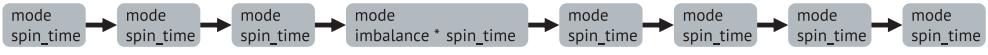
Как было описано в главе 2, при создании конвейерных фильтров задается также режим выполнения: `serial_in_order`, `serial_out_of_order` или `parallel`. В режиме `serial_in_order` фильтр может обрабатывать не более одного элемента в каждый момент времени, причем в том же порядке, в каком их сгенерировал самый первый фильтр. В режиме `serial_out_of_order` элементы могут выполняться в любом порядке. А в режиме `parallel` фильтр может обрабатывать несколько элементов параллельно. Ниже в этом разделе мы рассмотрим, как различные режимы влияют на производительность.

При запуске конвейера необходимо задать аргумент `max_number_of_live_tokens`, ограничивающий количество элементов, которым разрешено находиться в конвейере в каждый момент времени.

На рис. 16.18 показана структура микротестов производительности, которыми мы будем пользоваться для исследования этих параметров. Оба конвейера на рисунке состоят из восьми фильтров, но в экспериментах их число будет меняться. Фильтры в верхнем конвейере работают в одном и том же режиме, и у всех них одинаковое значение `spin_time`, т. е. мы имеем очень хорошо сбалансированный конвейер. В нижнем конвейере один фильтр работает в течение времени `imbalance * spin_time` – мы будем изменять коэффициент `imbalance`, чтобы изучить влияние дисбаланса на ускорение.



(a) Сбалансированный конвейер с 8 фильтрами



(b) Несбалансированный конвейер с 8 фильтрами

Рис. 16.18 ❖ Микротесты производительности со сбалансированным и несбалансированным конвейерами

Сбалансированный конвейер

Сначала рассмотрим, насколько хорошо наше эвристическое правило для размера задач применимо к конвейерам. Верно ли, что одной микросекунды достаточно телу фильтра для сглаживания накладных расходов? На рис. 16.19 показано ускорение нашего сбалансированного конвейера, когда подается 8000 элементов, а количество элементов в конвейере равно всего лишь 1. Приведены результаты для разного времени выполнения фильтра. Поскольку в конвейере одновременно может находиться только один элемент, его выполнение, по сути дела, сериализовано (пусть даже установлен режим выполнения `parallel`).

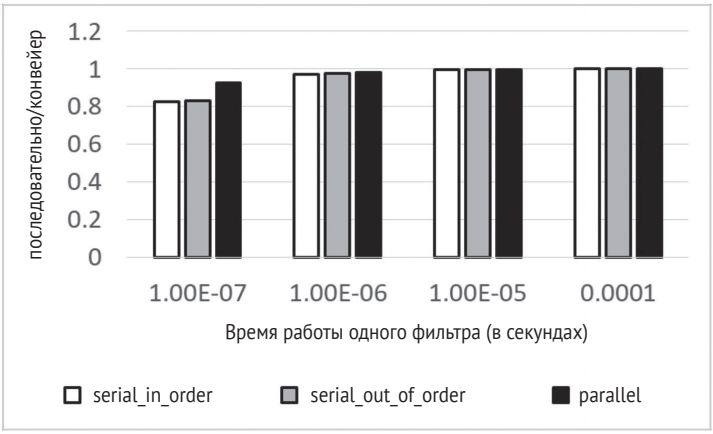


Рис. 16.19 ❖ Накладные расходы в различных режимах выполнения фильтра, когда конвейер с 8 фильтрами сбалансирован, максимальное количество маркеров равно 1 и на вход подается 8000 элементов

Сравнение с истинно последовательным выполнением, когда выполняется нужное количество итераций цикла for, показывает, как влияют накладные расходы на управление конвейером. Мы видим, что когда `spin_time` близко к 1 мкс, накладные расходы малы, так что время параллельного и последовательного выполнения почти одинаково. Похоже, наше эвристическое правило применимо и к конвейерам TBB!

Теперь посмотрим, как влияет на производительность количество фильтров. В последовательном конвейере параллелизм достигается только за счет совмещения разных фильтров во времени. В конвейере с параллельными фильтрами в параллелизм вносит вклад также одновременная обработка нескольких элементов одним фильтром. Наша платформа поддерживает восемь потоков, поэтому можно ожидать 8-кратного ускорения от распараллеливания.

На рис. 16.20 показано ускорение сбалансированного конвейера, когда количество маркеров равно 8. В обоих последовательных режимах ускорение тем больше, чем больше количество фильтров. Это важно запомнить, потому что ускорение последовательного конвейера не масштабируется вместе с размером набора данных, как в случае циклических TBB-алгоритмов. Но сбаланси-

рованный конвейер, состоящий только из параллельных фильтров, достигает ускорения 8, даже когда имеется всего один фильтр. Это объясняется тем, что 8000 поданных на вход элементов могут обрабатываться этим единственным фильтром параллельно – не существует никакого последовательного фильтра, который мог бы стать узким местом.

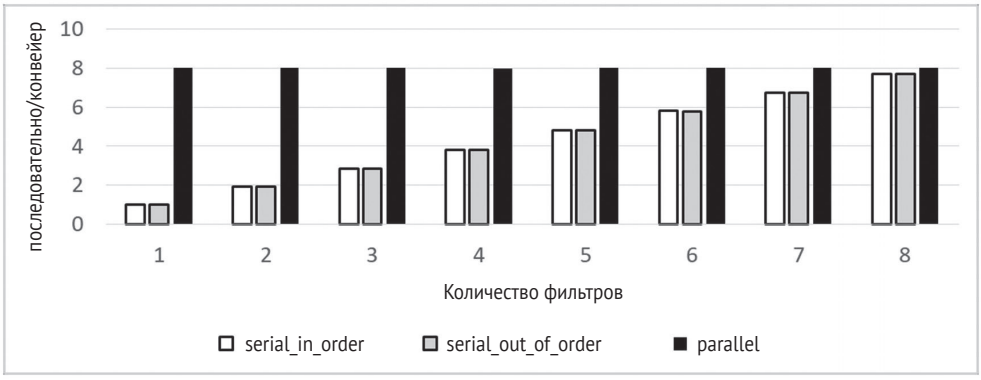


Рис. 16.20 ❖ Ускорение, достигаемое в различных режимах выполнения, когда конвейер с 8 маркерами сбалансирован, на вход подается 8000 элементов, а количество фильтров постепенно увеличивается. Время работы каждого фильтра равно 100 мкс

На рис. 16.21 мы видим ускорение нашего сбалансированного конвейера, когда фильтров восемь, но количество маркеров изменяется. Поскольку на нашей платформе восемь потоков, при количестве маркеров, меньшем 8, элементов будет недостаточно, чтобы занять все потоки. Но когда в конвейере находится по крайней мере восемь элементов, все потоки заняты делом. Дальнейшее увеличение количества маркеров почти не влияет на производительность.

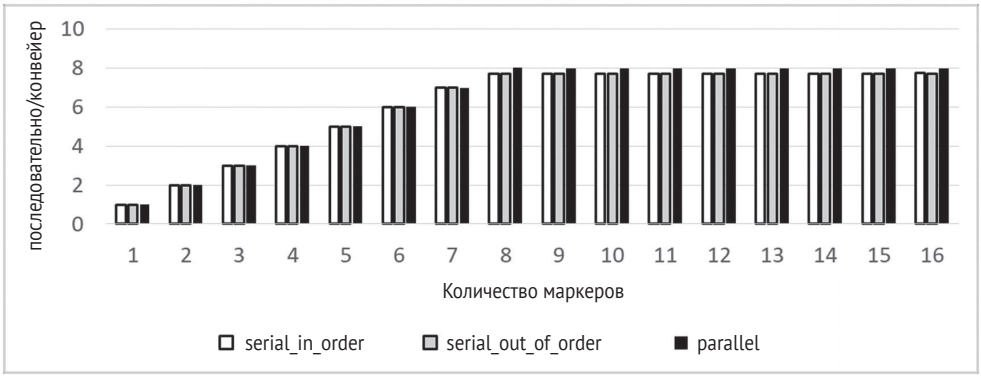


Рис. 16.21 ❖ Ускорение, достигаемое в различных режимах выполнения, когда конвейер с 8 фильтрами сбалансирован, на вход подается 8000 элементов, а количество маркеров постепенно увеличивается. Время работы каждого фильтра равно 100 мкс

Несбалансированный конвейер

Теперь рассмотрим производительность несбалансированного конвейера на рис. 16.18. В этом микротесте все фильтры работают `spin_time` секунд, кроме одного, работающего `spin_time * imbalance` секунд. Таким образом, время, необходимое для обработки N элементов, проходящих по нашему несбалансированному конвейеру с 8 фильтрами, равно

$$T_1 = N*(7*spin\_time + spin\_time*imbalance).$$

В стационарном состоянии последовательный конвейер ограничен самым медленным последовательным этапом. Длина критического пути того же конвейера, когда несбалансированный фильтр выполняется в последовательном режиме, равна

$$T_\infty = N*\max(spin\_time, spin\_time*imbalance).$$

На рис. 16.22 показаны результаты для несбалансированного конвейера, исполняемого на нашей тестовой платформе с разными коэффициентами `imbalance`. Мы включили также теоретически максимальное ускорение, помеченное как «работа/критический путь» и вычисляемое по формуле

$$Speedup_{\max} = \frac{7*spin\_time + spin\_time*imbalance}{\max(spin\ time, spin\_time*imbalance)}.$$

Как и следовало ожидать, по рис. 16.22 видно, что последовательные конвейеры ограничены самыми медленными фильтрами – и результаты измерений близки к предсказанной величине работа/критический путь.

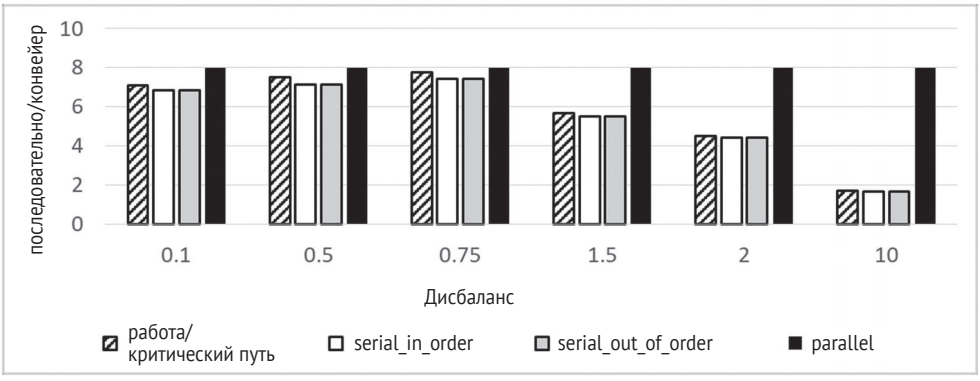


Рис. 16.22 ❖ Ускорение, достигаемое в различных режимах выполнения, когда конвейер с 8 фильтрами не сбалансирован и на вход подается 8000 элементов при разных коэффициентах дисбаланса. Семь фильтров работают 100 мкс, а один – `imbalance * 100` мкс

С другой стороны, показано, что параллельный конвейер на рис. 16.22 не ограничен самым медленным этапом, потому что планировщик может совместить выполнение самого медленного фильтра с другими вызовами того

же фильтра. Может возникнуть вопрос, не станет ли лучше, если количество маркеров будет больше 8. Нет, в данном случае не станет. В нашей тестовой системе всего 8 потоков, поэтому невозможно совместить более восьми экземпляров самого медленного фильтра. Бывают случаи, когда временный дисбаланс нагрузки можно выправить, взяв больше маркеров, чем имеется потоков, но в нашем микротесте, где дисбаланс задается постоянным коэффициентом, мы на самом деле ограничены длиной критического пути и количеством потоков, и никакое количество дополнительных маркеров этого не изменит.

Однако существуют алгоритмы, в которых недостаточное число маркеров не дает планировщику TBB с заимствованием работ автоматически сбалансировать нагрузку. Так бывает, когда этапы конвейера плохо сбалансированы и имеются последовательные этапы, которые тормозят конвейер. В работе A. Navarro et al. (см. раздел «Дополнительная информация») продемонстрировано, что алгоритм `pipeline`, реализованный в TBB, может давать оптимальную производительность, если указать правильное число маркеров. Она построила аналитическую модель на основе теории массового обслуживания, которая позволила найти этот ключевой параметр. Главный вывод этой работы заключается в том, что когда количество маркеров достаточно велико, заимствование работ в TBB моделирует глобальную очередь, способную питать данными все потоки (в теории массового обслуживания известно, что теоретически идеальным случаем является централизованная система с единой глобальной очередью, обслуживающей все ресурсы). Но в реальности одна глобальная очередь, обслуживающая много потоков, порождала бы состязание между ними. Фундаментальное преимущество TBB заключается в том, что в ней реализовано распределенное решение с одной очередью на каждый поток, которое благодаря планировщику с заимствованием работ ведет себя как глобальная очередь. Эта децентрализованная реализация работает так же производительно, как идеальная централизованная система, но без присущего ей бутылочного горлышка.

Конвейеры, локальность данных и привязка к потоку

В сочетании с циклическими алгоритмами мы использовали типы блочных диапазонов, а также разбиватели `affinity_partitioner` и `static_partitioner` для настройки производительности кеша. У функции `parallel_pipeline` и класса `pipeline` нет похожих средств. Но не все потеряно! Встроенный в конвейеры TBB порядок выполнения спроектирован так, что временная локальность данных учитывается без каких-либо специальных усилий.

Когда мастер-поток или рабочий поток завершает выполнение фильтра TBB, он приступает к выполнению следующего фильтра в конвейере, если только это не запрещено ограничениями, налагаемыми режимом выполнения. Например, если фильтр f_0 порождает элемент `i` и его выход передается следующему фильтру f_1, то тот поток, который выполнял f_0, перейдет к выполнению f_1 – если только для этого следующего фильтра не выполняется одно из условий: он работает в режиме `serial_out_of_order` и обрабатывает в данный момент что-то другое или он работает в режиме `serial_in_order` и элемент `i` не является следу-

ющим по порядку. Тогда элемент буферизуется в следующем фильтре, а поток ищет себе другую работу. В противном случае для максимизации локальности поток следует за данными, которые сам только что сгенерировал, и обрабатывает их, выполняя следующий фильтр.

Под капотом обработка одного элемента в фильтре f_0 реализована как задача, исполняемая потоком (ядром). Когда фильтр закончит работу, задача рециклирует себя (о рециклинге задач см. главу 10), чтобы выполнить следующий фильтр f_1. По существу, умирающая задача f_0 воскресает в виде новой задачи f_1, обходя планировщик, – тот же поток, который выполнял f_0, будет выполнять и f_1. С точки зрения локальности данных и производительности, это намного лучше, чем наивная реализация конвейера: фильтр f_0 (обслуживаемый одним или несколькими потоками) ставит элемент в очередь фильтра f_1 (причем f_1 тоже обслуживается одним или несколькими потоками). При такой наивной реализации локальность данных страдает, потому что элемент, обработанный фильтром f_0 на одном ядре, скорее всего, продолжит обрабатываться фильтром f_1 уже на другом ядре. В TBB, если f_0 и f_1 удовлетворяют сформулированным выше условиям, такого не произойдет. В результате конвейер TBB склонен к завершению обработки уже начатых элементов, до того как в начало конвейера будут помещены новые элементы; при таком поведении не только повышается локальность данных, но и потребляется меньше памяти, поскольку уменьшается размер очередей в последовательных фильтрах.

К сожалению, конвейерные фильтры в TBB *не* поддерживают указаний о привязке. Невозможно сказать, что мы хотим, чтобы конкретный фильтр исполнялся конкретным рабочим потоком. Но, как ни странно, существует механизм жесткой привязки, `thread_bound_filter`. Однако его использование требует небезопасного относительно типов, подверженного ошибкам интерфейса `tbb::pipeline`, который мы опишем в следующем разделе.

В глубоких водах

В этом разделе рассматриваются средства, которые редко бывают нужны пользователям TBB, но уж если нужны, то могут оказаться очень полезными. Вы можете пропустить этот раздел и вернуться к нему, если возникнет необходимость создать собственный тип диапазона или включить фильтр типа `thread_bound_filter` в конвейер TBB. Ну а если хотите узнать о TBB как можно больше, читайте дальше!

Создание собственного типа диапазона

Как уже было сказано выше, типы блочных диапазонов охватывают большинство типичных ситуаций. За годы работы с TBB нам встретилось совсем немного случаев, когда имело смысл реализовать собственный тип диапазона. Но при необходимости мы можем сделать это, реализовав классы, моделирующие концепцию диапазона, представленную на рис. 16.1.

Чтобы продемонстрировать пример полезного, но нетипичного типа диапазона, вернемся к алгоритму быстрой сортировки, показанному на рис. 16.23.

```
struct SortData {
  int id;
  double value;
  SortData(int i, double v) : id(i), value(v) {}
  bool operator<(const SortData &other) const {
    return value < other.value;
  }
  bool operator==(const SortData &other) const {
    return value == other.value;
  }
};

using QSVector = std::vector<SortData>;

QSVector::iterator doShuffle(QSVector::iterator b,
                             QSVector::iterator e) {
  QSVector::iterator i = b, j = e-1;
  double pivot_value = b->value;
  while (i != j) {
    while (i != j && pivot_value < j->value) --j;
    while (i != j && i->value <= pivot_value) ++i;
    std::iter_swap(i, j);
  }
  std::iter_swap(b, i);
  return i;
}

void serialQuicksort(QSVector::iterator b,
                     QSVector::iterator e) {
  if (b >= e) return;
  QSVector::iterator i = doShuffle(b,e);
  serialQuicksort(b, i);
  serialQuicksort(i+1, e);
}
```

Рис. 16.23 ❖ Последовательная реализация быстрой сортировки

Теперь для распараллеливания быстрой сортировки мы вообще не будем прибегать к явной рекурсии, а воспользуемся алгоритмом `parallel_for` и собственным типом `ShuffleRange`. Наша реализация `pforQuicksort` показана на рис. 16.24.

На рис. 16.24 мы видим, что лямбда-выражение в теле `parallel_for` является базовым случаем, в котором вызывается `serialQuicksort`. Мы также используем `simple_partitioner`, а это означает, что наш диапазон будет рекурсивно разбиваться, пока метод `is_divisible` не вернет `false`. А вся магия перестановки будет вершиться в классе `ShuffleRange` в процессе разбиения на поддиапазоны. Определение класса `ShuffleRange` также приведено на рис. 16.24.

`ShuffleRange` моделирует концепцию диапазона, т. е. в нем определены копирующий конструктор, расщепляющий конструктор, методы `empty` и `is_divisible`, а также переменная-член `is_splittable_in_proportion`, равная `false`. В классе

также хранятся итераторы begin и end, которые ограничивают отрезок массива, и порог отсечения cutoff.

```
class ShuffleRange {
  QSVector::iterator myBegin;
  QSVector::iterator myEnd;

public:
  static const bool is_splittable_in_proportion = false;
  static const int cutoff = 100;

  // конструкторы
  ShuffleRange(const QSVector::iterator b,
               const QSVector::iterator e )
    : myBegin(b), myEnd(e) { }

  ShuffleRange(const ShuffleRange &r)
    : myBegin(r.myBegin), myEnd(r.myEnd) { }

  ShuffleRange(ShuffleRange &r, tbb::split)
      : myBegin(r.myBegin), myEnd(r.myEnd) {
      QSVector::iterator b = r.myBegin;
      QSVector::iterator e = r.myEnd;
      QSVector::iterator i = doShuffle(b,e);
      r.myEnd = i;
      myBegin = i+1;
  }

  bool empty() const { return myBegin >= myEnd; }
  bool is_divisible() const { return myEnd-myBegin >= cutoff;
}
  QSVector::iterator begin() const { return myBegin; }
  QSVector::iterator end() const { return myEnd; }
};
```

```
void pforQuicksort(QSVector::iterator b, QSVector::iterator e)
{
  tbb::parallel_for(ShuffleRange(b, e),
    [](const ShuffleRange &r) {
      serialQuicksort(r.begin(), r.end());
    },
    tbb::simple_partitioner()
  );
}
```

Рис. 16.24 ❖ Реализация параллельной быстрой сортировки с помощью parallel_for и класса ShuffleRange, моделирующего концепцию диапазона

Начнем с метода empty. Диапазон пуст, если его итератор begin указывает на итератор end или на позицию после него.

Порог отсечения определяет, следует ли разбивать диапазон дальше. Напомним, что мы используем разбиватель simple_partitioner, поэтому parallel_for будет разбивать диапазоны, пока is_divisible не вернет false. Следовательно, реализация is_divisible в классе ShuffleRange просто сравнивает с cutoff.

Теперь пора заняться главной частью реализации, расщепляющим конструктором, показанным на рис. 16.24. Он получает ссылку на объект ShuffleRange r, который требуется расщепить, и объект tbb::split, нужный для того, чтобы отличить этот конструктор от копирующего. Тело конструктора представляет собой базовый алгоритм выбора граничного значения и перестановки. Он обновляет исходный диапазон r, так что тот становится левой половиной, а вновь сконструированный объект ShuffleRange делает правой половиной.

Выполнение pforQuicksort на нашей тестовой платформе дает результаты, по производительности очень похожие на реализацию с помощью parallel_invoke в главе 2. Но этот пример демонстрирует исключительную гибкость концепции диапазона. Можно было бы подумать, что операция рекурсивного разбиения диапазона настолько быстрая, что ей можно пренебречь, но в нашей реализации pforQuicksort это не так. Мы полагаемся на то, что для расщепления ShuffleRange нужно выполнить заметный объем работы.

Класс Pipeline и фильтры, привязанные к потоку

Мы уже сказали, что алгоритм tbb::parallel_pipeline не поддерживает указания о привязке. Мы не можем выразить пожелание исполнять данный фильтр в конкретном потоке. Но привязанные к потоку фильтры поддерживаются в более старом, небезопасном относительно типов классе tbb::pipeline! Такие фильтры вообще не обрабатываются рабочими потоками TBB, мы должны явно обрабатывать элементы в них, вызывая функцию process_item или try_process_item.

Обычно thread_bound_filter используется не для улучшения локальности данных, а когда фильтр обязательно нужно выполнить в конкретном потоке – быть может, потому что только этот поток имеет право доступа к ресурсам, необходимым для выполнения действия, реализованного фильтром. Такие ситуации возникают в реальных приложениях, например когда коммуникационная или разгрузочная библиотека требует, чтобы все взаимодействие производилось в определенном потоке.

Рассмотрим искусственный пример, моделирующий такую ситуацию, когда только главный поток имеет доступ к открытому файлу. Чтобы воспользоваться фильтром thread_bound_filter, нам понадобятся небезопасные относительно типов интерфейсы класса tbb::pipeline. Мы не можем создать объект thread_bound_filter при использовании функции tbb::parallel_pipeline. Скоро мы увидим, что использовать thread_bound_filter совместно с parallel_pipeline в любом случае бессмысленно.

В нашем примере создается три фильтра. По большей части они наследуют функциональность от класса tbb::filter, переопределяя только функцию operator():

```
namespace tbb {
    class filter {
    public:
        enum mode {
            parallel = implementation-defined,
            serial_in_order = implementation-defined,
            serial_out_of_order = implementation-defined
        };
        bool is_serial() const;
        bool is_ordered() const;
        virtual void* operator()( void* item ) = 0;
        virtual void finalize( void* item ) {}
        virtual ~filter();
    protected:
        explicit filter( mode );
    };
}
```

Фильтр `SourceFilter`, показанный на рис. 16.25, работает в режиме `serial_in_order`, наследует `tbb::filter` и порождает последовательность чисел. Небезопасные относительно типов интерфейсы класса `tbb::pipeline` требуют, чтобы возвращаемый выход каждого фильтра имел тип `void *`. Значение `NULL` обозначает конец входного потока. Легко видеть, почему новый интерфейс `parallel_pipeline` предпочтительнее в тех случаях, когда он применим.

Второй созданный нами тип фильтра, `MultiplyFilter`, умножает входное значение на 2 и возвращает результат. Он также работает в режиме `serial_in_order filter` и наследует `tbb::filter`.

Наконец, класс `BadWriteFilter` реализует фильтр, который записывает свой выход в файл. Он также наследует `tbb::filter`, как показано на рис. 16.25.

Функция `fig_16_25` сводит эти классы воедино и при этом намеренно вносит ошибку. Она создает трехэтапный конвейер, в котором используются наши классы фильтров и интерфейс `tbb::pipeline`. Сначала создается объект `pipeline`, а затем в него один за другим добавляются фильтры. Для запуска конвейера вызывается функция `pipeline::run(size_t max_number_of_live_tokens)`, которой передается максимальное число маркеров – восемь.

При выполнении этого примера `BadWriteFilter wf` иногда выполняется не в мастер-потоке (как и следовало ожидать), и тогда мы видим сообщение

```
Error!
Done.
```

Пример выглядит искусственно, но напомним, что мы пытаемся смоделировать реальную ситуацию, в которой требуется выполнение в конкретном потоке. Поэтому предположим, что мы не можем сделать `ofstream` доступным всем потокам, а вместо этого обязаны производить запись в главном потоке.

На рис. 16.26 показано, как можно разрешить эту проблему с помощью `thread_bound_filter`. Для этого создадим класс фильтра `ThreadBoundWriteFilter`, наследующий `thread_bound_filter`. На самом деле, кроме базового класса, он ничем не отличается от `BadWriteFilter`.

```
class SourceFilter : public tbb::filter {
public:
  SourceFilter(size_t n);
  void *operator() (void *) override;
private:
  size_t numItems;
};

class MultiplyFilter : public tbb::filter {
public:
  MultiplyFilter();
  void *operator() (void *v) override;
};

thread_local std::ofstream output;

class BadWriteFilter : public tbb::filter {
public:
  BadWriteFilter()
    : tbb::filter(tbb::filter::serial_in_order),
      issued_error(false) { }
  void *operator() (void *v) override {
    if (output.is_open()) {
       output << reinterpret_cast<size_t>(v) << std::endl;
    } else if (!issued_error) {
      std::cerr << "Error!" << std::endl;
      issued_error = true;
    }
  }
private:
  bool issued_error;
};

void fig_16_25() {
  output.open("output.txt", std::ofstream::out);

  SourceFilter sf(100);
  MultiplyFilter mf;
  BadWriteFilter wf;
  tbb::pipeline p;
  p.add_filter(sf);
  p.add_filter(mf);
  p.add_filter(wf);
  p.run(tbb::task_scheduler_init::default_num_threads());
  std::cout << "Done." << std::endl;
}
```

Рис. 16.25 ❖ Пример с ошибкой – программа аварийно завершается, если `BadWriteFilter` пытается записать свой выход в файл из рабочего потока

```
thread_local std::ofstream output;

class ThreadBoundWriteFilter : public tbb::thread_bound_filter {
  bool issued_error;
public:
  ThreadBoundWriteFilter()
    : tbb::thread_bound_filter(tbb::filter::serial_in_order) { }
  void *operator() (void *v) override {
    if (output.is_open()) {
       output << reinterpret_cast<size_t>(v) << std::endl;
    } else if (!issued_error) {
      std::cerr << "Error!" << std::endl;
      issued_error = true;
    }
  }
};

void fig_16_26() {
  output.open("output.txt", std::ofstream::out);

  SourceFilter sf(100);
  MultiplyFilter mf;
  ThreadBoundWriteFilter wf;

  tbb::pipeline p;
  p.add_filter(sf);
  p.add_filter(mf);
  p.add_filter(wf);

  std::thread t([&]() {
    p.run(tbb::task_scheduler_init::default_num_threads());
  });
  while (wf.process_item() !=
         tbb::thread_bound_filter::end_of_stream)
    continue;

  t.join();
  std::cout << "Done." << std::endl;
}
```

Рис. 16.26 ❖ Выход фильтра записывается в файл только из главного потока

Хотя реализации классов похожи, использовать этот фильтр нужно совсем иначе – как показано в функции `fig_16_26`. Теперь конвейер запускается в отдельном потоке. Это необходимо, потому что главный поток должен быть свободен для обслуживания привязанного к потоку фильтра. Мы добавили также цикл while, в котором повторно вызывается функция `process_item` для нашего объекта `ThreadBoundWriteFilter`. Именно здесь выполняется фильтр. Цикл while продолжает работать, пока `process_item` не вернет значение `tbb::thread_bound_filter::end_of_stream`, говорящее, что больше обрабатывать нечего.

Прогон примера на рис. 16.26 показывает, что ошибка исправлена:

```
Done.
```

Резюме

В этой главе мы более основательно рассмотрели средства настройки TBB-алгоритмов. Обсуждение было построено вокруг трех концепций, важных для настройки приложений TBB: зернистость задач, располагаемый параллелизм и локальность данных.

При рассмотрении циклических алгоритмов мы акцентировали внимание на типах блочных диапазонов и разбивателей. Мы выяснили, что в общем случае задача должна работать не менее 1 мкс, чтобы амортизировать накладные расходы на планирование. Эта приблизительная рекомендация справедлива как для циклических алгоритмов типа `parallel_for`, так и для времени работы фильтра в конвейере `parallel_pipeline`.

Мы обсудили, как использовать блочные диапазоны для управления зернистостью и для оптимизации иерархии памяти. Мы воспользовались диапазоном `blocked_range2d` и разбивателем `simple_partitioner` для реализации кеш-независимого транспонирования матрицы. Затем мы показали, как можно использовать `affinity_partitioner` или `static_partitioner`, чтобы воспроизвести планирование диапазонов, так чтобы одни и те же данные каждый раз обрабатывались одними и теми же потоками. Мы показали, что `static_partitioner` – лучший разбиватель для хорошо сбалансированных нагрузок при работе в пакетном режиме, но если нагрузка не сбалансирована или система перегружена, то сказывается его неспособность динамически балансировать нагрузку с помощью заимствования работ. Далее мы еще немного поговорили о детерминированности и описали, каким образом алгоритм `deterministic_parallel_reduce` дает детерминированные результаты, правда, дорогой ценой: нужно либо использовать `simple_partitioner` и тщательно выбирать степень детализации, либо применить `static_partitioner` и пожертвовать динамической балансировкой.

Затем мы обратились к алгоритму `parallel_pipeline` и влиянию на производительность количества фильтров, режима выполнения и количества маркеров. Мы обсудили поведение сбалансированных и несбалансированных конвейеров. Наконец, мы отметили, что конвейеры TBB не предоставляют средств для привязки к потоку, зато обеспечивают временную локальность данных, поскольку потоки следуют за обрабатываемыми элементами по всему конвейеру.

И в заключение мы коснулись более сложных тем: как создать собственный тип диапазона и как использовать фильтры типа `thread_bound_filter`.

Дополнительная информация

О кеш-независимых алгоритмах:

- *Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran.* Cache-Oblivious Algorithms // ACM Trans. Algorithms 8, 1, Article 4 (January 2012), 22 p.

О конвейерном параллелизме:

- *Angeles Navarro et al*. Analytical Modeling of Pipeline Parallelism // ACM-IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'09). 2009.

О проблеме несметной орды:

- https://en.wikipedia.org/wiki/Thundering_herd_problem.

Глава 17

Потоковые графы: дополнительные сведения

В этой главе приводятся важные советы о том, как добиться максимальной производительности от потоковых графов в TBB. Потоковые графы структурированы слабее, чем другие конструкции TBB, и, следовательно, более выразительны, поэтому чтобы получить оптимальную масштабируемую производительность, нужно как следует подумать – и мы расскажем, как настроить потоковый граф, чтобы полнее раскрыть его потенциал.

В главе 3 мы познакомились с классами и функциями в пространстве имен `tbb::flow` и способами их использования для описания простых графов потоков данных и графов зависимостей. В этой главе мы обсудим дополнительные вопросы, возникающие при использовании потоковых графов. Как и в главе 16, разговор будет вращаться в основном вокруг зернистости, эффективного использования памяти и организации достаточного параллелизма. Но поскольку API потоковых графов позволяет выражать параллелизм, не так сильно структурированный, как в параллельных алгоритмах, описанных в главе 16, то мы также дадим некоторые рекомендации о том, что стоит, а чего не стоит делать при обдумывании архитектуры потокового графа.

В разделе «Главное о потоковых графах: что можно, а чего нельзя» приводятся очень конкретные эвристические правила, бесценные при работе с потоковыми графами в TBB.

В заключение мы дадим краткий обзор инструмента Flow Graph Analyzer (FGA), входящего в состав Intel Parallel Studio XE. Он весьма полезен для графического проектирования и анализа потоковых графов. Хотя использовать FGA для работы с потоковыми графами необязательно, визуализация графа в процессе проектирования и анализа может помочь. Эта программа совершенно бесплатна, и мы рекомендуем ее всем, кто намерен серьезно заниматься потоковыми графами в TBB.

Оптимизация зернистости, локальности и степени параллелизма

В этом разделе мы будем заниматься теми же тремя концепциями, которые были в центре нашего внимания в главе 16. Сначала рассмотрим влияние зернистости узлов графа на производительность. Поскольку потоковые гра-

фы применяются в слабее структурированных алгоритмах, при обсуждении зернистости придется учитывать степень параллелизма – требует ли структура большого количества заимствований или порождение задач хорошо распределено между всеми потоками? Кроме того, иногда желательно включать в граф очень маленькие узлы, просто потому что они упрощают дизайн – в таких случаях мы описываем, как использовать узел с облегченной политикой выполнения, чтобы ограничить накладные расходы. Мы также рассмотрим вопрос о локальности данных. В отличие от параллельных TBB-алгоритмов, API потоковых графов не предоставляет таких абстракций, как диапазон или разбиватель; зато он спроектирован так, что локальность обеспечивается естественным образом. Мы обсудим, как потоки следуют за данными, чтобы воспользоваться локальностью. И третий вопрос – организация достаточного параллелизма. Как и в главе 16, за оптимизацию зернистости и локальности зачастую приходится расплачиваться ограниченным параллелизмом, поэтому мы должны пройти по острию этого ножа очень осторожно.

Зернистость узла: какой будет достаточно?

В главе 16 мы обсудили, как диапазоны и разбиватели применяются, для того чтобы задачи, создаваемые обобщенными TBB-алгоритмами, оказались достаточно большими, чтобы амортизировать накладные расходы на планирование, и в то же время достаточно малыми, чтобы независимых работ хватало для масштабируемости. Потоковые графы не поддерживают диапазонов и разбивателей, но от этого важность зернистости задач не уменьшается.

Чтобы понять, применимо ли к узлам графов правило одной микросекунды, сформулированное в главе 16 для тел параллельных алгоритмов, мы напишем несколько простых микротестов производительности, на которых изучим крайние случаи, встречающиеся в потоковых графах. Мы сравним время выполнения четырех функций, варьируя объем работы в узле. Эти функции будем называть **Последовательный цикл**, **ПГ-цикл**, **Мастер-цикл** и **ПГ-цикл на каждый рабочий поток**.

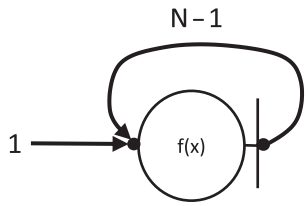
Мы полагаем, что изучение этих примеров (рис. 17.1–17.4) критически важно, т. к. позволяет интуитивно уяснить, что именно отличает хорошо масштабируемый потоковый граф от графа, который не вызывает ничего, кроме разочарования. Сами API, документированные в приложении B, не дают ответа на этот вопрос – мы надеемся, что вы хорошенько изучите приведенные примеры, поскольку это позволит извлечь максимум пользы из потоковых графов (на рис. 17.5 количественно представлены выгоды, которые приносит понимание этих вопросов!).

Последовательный цикл станет для нас точкой отсчета, это цикл for, в котором N раз вызывается функция активного ожидания (рис. 17.1).

На рис. 17.2 представлена функция **ПГ-цикла**. Она строит потоковый граф с одним узлом типа `multifunction_node`, в котором выход соединен с входом ребром. Цикл запускается одним сообщением, после чего узел выполняет активное ожидание и отправляет сообщение обратно на вход. Так повторяется N – 1 раз. Поскольку узел ждет, прежде чем отправить сообщение на вход, этот граф по-прежнему представляет собой последовательный цикл – большая часть ра-

```
double fig_17_1(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int i = 0; i < N; ++i) {
      spinWaitForAtLeast(per_node_time);
    }
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

Рис. 17.1 ❖ Последовательный цикл:
функция для хронометража эталонного последовательного цикла



(a) Диаграмма последовательного потокового графа

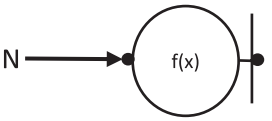
```
double fig_17_2(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n{g, tbb::flow::unlimited,
    [N, per_node_time](int i, node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
      if (i+1 < N) {
        std::get<0>(p).try_put(i+1);
      }
    }
  };
  tbb::flow::make_edge(tbb::flow::output_port<0>(n), n);

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    n.try_put(0);
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

(b) Исходный код реализации

Рис. 17.2 ❖ ПГ-цикл:
функция для хронометража последовательного потокового графа

боты, выполняемой задачами тела, не пересекается во времени. Однако поскольку сообщение отправляется до возврата из тела, существует короткий промежуток времени, в течение которого другой поток может позаимствовать задачу, порождаемую функцией `try_put`. Этот граф можно рассматривать как иллюстрацию базовых накладных расходов на поддержание инфраструктуры потокового графа.

Следующий микротест, функция **Мастер-цикла**, показан на рис. 17.3. Она не создает петлю, а посылает все N сообщений узлу `multifunction_node` непосредственно из мастер-потока в последовательном цикле. Поскольку `multifunction_node` допускает неограниченный параллелизм, а последовательный цикл посылает сообщения очень быстро, то создается очень много параллельных задач. Но так как мастер-поток – единственный поток, вызывающий метод `try_put` узла `n`, то все запущенные задачи попадают в локальную двустороннюю очередь мастер-потока. Рабочие потоки, участвующие в выполнении этого графа, будут вынуждены заимствовать каждую задачу – и сначала им еще нужно будет случайным образом выбрать мастер-поток в качестве жертвы. Этот пример можно использовать для демонстрации поведения потокового графа с доста-



(a) Диаграмма параллельного потокового графа с N начальными сообщениями

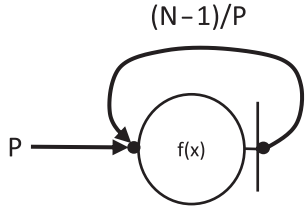
```
double fig_17_3(int num_trials, int N, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n(g, tbb::flow::unlimited,
    [N, per_node_time](int i, node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
    }
  );

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int i = 0; i < N; ++i) {
      n.try_put(0);
    }
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds();
}
```

(b) Исходный код реализации

Рис. 17.3 ❖ Мастер-цикл: функция отправляет сообщения только из мастер-потока, рабочие потоки должны заимствовать каждую задачу

точной степенью параллелизма, который тем не менее вынужден чрезмерно часто заимствовать работу.

Наконец, на рис. 17.4 показана функция **ПГ-цикла на каждый рабочий поток**. Она распределяет задачи между локальными двусторонними очередями мастера и рабочих потоков, поскольку, после того как некоторый поток позаимствовал для себя начальную задачу, он запускает последующие задачи через свою локальную очередь. На этом варианте можно изучить поведение потокового графа с очень небольшим количеством заимствований.



(a) Диаграмма параллельного потокового графа с P начальными сообщениями

```
double fig_17_4(int num_trials, int P,
                int N_per_P, double per_node_time) {
  tbb::tick_count t0, t1;
  using node_t = tbb::flow::multifunction_node<int, std::tuple<int>>;
  tbb::flow::graph g;
  node_t n(g, tbb::flow::unlimited,
    [N_per_P, per_node_time](int i,
                             node_t::output_ports_type &p) {
      spinWaitForAtLeast(per_node_time);
      if (i+1 < N_per_P) {
        std::get<0>(p).try_put(i+1);
      }
    }
  );
  tbb::flow::make_edge(tbb::flow::output_port<0>(n), n);

  for (int t = -1; t < num_trials; ++t) {
    if (!t) t0 = tbb::tick_count::now();
    for (int p = 0; p < P; ++p) {
      n.try_put(0);
    }
    g.wait_for_all();
  }
  t1 = tbb::tick_count::now();
  return (t1-t0).seconds()/num_trials;
}
```

(b) Исходный код реализации

Рис. 17.4 ❖ ПГ-цикл на каждый рабочий поток: функция организует ровно столько параллелизма, чтобы удовлетворить потребности рабочих потоков. Позаимствовав начальную задачу, рабочий поток берет остальные задачи из своей локальной двусторонней очереди

Если явно не оговорено противное, все данные о производительности, представленные в этой главе, были собраны на односокетном сервере с процессором Intel Xeon E3-1230, оснащенном 4 ядрами с двумя аппаратными потоками на ядро. Процессор имеет базовую частоту 3,4 ГГц, разделяемый L3-кеш 8 МБ и L2-кеши размером 256 КБ на каждое ядро. Система работала под управлением SUSE Linux Enterprise Server 12. Для компиляции всех примеров использовался компилятор Intel C++ Compiler 19.0 с библиотекой Threading Building Blocks 2019 с флагами `-std=c++11 -O2 -tbb`.

Микротесты производительности прогонялись с N = 65 536, а время активного ожидания составляло 100 нс, 1 мкс, 10 мкс и 100 мкс. Время выполнения, усредненное по 10 прогонам, представлено на рис. 17.5. Из этих результатов видно, что при очень малом размере задач, скажем 100 нс, накладные расходы на организацию инфраструктуры потокового графа во всех случаях приводят к падению производительности. Если размер задачи не менее 1 мкс, то распараллеливание начинает приносить плоды. А когда размер задачи достигает 100 мкс, мы уже приближаемся к идеально линейному ускорению.

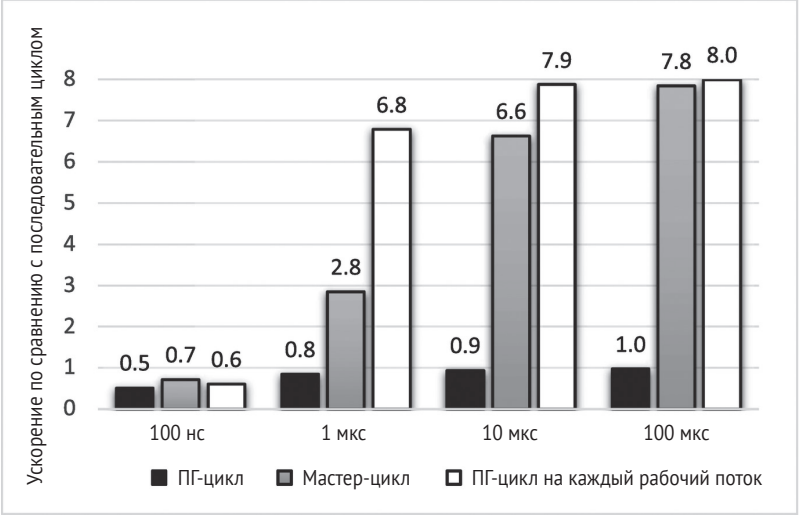


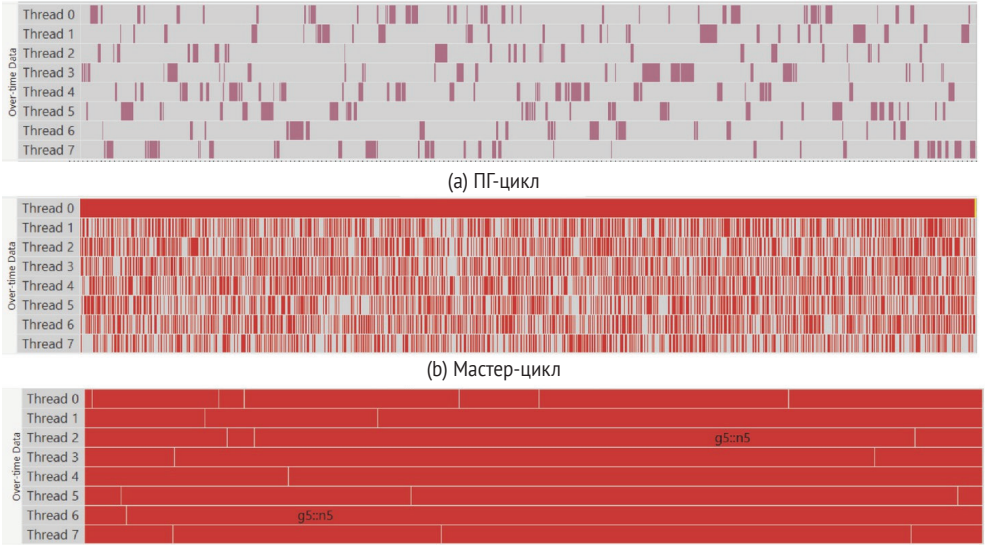
Рис. 17.5 ❖ Ускорение $T_{serial}/T_{benchmark}$ при различном времени активного ожидания

Лучше разобраться в производительности наших микротестов поможет сбор и просмотр данных трассировки в инструменте Flow Graph Analyzer (FGA), который подробно описан в конце главы. На рис. 17.6 показаны хронологии каждого потока для разных функций при времени активного ожидания 1 мкс. Все они одной длины и показывают, что делал поток на протяжении времени работы. Промежутки (серого цвета) означают, что поток не исполнял тело какого-либо узла. На рис. 17.6(а) мы видим поведение **ПГ-цикла**, который работает как последовательный цикл. Но видно, что небольшой промежуток времени между вызовом `try_put` и выходом из задачи позволяет перебрасывать задачи между потоками, потому что потоки умудряются позаимствовать задачу сразу

после ее запуска. Это отчасти объясняет довольно высокие накладные расходы в этом микротесте, показанные на рис. 17.5. Ниже в этой главе мы объясним, что большинство функциональных узлов пользуются обходом планировщика, чтобы по возможности следовать за своими данными в следующий узел (см. обсуждение конвейеров, локальности данных и привязки к потоку в главе 16, где подробно описано, почему обход планировщика улучшает производительность кеша). Поскольку узел типа `multifunction_node` помещает выходные сообщения в свои выходные порты напрямую внутри тела, он не может сразу же последовать за данными в следующий узел, минуя планировщик, – сначала он должен завершить исполнение собственного тела! Поэтому `multifunction_node` не использует обход планировщика для оптимизации локальности. В результате производительность на рис. 17.6(a) наихудшая.

На рис. 17.6(b) показан случай, когда мастер-поток генерирует все задачи, а рабочие потоки должны заимствовать каждую задачу, но после заимствования задачи могут выполняться параллельно. Поскольку рабочие потоки вынуждены заимствовать, они находят задачи гораздо медленнее, чем мастер-поток. На рисунке видно, что мастер-поток постоянно занят, т. к. может быстро извлечь задачу из своей локальной очереди, тогда в хронологиях рабочих потоков видны промежутки, в течение которых они борются друг с другом за то, чтобы позаимствовать следующую задачу из локальной очереди мастера.

На рис. 17.6(c) мы видим хорошее поведение **ПГ-цикла на каждый рабочий поток**, когда каждый поток может быстро извлечь следующую задачу из своей локальной очереди. Теперь в хронологиях очень мало пустых промежутков.



(a) ПГ-цикл

(b) Мастер-цикл

(c) ПГ-цикл на каждый рабочий поток

Рис. 17.6 ❖ Двухмиллисекундные отрезки хронологий каждого микротеста в случае, когда время активного ожидания равно 1 мкс

Глядя на эти крайние образцы поведения и на данные о производительности на рис. 17.5, мы можем без опасений порекомендовать эвристическое

правило для узлов потокового графа. В патологическом случае **Мастер-цикла** ускорение составляет всего 2,8 для тела длительностью 1 мкс, но все-таки это ускорение. Если же работа более сбалансирована, как в случае **ПГ-цикла на каждый рабочий поток**, то для тела длительностью 1 мкс получается вполне приличное ускорение. Имея в виду эти подводные камни, мы снова предлагаем грубую оценку времени выполнения – 1 мкс.

Эвристическое правило. Чтобы получить выигрыш от распараллеливания, время выполнения узлов потокового графа должно составлять не менее 1 мкс. Это несколько тысяч тактов ЦП, и если вы предпочитаете выражение в тактах, то мы предлагаем взять за основу 10 000 тактов.

Как и в случае TBB-алгоритмов, это правило *не* означает, что нужно любой ценой избегать узлов, работающих менее 1 мкс. Проблема возникает, только если в потоковом графе преобладают узлы с малым временем выполнения. Если же имеются узлы с разным временем выполнения, то накладные расходы в малых узлах могут оказаться пренебрежимо малы по сравнению со временем выполнения больших узлов.

Что делать, если узлы слишком малы

Если некоторые узлы потокового графа меньше рекомендуемого порога 1 мкс, то у нас есть три варианта: (1) ничего не делать, если узел не оказывает существенного влияния на общее время выполнения приложения; (2) объединить узел с окружающими его узлами, чтобы повысить зернистость; (3) использовать облегченную политику выполнения.

Если зернистость узла мала, но и его вклад в общее время выполнения тоже мал, то этот узел можно спокойно проигнорировать – пусть живет, как есть. В таких случаях чистота дизайна может быть важнее несущественной потери эффективности.

Если мелкой зернистостью узла нельзя пренебречь, то один из вариантов – объединить его с соседними узлами. Так ли необходимо инкапсулировать логику, отделив ее от предшественников и преемников? Если у узла имеется один предшественник или один преемник с таким же уровнем конкурентности, то объединить узлы, скорее всего, будет несложно. Если предшественников или преемников несколько, то, быть может, операции, выполняемые узлом, можно скопировать в каждый из них. Как бы то ни было, объединение узлов может стать выходом из положения, если семантика графа при этом не изменяется.

Наконец, узел можно изменить, так чтобы использовалась облегченная политика выполнения. Для этого нужно задать аргумент шаблона при конструировании узла, например:

```
tbb::flow::function_node<int, int, lightweight> n(...)
```

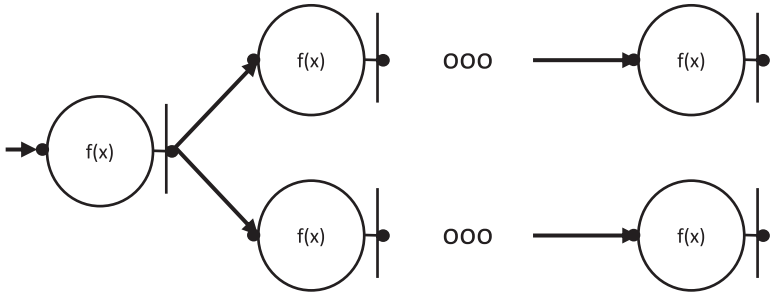
Эта политика показывает, что в теле узла мало работы, поэтому его по возможности следует выполнять без накладных расходов на планирование задачи.

Существует три облегченные политики: `queueing_lightweight`, `rejecting_lightweight` и `lightweight`. Подробно они описаны в приложении B. Все функциональные узлы, кроме `source_node`, поддерживают облегченные политики. Облегчен-

ный узел может не запускать задачу для выполнения тела, а выполнить его прямо внутри `try_put` в контексте вызывающего потока. В таком случае накладных расходов на запуск задачи не будет, но и у других потоков не будет возможности позаимствовать задачу, а значит, параллелизм ограничивается!

На рис. 17.7 показано два простых графа, которые можно использовать для демонстрации преимуществ и рисков облегченных политик: первый представляет собой линейную цепочку объектов `multifunction_node`, второй – объект `multifunction_node`, связанный с двумя линейными цепочками таких же объектов.

(a) Одна цепочка узлов `multifunction_node`



(b) Две цепочки узлов `multifunction_node`

Рис. 17.7 ❖ Потоковый граф для изучения последствий облегченных политик

На рис. 17.8 показано влияние облегченной политики на графы, изображенные на рис. 17.7, когда цепочки состоят из 1000 узлов и во всех используется одна и та же политика выполнения (не важно, облегченная или нет). Мы отправляем в каждый граф единственное сообщение и изменяем время работы узла с 0 до 1 мс. Отметим, что в случае единственной цепочки никакого параллелизма при одном сообщении быть не может, а две цепочки могут достичь максимального ускорения 2.

Облегченная политика не может ограничить параллелизм для графа с одной цепочкой, поскольку в нем изначально нет параллелизма. Поэтому на рис. 17.8 мы видим, что она улучшает производительность во всех случаях, хотя ее влияние убывает по мере возрастания зернистости. Для одной цепочки отношение стремится к 1.0, поскольку накладные расходы на запуск задач становятся пренебрежимо малы по сравнению со временем работы тела. При двух цепочках параллелизм потенциально имеется. Но если во всех узлах используется облегченная политика, то обе цепочки будут исполняться потоком, который выполнил первый узел `multifunction_node`, поэтому потенциальный параллелизм не реализуется. Как и предсказывает наше эвристическое правило, когда время выполнения приближается к одной микросекунде, преимущества облегченной политики отступают перед ограниченностью параллелизма. Даже если время

активного ожидания в узле равно 0,1 мкс, отношение уже опускается ниже 1. И приближается к 0,5, поскольку сериализация графа сводит на нет ожидаемое ускорение 2 от использования двух цепочек.

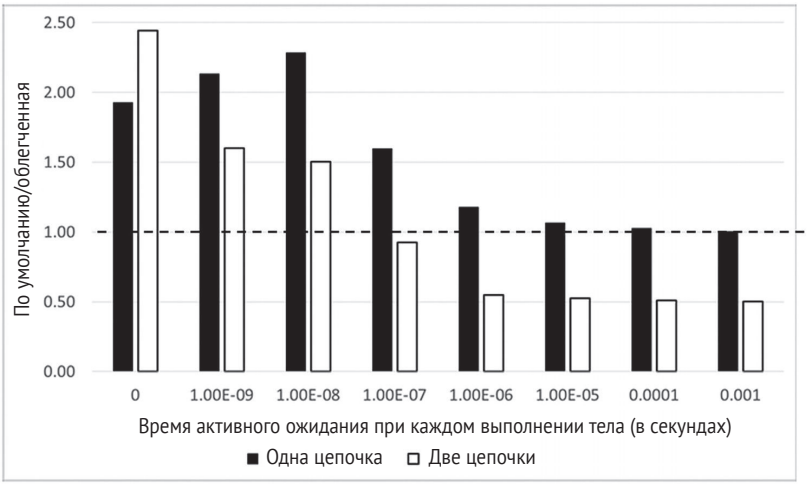


Рис. 17.8 ❖ Влияние облегченной политики для графа с одной и двумя цепочками узлов. Значение больше 1 означает, что облегченная политика повысила производительность

Решение проблемы зернистости с помощью объединения узлов или использования облегченной политики может снизить накладные расходы, но, как мы видим, оно также может ограничить масштабируемость. Эти «оптимизации» способны привести к значительному улучшению, но использовать их надо осмотрительно, чтобы вреда не оказалось больше, чем пользы.

Потребление памяти и локальность данных

В отличие от параллельных TBB-алгоритмов, которые обходят структуры данных, потоковый граф передает структуры данных от одного узла к другому. В роли сообщений могут выступать примитивные типы, объекты, указатели или, в случае графа зависимостей, объекты типа `tbb::flow::continue_msg`. Для достижения максимальной производительности мы должны учитывать как локальность данных, так и потребление памяти. Оба вопроса обсуждаются в этом разделе.

Локальность данных в потоковых графах

Данные передаются между узлами, и когда узел получает сообщение, он применяет к нему свое тело как задачу TBB. Эта задача планируется тем же диспетчером с заимствованием работ, что и все остальные задачи TBB. На рис. 17.6(а), где последовательный цикл выполняется как потоковый граф, мы видели, что задача, запущенная одним потоком, может быть выполнена другим. Однако мы отметили, что это отчасти связано с тем, что в наших микротестах использовались узлы типа `multifunction_node`, которые ради оптимизации производительности не пользуются обходом планировщика.

В общем случае функциональные узлы, в т. ч. source_node, function_node и continue_node, применяют обход планировщика, если какой-нибудь преемник может быть выполнен немедленно. Если данные, к которым обращается такой узел, помещаются в кеш, то их сможет повторно использовать тот же поток, когда станет исполнять роль преемника.

Поскольку в потоковом графе мы можем получить выигрыш от локальности данных, то имеет смысл учитывать размер данных и даже разбивать данные на меньшие порции, так чтобы локальность действительно приносила выгоду в результате обхода планировщика. Для демонстрации этого эффекта вернемся к программе транспонирования матрицы из главы 16. Теперь будем передавать три пары матриц a, b в структуре FGMsg, показанной на рис. 17.9. На

```
struct FGMsg {
  int N;
  double *a;
  double *b;
  FGMsg() : N(0), a(0), b(0) {}
  FGMsg(int _N, double *_a, double *_b) : N(_N), a(_a), b(_b) {}
};

double fig_17_9(int N, double *a[3], double *b[3]) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::flow::graph g;
  int i = 0;
  tbb::flow::source_node<FGMsg> initialize{g,
  [&](FGMsg &msg) -> bool {
    if (i < 3) {
      msg = {N, setArray(N, a[i]), setArray(N, b[i])};
      ++i;
      return true;
    } else {
      return false;
    }
  }, false};
  tbb::flow::function_node<FGMsg, FGMsg> transpose{g,
    tbb::flow::unlimited,
    [](const FGMsg &msg) -> FGMsg {
      serialTranspose(msg.N, msg.a, msg.b);
      return msg;
  }};
  tbb::flow::function_node<FGMsg> check{g, tbb::flow::unlimited,
    [](const FGMsg &msg) -> FGMsg {
    checkArray(msg.N, msg.b);
  }};
  tbb::flow::make_edge(initialize, transpose);
  tbb::flow::make_edge(transpose, check);
  initialize.activate();
  g.wait_for_all();
}
```

Рис. 17.9 ❖ Граф, который отправляет последовательность матриц функции transpose, где каждая транспонируется с применением простой последовательной процедуры из главы 16

рис. 16.6–16.13 мы видели три реализации ядра транспонирования: последовательную, без учета кеша и с помощью parallel_for.

В этой простой реализации передаются полные матрицы, которые транспонируются с учетом кеша. Как и следовало ожидать, производительность оставляет желать лучшего. На тестовой машине эта программа выполнялась всего на 8 % быстрее, чем последовательная кеш-зависимая реализация транспонирования из главы 16, примененная трижды подряд, по одному разу для каждой пары матриц. Это не удивительно, поскольку тест потребляет много памяти – попытка выполнить несколько транспонирований параллельно мало что дает, если мы не можем подавать на вход одной операции транспонирования нужные ей данные из кеша. Если сравнить этот простой потоковый граф с последовательным кеш-независимым транспонированием из главы 16, то ситуация выглядит еще хуже – для обработки трех пар матриц требуется в 2,5 раза больше времени. По счастью, есть много способов улучшить производительность этого потокового графа. Например, можно использовать последовательную кеш-независимую реализацию в узле transpose. Или воспользоваться реализацией на основе parallel_for из главы 16, применив в узле transpose диапазон blocked_range2d и разбиватель simple_partitioner. Вскоре мы увидим, что оба варианта дают значительно большее ускорение, чем 1,08.

Однако мы должны будем посылать в сообщениях блоки матриц, а не пары матриц a и b целиком. Для этого добавим в структуру сообщения диапазон blocked_range2d:

```
using RType = tbb::blocked_range2d<int, int>;
struct FGTiledMsg {
  int N;
  double *a;
  double *b;
  RType r;
  FGTiledMsg() : N(0), a(0), b(0), r(0, 0, 0, 0, 0, 0) {}
  FGTiledMsg(int _N, double *_a, double *_b, const RType &_r)
    : N(_N), a(_a), b(_b), r(_r) {}
};
```

После этого можно построить реализацию, в которой узел initialize отправляет блоки матриц a и b в виде сообщений – сначала все блоки одной пары матриц, потом следующей. На рис. 17.10 приведена одна из возможных реализаций. В ней узел source_node хранит стек, имитируя разбиение в глубину и выполнение блоков, которое имело бы место при рекурсивном разбиении диапазонов в цикле parallel_for. Мы не станем подробно описывать реализацию на рис. 17.10, а просто отметим, что она отправляет блоки, а не матрицы целиком.

```
double fig_17_10(int N, double *a[3], double *b[3], int gs) {
  tbb::tick_count t0 = tbb::tick_count::now();
  tbb::flow::graph g;
  int i = 0;
  std::vector<RType> stack;
  stack.push_back(RType(0, N, gs, 0, N, gs));
  tbb::flow::source_node<FGTiledMsg> initialize{g,
  [&](FGTiledMsg &msg) -> bool {
    if (i < 3) {
      if (stack.empty()) {
        if (++i == 3) return false;
        stack.push_back(RType(0, N, gs, 0, N, gs));
      }
      RType r = stack.back();
      stack.pop_back();
      while (r.is_divisible()) {
        RType rhs(r, tbb::split());
        stack.push_back(rhs);
      }
      msg = {N, setBlock(r, a[i]), setTransposedBlock(r, b[i]), r};
      return true;
    } else {
      return false;
    }
  }, false};
  tbb::flow::function_node<FGTiledMsg, FGTiledMsg>
  transpose{g, tbb::flow::unlimited,
    [](const FGTiledMsg &msg) {
    double *a = msg.a, *b = msg.b;
    int N = msg.N, ie = msg.r.rows().end(), je = msg.r.cols().end();
    for (int i = msg.r.rows().begin(); i < ie; ++i) {
      for (int j = msg.r.cols().begin(); j < je; ++j) {
        b[j*N + i] = a[i*N + j];
      }
    }
    return msg;
  }};
  tbb::flow::function_node<FGTiledMsg> check{g, tbb::flow::unlimited,
    [](const FGTiledMsg &msg) {
    checkTransposedBlock(msg.r, msg.b);
  }};
  tbb::flow::make_edge(initialize, transpose);
  tbb::flow::make_edge(transpose, check);
  initialize.activate();
  g.wait_for_all();
  return total_time;
}
```

Рис. 17.10 ❖ Граф, который отправляет последовательность плиток матрицы, подлежащих транспонированию, используя для этой цели диапазон `blocked_range2d`, описанный в главе 16

На рис. 17.11 показано ускорение, достигаемое в нескольких вариантах транспонирования матрицы на нашей тестовой машине. Мы видим, что в первой реализации, помеченной «потоковый граф», улучшение составило жалкие 8 %. Реализация pfor-br2d основана на цикле `parallel_for` (рис. 16.11) с применением `blocked_range2d` и `simple_partitioner` трижды для каждой пары матриц. Остальные столбики соответствуют оптимизированным версиям потокового графа: «потоковый граф + кеш-независмый» аналогична версии рис. 17.9, но из тела узла `transpose` производятся обращения к последовательной кеш-независимой реализации транспонирования матрицы; в версии «потоковый граф + pfor-br2d» в теле `transpose` используется `parallel_for`; версия «потоковый граф с замощением» – это реализация на рис. 17.10, а версия «потоковый граф с замощением + pfor2d» аналогична версии рис. 17.10, но для обработки плиток используется `parallel_for`. Лучше всего работает потоковый граф с замощением, показанный на рис. 17.10.

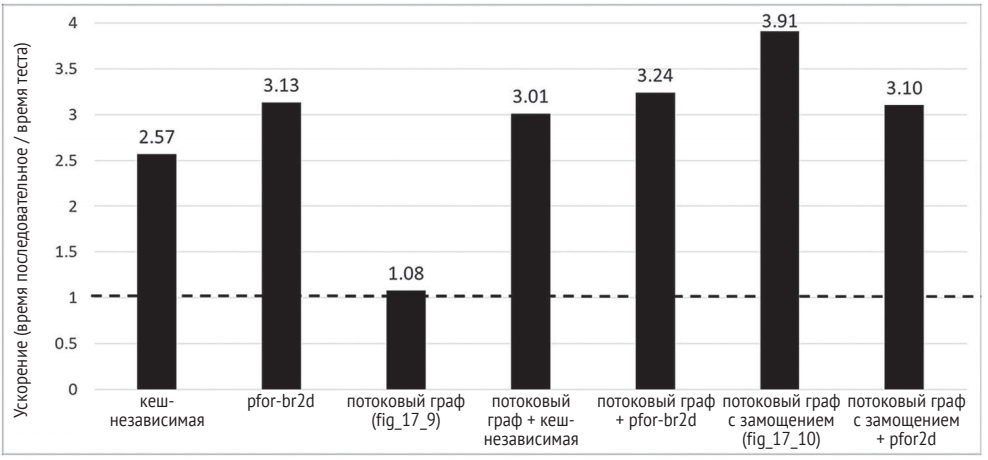


Рис. 17.11 ❖ Ускорение для различных вариантов транспонирования матрицы. Мы использовали 32×32 плиток, потому что такая конфигурация давала наилучший результат в нашей тестовой системе

Может показаться удивительным, что потоковый граф с замощением с вложенными `parallel_for` работал хуже, чем граф с замощением, но без вложенного параллелизма. В главе 9 мы говорили, что в TBB вложенный параллелизм можно использовать безнаказанно, так в чем же дело? Вмешалась грубая реальность – при настройке TBB-приложений часто приходится отказываться от полной компонуемости в пользу производительности (см. врезку «Аспекты производительности»). В данном случае вложенный параллелизм конфликтует с оптимизацией кеша, которую мы так стремились реализовать. Каждому узлу отправлялась для обработки плитка, которая помещается в его кеш данных, а вложенный параллелизм сводит все усилия на нет, поскольку эта плитка разделяется с другими потоками.

Аспекты компонуемости

Говоря о компонуемости, можно выделить три пожелания:

1) корректность (безусловный императив);
2) возможность использования (практический вопрос);
3) производительность (крайне желательно).

Что касается первого пункта, то мы надеемся, что сможем комбинировать код произвольным образом, не опасаясь, что он вдруг перестанет работать (даст неверный результат). TBB дает нам такую возможность, и это, в общем-то, можно считать решенной проблемой, если не считать одной мелочи – из-за недетерминированного порядка выполнения ответ может меняться при работе с числами конечной точности, например с плавающей точкой. Как обеспечить «корректность» компонуемости в этом случае, мы обсуждали в главе 16.

Что касается второго пункта, то мы надеемся, что программа не «упадет». Во многих случаях это чисто практический вопрос, потому что самая типичная проблема (неограниченное потребление памяти) теоретически могла бы быть решена при наличии бесконечной памяти. ☺ TBB много делает в этом направлении, предлагая преимущества перед другими моделями программирования (например, OpenMP). Но и TBB нуждается в некоторой помощи при работе со слабо структурированными потоковыми графами, поэтому мы обсудим использование узлов типа `limiter_node`, которые позволяют держать потребление памяти в узде – это особенно важно для больших потоковых графов.

Наконец, что касается оптимальной производительности, то мы не знаем общего решения, которое позволило бы в полной мере совместить производительность с компонуемостью. Реальность такова, что когда высокооптимизированный код конкурирует с другим кодом, работающим на том же оборудовании, страдает производительность того и другого. Это означает, что ручная настройка кода может принести пользу. К счастью, TBB предлагает нам средства настройки, а инструменты типа Flow Graph Analyzer помогают настраивать программу зряче. Наш опыт показывает, что хорошо настроенный код может и работать быстро, и компонуемость сохранить, но технологии, которая позволила бы слепо использовать код и надеяться на наивысшую производительность, не существует. «Достаточно хорошая» производительность встречается часто, но для получения «отличной» придется поработать.

Не следует придавать слишком много веса конкретным результатам на рис. 17.11 – в конце концов, это лишь один микротест с ограничением размера памяти. Но все же из него понятно, что выигрыш можно получить, рассматривая размер узлов с точки зрения не только зернистости, но и локальности данных. При переходе от наивной реализации, когда массивы передавались целиком без настройки кода в узлах, к версии с потоковым графом с замощением и учетом кеша был получен значительный прирост производительности.

Выбор наилучшего типа сообщения и ограничение количества сообщений в графе

Подавая сообщения в граф или делая их копии при прохождении по разным путям, мы потребляем память. Поэтому нас может интересовать не только локальность данных, но и ограничение роста потребления.

Когда сообщение попадает в узел графа потоков данных, оно может быть скопировано во внутренний буфер узла. Например, если последовательный узел вынужден отложить запуск задачи, он сохраняет поступающие сообщения

в очереди до тех пор, пока не сможет запустить задачу для их обработки. Если в графе циркулируют очень большие объекты, то копирование станет в копеечку! Поэтому всюду, где возможно, лучше передавать указатели на большие объекты, а не сами объекты.

В стандарте C++11 появились классы (в пространстве имен `std`) `unique_ptr` и `shared_ptr`, очень полезные, когда нужно упростить управление памятью, выделенной объектам, которые передаются в потоковом графе по указателю. Например, предположим, что объекты класса `BigObject` на рис. 17.12 очень велики и конструируются долго. Если передавать объект по указателю `shared_ptr`, то копировать во входной буфер последовательного узла `n` придется только указатель, а не весь объект `BigObject`. Кроме того, раз мы используем `shared_ptr`, то каждый объект `BigObject` автоматически уничтожается по достижении конца графа, когда счетчик ссылок на него обращается в ноль. Очень удобно!

```
class BigObject {
  const int id;  // плюс непоказанные данные большого размера
public:
  BigObject() : id(-1) { }
  BigObject(int i) : id(i) { spinWaitForAtLeast(0.001); }
  BigObject(const BigObject &b) : id(b.id) {
    spinWaitForAtLeast(0.001); // моделирует время копирования
  }
  int get_id() const {return id;}
};

void fig_17_12() {
  tbb::flow::graph g;

  tbb::flow::function_node<std::shared_ptr<BigObject>, int>
  n(g, tbb::flow::serial,
    [](std::shared_ptr<BigObject> b) -> int {
      int id = b->get_id();
      spinWaitForAtLeast(0.01);
      return id;
    }
  );

  for (int i = 0; i < 100; ++i) {
    n.try_put(std::make_shared<BigObject>(i));
  }
  g.wait_for_all();
}
```

Рис. 17.12 ❖ Использование `std::shared_ptr`, чтобы избежать медленного копирования и упростить управление памятью

Разумеется, при работе с указателями на объекты нужна осторожность. Раз мы передаем указатель, а не объект, то в каждый момент времени к объекту может обращаться несколько узлов. Особенно это относится к графам с функциональным параллелизмом, когда одно и то же сообщение транслируется нескольким узлам. Класс `shared_ptr` корректно обрабатывает увеличение и уменьшение счетчика ссылок, но мы сами должны гарантировать правильное использование ребер, чтобы предотвратить потенциальную гонку при доступе к объекту, на который ведет указатель.

При обсуждении отображения узлов на задачи мы видели, что когда сообщение приходит в функциональный узел, либо запускается задача, либо сообщение запоминается в буфере. При проектировании графа потоков данных мы не должны забывать о том, сколько памяти потребляют эти буферы и задачи.

Например, взгляните на рис. 17.13. На нем показаны два узла, `serial_node` и `unlimited_node`, оба содержат длинный цикл активного ожидания. В цикле for быстро создается много входных сообщений для обоих узлов. Узел `serial_node` последовательный, поэтому его внутренний буфер будет быстро расти, поскольку он получает сообщения быстрее, чем завершаются его задачи. Наоборот, узел `unlimited_node` запускает задачу сразу по приходе сообщения и быстро затопляет систему очень большим количеством задач – гораздо большим, чем количество рабочих потоков. Запущенные задачи буферизуются во внутренних очередях рабочих потоков. В обоих случаях занятая графом память быстро растет, потому что сообщениям `BigObject` разрешено поступать в граф быстрее, чем они могут быть обработаны.

В нашем примере для подсчета количества объектов, существующих в данный момент времени, используется атомарный счетчик `bigObjectCount`. В конце выполнения печатается максимальное встретившееся значение счетчика. После выполнения кода на рис. 17.13 при `A_VERY_LARGE_NUMBER=4096` было напечатано `maxCount == 8094`. При использовании как `serial_node`, так и `unlimited_node` количество объектов `BigObject` растет очень быстро!

Существует три распространенных подхода к управлению потреблением ресурсов в потоковом графе: (1) использовать узел типа `limiter_node`; (2) использовать лимиты конкурентности; (3) использовать передачу маркера.

Мы воспользуемся узлом `limiter_node`, чтобы ограничить количество сообщений в графе в каждый момент времени. На рис. 17.14 показана часть интерфейса класса `limiter_node`.

```
tbb::atomic<int> bigObjectCount;
int maxCount = 0;

class BigObject {
   const int id;
   /* И много других данных */
public:
   BigObject() : id(-1) { }
   BigObject(int i) : id(i) {
     int cnt = bigObjectCount.fetch_and_increment() + 1;
     if (cnt > maxCount)
       maxCount = cnt;
   }
   BigObject(const BigObject &b) : id(b.id) { }
   virtual ~BigObject() {
     bigObjectCount.fetch_and_decrement();
   }
   int get_id() const {return id;}
};

using BigObjectPtr = std::shared_ptr<BigObject>;

void fig_17_13() {
  tbb::flow::graph g;
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr>
  serial_node{g, tbb::flow::serial,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr>
  unlimited_node{g, tbb::flow::unlimited,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  bigObjectCount = 0;
  for (int i = 0; i < A_VERY_LARGE_NUMBER; ++i) {
    serial_node.try_put(std::make_shared<BigObject>(i));
    unlimited_node.try_put(std::make_shared<BigObject>(i));
  }
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

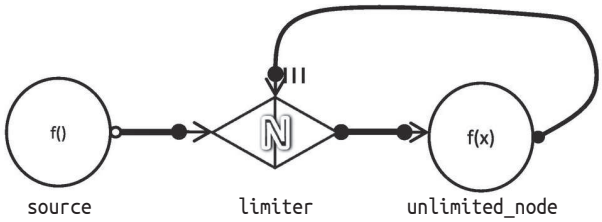
Рис. 17.13 ❖ Пример использования последовательного узла типа `function_node`, `serial_node` и неограниченного узла `unlimited_node` типа `function_node`

```
class limiter_node : public graph_node,
  public receiver<T>, public sender<T> {
public:
    limiter_node( graph &g,  size_t threshold,
                  int number_of_decrement_predecessors = 0 );
    limiter_node( const limiter_node &src );

    // a continue_receiver
     implementation-dependent-type decrement;

    // receiver<T>
    typedef T input_type;
    bool try_put( const input_type &v );

    // sender<T>
    typedef T output_type;
};
```

Рис. 17.14 ❖ Часть интерфейса `limiter_node`, используемая в примерах

Объект `limiter_node` хранит внутри себя счетчик проходящих через него сообщений. Сообщение, отправленное в порт `decrement` узла, уменьшает счетчик на единицу, в результате чего узел может пропустить дополнительные сообщения. Если счетчик становится равен величине `threshold`, то новые сообщения, поступающие во входной порт, отвергаются.

На рис. 17.15 узел `source` типа `source_node` генерирует много объектов `BigObject`. Но `source_node` запускает новую задачу для генерации сообщения только тогда, когда предыдущее сообщение потреблено. Мы вставили узел `limiter` типа `limiter_node`, сконструированный с `limit=3`, между узлами `source` и `unlimited_node`, чтобы ограничить количество сообщений, посылаемых `unlimited_node`. Мы также добавили ядро, идущее из `unlimited_node` обратно в порт `decrement` узла `limiter_node`. Теперь количество сообщений, прошедших через `limiter`, будет не более чем на 3 превосходить количество сообщений, полученных через его порт `decrement`.



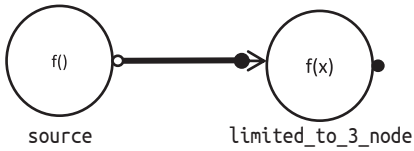
(a) Структура графа

```
void fig_17_15() {
  tbb::flow::graph g;

  int src_count = 0;
  tbb::flow::source_node<BigObjectPtr>
  source{g, [&src_count] (BigObjectPtr &m) -> bool {
      if (src_count < A_VERY_LARGE_NUMBER) {
        m = std::make_shared<BigObject>(src_count++);
        return true;
      }
      return false;
    }, false
  };
  tbb::flow::limiter_node<BigObjectPtr> limiter{g, 3};
  tbb::flow::function_node<BigObjectPtr,
tbb::flow::continue_msg>
  unlimited_node{g, tbb::flow::unlimited,
    [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return tbb::flow::continue_msg();
    }
  };
  tbb::flow::make_edge(source, limiter);
  tbb::flow::make_edge(limiter, unlimited_node);
  tbb::flow::make_edge(unlimited_node, limiter.decrement);

  bigObjectCount = 0;
  maxCount = 0;
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

(b) Реализация графа

Рис. 17.15 ❖ Использование `limiter_node`, чтобы в каждый момент времени количество объектов `BigObject` в узле `unlimited_node` было не больше 3

Для ограничения потребления ресурсов можно также использовать лимиты конкурентности, как показано на рис. 17.16. В этой программе имеется узел, который можно безопасно выполнить, даже если конкурентность неограниченна, но мы решили наложить ограничение на количество одновременно запускаемых задач.



(a) Структура графа

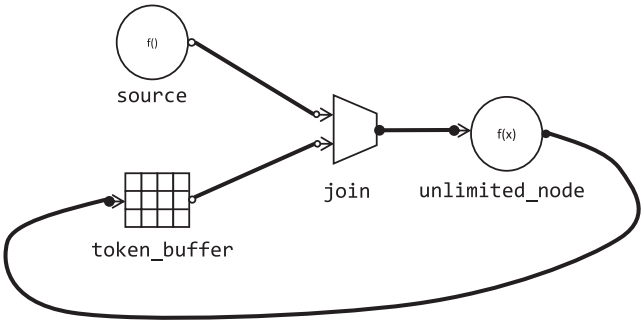
```
void fig_17_16() {
  tbb::flow::graph g;

  int src_count = 0;
  tbb::flow::source_node< BigObjectPtr > source{g,
    [&] (BigObjectPtr &m) -> bool {
      if (src_count < A_VERY_LARGE_NUMBER) {
        m = std::make_shared<BigObject>(src_count++);
        return true;
      }
      return false;
    }, false};
  tbb::flow::function_node<BigObjectPtr, BigObjectPtr,
                           tbb::flow::rejecting>
  limited_to_3_node{g, 3, [] (BigObjectPtr m) {
      spinWaitForAtLeast(0.0001);
      return m;
    }};
  tbb::flow::make_edge(source, limited_to_3_node);

  bigObjectCount = 0;
  maxCount = 0;
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

(b) Реализация графа

Рис. 17.16 ❖ Использование `tbb::flow::rejecting policy` и лимита конкурентности, чтобы в узле `limited_to_3_node` одновременно присутствовало не более трех объектов `BigObject`

Мы можем отключить внутреннюю буферизацию в узле `function_node`, задав при конструировании политику выполнения `flow::rejecting` или `flow::rejecting_lightweight`. Узел `source_node` на рис. 17.16 продолжает генерировать новые сообщения, только если они кем-то потребляются.

И последний подход к ограничению потребления ресурсов в графе потоков данных – воспользоваться системой с передачей маркера. В главе 2 мы видели, что в алгоритме `tbb::parallel_pipeline` маркеры используются, чтобы ограничить количество элементов, одновременно находящихся в конвейере. Аналогичную систему можно создать с помощью маркеров и резервирующего узла типа `join_node`, как показано на рис. 17.17. В этом примере мы создаем узел `source` типа `source_node` и узел `token_buffer` типа `buffer_node`. Эти два узла со-



(a) Структура графа

```
void fig_17_17() {
  using token_t = int;
  tbb::flow::graph g;
  int src_count = 0;
  tbb::flow::source_node<BigObjectPtr> source{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src_count < A_VERY_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src_count++);
      return true;
    }
    return false;}, false};
  tbb::flow::buffer_node<token_t> token_buffer{g};
  tbb::flow::join_node<std::tuple<BigObjectPtr, token_t>,
                       tbb::flow::reserving> join{g};
  tbb::flow::function_node<std::tuple<BigObjectPtr,token_t>, token_t>
  unlimited_node{g, tbb::flow::unlimited,
  [] (const  std::tuple<BigObjectPtr, token_t> &m) {
    spinWaitForAtLeast(0.0001);
    return std::get<1>(m);}};
  tbb::flow::make_edge(source, tbb::flow::input_port<0>(join));
  tbb::flow::make_edge(token_buffer, tbb::flow::input_port<1>(join));
  tbb::flow::make_edge(join, unlimited_node);
  tbb::flow::make_edge(unlimited_node, token_buffer);
  bigObjectCount = 0; maxCount = 0;
  for (token_t i = 0; i < 3; ++i) token_buffer.try_put(i); // fill it
  source.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

(b) Реализация графа

Рис. 17.17 ❖ В схеме с передачей маркеров используются маркеры и узел типа `join_node` с политикой выполнения `tbb::flow::reserving`, чтобы ограничить количество элементов в узле `unlimited_node`

единены с входами резервирующего узла join типа join_node. Узел join_node< tuple< BigObjectPtr, token_t >, flow::reserving > потребляет элементы, только когда может предварительно зарезервировать емкость для входных сообщений на каждом из своих портов. Поскольку source_node прекращает генерировать новые сообщения, если предыдущие не были потреблены, то наличие свободных маркеров в token_buffer ограничивает количество элементов, порождаемых узлом source_node. Когда маркеры возвращаются в token_buffer узлом unlimited_node, их можно сопрячь с дополнительными сообщениями, генерируемыми узлом source, а значит, дать source возможность запустить новые задачи.

На рис. 17.18 показано ускорение по сравнению с последовательным выполнением тел узлов при каждом из вышеописанных подходов. Время работы узла составляло 100 мкс, и мы видим, что подход с передачей маркеров имеет чуть более высокие накладные расходы, хотя все три подхода демонстрируют ожидаемое ускорение, близкое к 3.

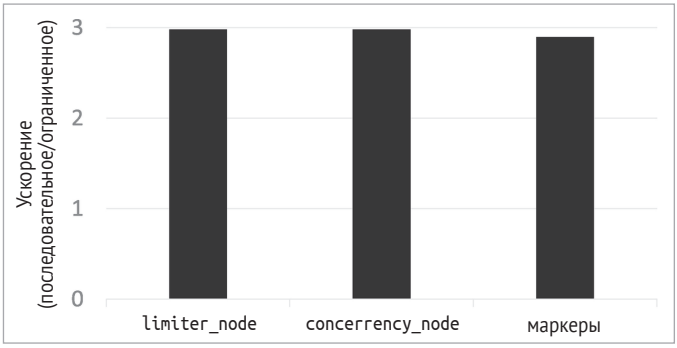


Рис. 17.18 ❖ Все три подхода ограничивают ускорение, потому что только три узла могут одновременно присутствовать в узле n

На рис. 17.18 маркер имеет тип int. В общем случае маркер может иметь любой тип, в т. ч. быть большим объектом или указателем. Например, можно было бы использовать в качестве маркеров объекты BigObjectPtr, если бы мы хотели рециклировать объекты BigObject, а не создавать их для каждого нового входного сообщения.

Арены задач и потоковый граф

На поведение задачи и обобщенных параллельных TBB-алгоритмов влияют как явные, так и неявные арены задач. Арена, на которой запущены задачи, контролирует, какие потоки могут принимать участие в их выполнении. В главе 11 мы видели, как явные и неявные арены используются для управления количеством потоков, участвующих в параллельном выполнении работ. В главах 12–14 было показано, что явные арены задач можно использовать совместно с объектами task_sheduler_observer, чтобы задавать свойства потоков, заходящих на арену. Поскольку арены задач влияют на располагаемый параллелизм и локальность данных, в этом разделе мы внимательнее рассмотрим, как арены сочетаются с потоковыми графами.

Арена по умолчанию для потокового графа

Когда объект `tbb::flow::graph` конструируется, он запоминает ссылку на арену сконструировавшего его потока. Любая задача, запущенная для выполнения какой-то работы в графе, запускается на этой арене, а не на арене запустившего ее потока.

Почему так?

Дело в том, что потоковые графы структурированы слабее, чем параллельные TBB-алгоритмы. В TBB-алгоритмах используется параллелизм типа разветвление–соединение, и поведение арен задач ему хорошо соответствует – у каждого мастер-потока имеется своя арена по умолчанию, так что если разные мастер-потоки выполняют алгоритмы конкурентно, то их задачи изолированы друг друга, находясь на разных аренах. Но в потоковом графе может быть один или несколько мастер-потоков, явно вставляющих сообщения в тот же самый граф. Если задачи, связанные с этими взаимодействиями, запускаются на арене каждого мастера, то одни задачи в графе окажутся изолированы от других задач в том же графе. Вряд ли это то поведение, которое нам нужно.

Поэтому все задачи запускаются на одной арене – того потока, который сконструировал объект графа.

Изменение арены задач, используемой потоковым графом

Арену задач, используемую графом, можно изменить, вызвав функцию графа `reset()`. В результате граф инициализируется повторно и в том числе запоминает новую арену задач. Это демонстрируется на рис. 17.19, где конструируется простой граф с одним узлом `function_node`, который печатает количество слотов на той арене, на которой выполняется его тело. Поскольку объект графа конструируется главным потоком, граф будет использовать арену по умолчанию, для которой при инициализации задано восемь слотов.

В первых трех вызовах `n.try_put` на рис. 17.19 граф `g` не сбрасывается, поэтому мы видим, что задачи выполняются на арене по умолчанию с 8 слотами.

```
Without reset:
default : 8
a2 : 8
a4 : 8
```

Но во второй части программы мы вызываем `reset` для повторной инициализации графа, и узел сначала выполняется на арене по умолчанию, потом на арене `a2` и, наконец, на арене `a4`.

```
With reset:
default : 8
a2 : 2
a4 : 4
```

```
void fig_17_19() {
  tbb::task_scheduler_init init{8};
  tbb::task_arena a2{2};
  tbb::task_arena a4{4};

  tbb::flow::graph g;
  tbb::flow::function_node<std::string> f{g, tbb::flow::unlimited,
    [](const std::string &str) {
      int P = tbb::this_task_arena::max_concurrency();
      std::cout << str << " : " << P << std::endl;
    }
  };

  std::cout << "Without reset:" << std::endl;
  f.try_put("default");
  g.wait_for_all();
  a2.execute( [&]() {
    f.try_put("a2");
    g.wait_for_all();
  } );
  a4.execute( [&]() {
    f.try_put("a4");
    g.wait_for_all();
  } );

  std::cout << "With reset:" << std::endl;
  f.try_put("default");
  g.wait_for_all();
  a2.execute( [&]() {
    g.reset();
    f.try_put("a2");
    g.wait_for_all();
  } );
  a4.execute( [&]() {
    g.reset();
    f.try_put("a4");
    g.wait_for_all();
  } );
}
```

Рис. 17.19 ❖ Использование `graph::reset` с целью изменить арену задач, применяемую графом

Задание количества потоков, привязки потока к ядру и т. д.

Теперь, зная, как ассоциировать арены задач с потоковыми графами, мы можем использовать все описанные в главах 11–14 оптимизации производительности, которые зависят от арены задач. Например, можно с помощью арен изолировать один потоковый граф от другого. Или привязать потоки к ядрам для конкретной арены задач, воспользовавшись объектом `task_scheduler_observer`, а затем ассоциировать эту арену с потоковым графом.

Рекомендации по работе с потоковыми графами: что полезно, а что вредно

API потокового графа гибкий – быть может, даже слишком гибкий. Когда впервые видишь потоковый граф, интерфейс ошеломляет – так много в нем возможностей. В этом разделе мы дадим несколько рекомендаций на основе собственного опыта работы с этим высокоуровневым интерфейсом. Однако, как и эвристическое правило для выбора времени работы узла, это не более чем советы. Существует много заслуживающих уважения паттернов, которые здесь не освещены, и мы уверены, что у некоторых паттернов, которых мы советуем избегать, могут быть вполне разумные применения. Мы представляем хорошо известные и апробированные методы, но у вас может быть другая ситуация и иная точка зрения.

Полезно: использовать вложенный параллелизм

Как и конвейер, потоковый граф лучше масштабируется, если использовать параллельные узлы (`flow::unlimited`), и хуже, если узлы последовательные. Один из способов увеличить масштабируемость – использовать вложенные параллельные алгоритмы внутри узлов потокового графа. В основе TBB лежит компонуемость, поэтому вложенный параллелизм следует использовать всюду, где возможно.

Вредно: использовать многофункциональные узлы вместо вложенного параллелизма

В этой книге мы уже не раз видели, что параллельные TBB-алгоритмы, например `parallel_for` и `parallel_reduce`, тщательно оптимизированы и включают такие средства, как диапазоны и разбиватели, которые позволяют дополнительно повысить производительность. Мы также видели, что интерфейс потоковых графов весьма выразителен – можно создавать графы с циклами и использовать узлы типа `multifunction_node`, которые при каждом выполнении выводят много сообщений. Поэтому следует сознательно искать в графах такие места, которые лучше выразить с помощью вложенного параллелизма. Простой пример приведен на рис. 17.20.

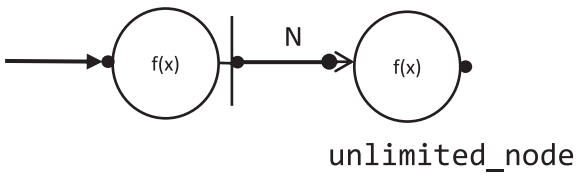


Рис. 17.20 ❖ Узел типа `multifunction_node`, который отправляет много сообщений для каждого полученного. Этот паттерн лучше выразить с помощью вложенного цикла `parallel_for`

На рис. 17.20 для каждого полученного сообщения узел multifunction_node генерирует много выходных сообщений, которые поступают узлу function_node с неограниченной степенью конкурентности. Этот граф ведет себя во многом как параллельный цикл, в котором multifunction_node играет роль управляющего цикла, а function_node – роль тела. Но для распределения работы потребуется часто заимствовать, как в Мастер-цикле на рис. 17.3 и 17.5. Хотя у этого паттерна и могут быть полезные применения, вероятно, более эффективно было бы использовать хорошо оптимизированный параллельный циклический алгоритм. Весь этот граф можно свернуть в один узел, содержащий, например, вложенный parallel_for. Конечно, возможность или желательность такой замены зависит от приложения.

Полезно: использовать узлы join_node, sequencer_node или multifunction_node для восстановления порядка в потоковом графе, когда это необходимо

Поскольку потоковый граф структурирован слабее, чем простой конвейер, иногда возникает необходимость установить определенный порядок сообщений в некоторых точках графа. Для этого есть три стандартных подхода: join_node с совпадением ключей, sequencer_node и multifunction_node.

Например, в главе 3 параллелизм в потоковом графе для создания стереоскопического эффекта допускал поступление левого и правого изображений в узел mergeImageBuffersNode не по порядку. Чтобы гарантировать, что на входах mergeImageBuffersNode будет правильная пара изображений, мы воспользовались узлом join_node с совпадением тегов. При наличии узла типа join_node данные в два входных порта могут поступать в разном порядке, но все равно будут соединяться правильно благодаря сравнению их тегов (или ключей). Дополнительные сведения о разных политиках соединения можно найти в приложении B.

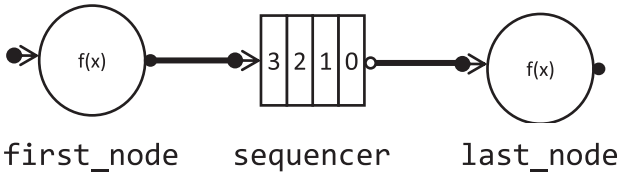
Другой способ установления порядка дает узел sequencer_node. Это буфер, который выводит сообщения по порядку, а для извлечения порядкового номера из входного сообщения используется написанный пользователем объект-тело.

На рис. 17.21 мы видим граф с тремя узлами: first_node, sequencer и last_node. Узел sequencer_node используется, чтобы восстановить порядок входных сообщений перед подачей конечному последовательному выходному узлу last_node. Поскольку узел first_node типа function_node неограничен, его задачи могут завершаться в любом порядке, поэтому генерируемые ими выходные сообщения могут следовать не по порядку. Узел sequencer_node восстанавливает порядок входных сообщений, ориентируясь на порядковый номер, присвоенный каждому сообщению в момент его конструирования.

Если выполнить аналогичный пример без sequencer_node с N = 10, то на выходе может наблюдаться произвольный порядок сообщений:

```
9 no sequencer
8 no sequencer
7 no sequencer
```

```
0 no sequencer
1 no sequencer
2 no sequencer
6 no sequencer
5 no sequencer
4 no sequencer
3 no sequencer
```



(a) Структура графа

```
struct Message {
  size_t my_seq_no;
  std::string my_string;
  Message(int i) : my_seq_no(i), my_string(std::to_string(i)) { }
};

using MessagePtr = std::shared_ptr<Message>;

void fig_17_21() {
  const int N = 10;
  tbb::flow::graph g;
  tbb::flow::function_node<MessagePtr, MessagePtr>
  first_node{g, tbb::flow::unlimited, [] (MessagePtr m) {
    m->my_string += " with sequencer";
    return m;
  }};
  tbb::flow::sequencer_node<MessagePtr>
  sequencer(g, [] (MessagePtr m) {
    return m->my_seq_no;
  });
  tbb::flow::function_node<MessagePtr, int, tbb::flow::rejecting>
  last_node{g, tbb::flow::serial, [] (MessagePtr m) {
      std::cout << m->my_string << std::endl;
      return 0;
  }};
  tbb::flow::make_edge(first_node, sequencer);
  tbb::flow::make_edge(sequencer, last_node);

  for (int i = 0; i < N; ++i)
    first_node.try_put(std::make_shared<Message>(i));
  g.wait_for_all();
}
```

(b) Реализация графа

Рис. 17.21 ❖ Узел типа `sequencer_node` используется, чтобы сообщения печатались в порядке, определяемом переменными-членами `my_seq_no`

При выполнении кода на рис. 17.21 результат будет таким:

```
0 with sequencer
1 with sequencer
2 with sequencer
3 with sequencer
4 with sequencer
5 with sequencer
6 with sequencer
7 with sequencer
8 with sequencer
9 with sequencer
```

Как видим, `sequencer_node` восстанавливает порядок сообщений, но требует, чтобы каждому сообщению был присвоен порядковый номер, а самому узлу было передано тело, умеющее извлекать этот номер из входного сообщения.

Последний подход к установлению порядка – использовать последовательный узел типа `multifunction_node`. Такой узел может для любого выходного сообщения выводить ноль или более сообщений в любой из своих выходных портов. Он не обязан выводить что-то для каждого входного сообщения, поэтому может буферизовать их и хранить до тех пор, пока не окажется выполнено определенное пользователем ограничение на порядок.

Например, на рис. 17.22 показано, как можно реализовать `sequencer_node` с помощью `multifunction_node`, если буферизовать входные сообщения, пока не поступит сообщение со следующим по порядку номером. В этом примере предполагается, что узлу `sequencer` посылается не более `N` сообщений с номерами от 0 до `N-1`. Создается вектор `v` с `N` элементами, инициализированными пустыми объектами `shared_ptr`. Когда сообщение поступает в узел `sequencer`, оно записывается в соответствующий ему элемент `v`. Затем отправляются все элементы `v` с последовательными номерами, начиная с номера последнего отправленного плюс 1, если таковые существуют, после чего номер последнего отправленного увеличивается. Для некоторых входных сообщений не будет отправлено ничего, для других будет отправлено сразу несколько сообщений (быть может, одно).

```
using MFNSequencer =
  tbb::flow::multifunction_node<MessagePtr,
std::tuple<MessagePtr>>;
using MFNPorts = typename MFNSequencer::output_ports_type;
int seq_i = 0;
std::vector<MessagePtr> v{N, MessagePtr{}};
MFNSequencer sequencer{g, tbb::flow::serial,
[&seq_i, &v](MessagePtr m, MFNPorts &p) {
  v[m->my_seq_no] = m;
  while (seq_i < N && v[seq_i].use_count()) {
    std::get<0>(p).try_put(v[seq_i++]);
  }
}};
```

Рис. 17.22 ❖ Для реализации `sequencer_node` используется `multifunction_node`

На рис. 17.22 показано, как multifunction_node можно использовать для переупорядочения сообщений по порядку, но в общем случае можно реализовать любой определенный пользователем порядок или пакетирование сообщений.

Полезно: использовать функцию isolate для вложенного параллелизма

В главе 12 мы говорили о том, что иногда при использовании TBB-алгоритмов возникает необходимость организовать изоляцию из соображений корректности или производительности. То же справедливо и для потоковых графов, а особенно когда имеет место вложенный параллелизм. На рис. 17.23 показан простой граф с узлами source и unlimited_node и вложенным параллелизмом внутри unlimited_node. Поток может шабашить (см. главу 12), ожидая завершения вложенного цикла parallel_for в узле unlimited_node, и выбрать другой экземпляр узла unlimited_node. Узел unlimited_node печатает «X started by Y», где X – номер экземпляра узла, а Y – идентификатор потока.

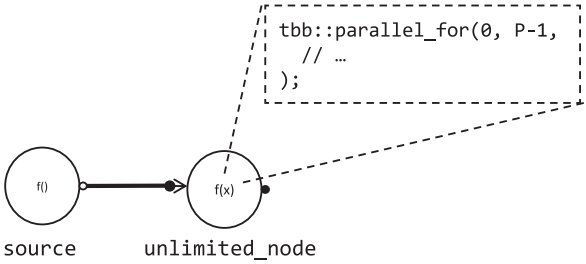
В нашей тестовой системе с 8 логическими ядрами распечатка показывает, что поток 0 так заскучал, что подобрал не один, а целых три разных экземпляра unlimited_node, ожидая, пока завершится его первый алгоритм parallel_for (рис. 17.24).

Как обсуждалось в главе 12, шабашка, вообще говоря, безвредна, и в данном случае это именно так, потому что ничего реального мы не вычисляем. Но как было отмечено в предыдущем обсуждении изоляции, такое поведение не всегда безобидно и может приводить как к некорректности, так и к падению производительности.

Мы можем разобраться с шабашкой в потоковом графе так же, как сделали это с задачами общего вида в главе 12, – с помощью функции this_task_arena::isolate или явных арен задач. Например, можно обращаться к parallel_for не напрямую в теле узла, а из вызова isolate:

```
tbb::this_task_arena::isolate([P,spin_time]() {
  tbb::parallel_for(0, P-1, [spin_time](int i) {
    spinWaitForAtLeast((i+1)*spin_time);
  });
});
```

После соответствующей модификации кода потоки перестают шабашить, и каждый поток занимается одним узлом до его полного завершения, как показано на рис. 17.25.



(a) Структура графа

```
void fig_17_23() {
  int P = tbb::task_scheduler_init::default_num_threads();
  tbb::concurrent_vector<std::string> trace;
  double spin_time = 1e-3;
  tbb::flow::graph g;

  int src_cnt = 0;
  tbb::flow::source_node<int> source{g,
    [&src_cnt, P, spin_time](int &i) -> bool {
       if (src_cnt < P) {
         i = src_cnt++;
         spinWaitForAtLeast(spin_time);
         return true;
       }
       return false;
    }, false};
  tbb::flow::function_node<int>
  unlimited_node(g, tbb::flow::unlimited,
    [&trace, P, spin_time](int i) {
      int tid = tbb::this_task_arena::current_thread_index();
      trace.push_back(std::to_string(i) + " started by "
                      + std::to_string(tid));
      tbb::parallel_for(0, P-1, [spin_time](int i) {
        spinWaitForAtLeast((i+1)*spin_time);
      });
      trace.push_back(std::to_string(i) + " completed by "
                      + std::to_string(tid));
    });

  tbb::flow::make_edge(source, unlimited_node);
  source.activate();
  g.wait_for_all();

  for (auto s : trace) std::cout << s << std::endl;
}
```

(b) Реализация графа

Рис. 17.23 ❖ Граф с вложенным параллелизмом

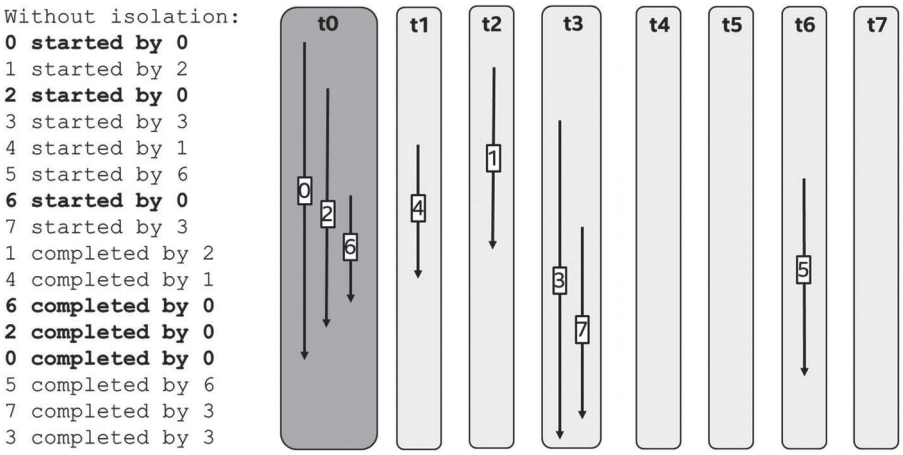


Рис. 17.24 ❖ Результат работы примера на рис. 17.23 показан слева, а диаграмма перекрывающегося выполнения – справа. Поток 0 участвует в выполнении трех разных узлов одновременно

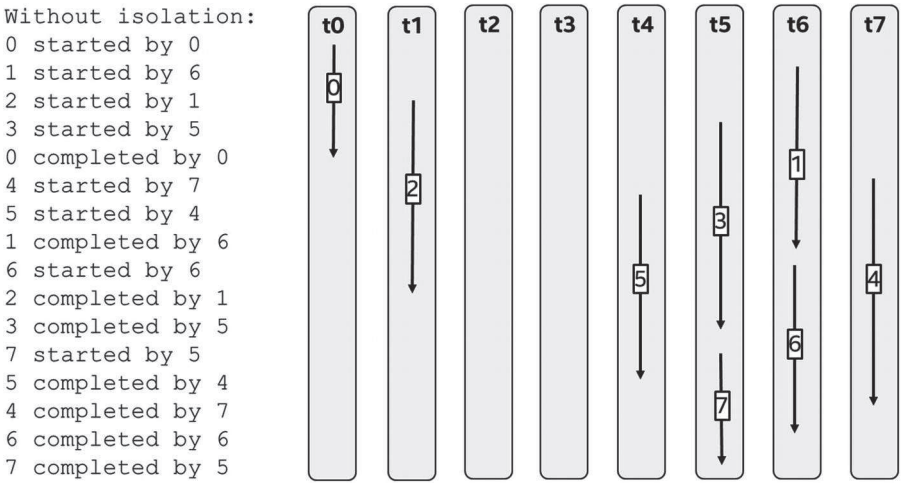


Рис. 17.25 ❖ Ни один поток не выполняет несколько узлов одновременно

Полезно: использовать отмену и обработку исключений в потоковых графах

В главе 15 мы обсуждали отмену задач и обработку исключений при использовании задач TBB в общем случае. Поскольку эта тема нам уже знакома, то здесь остановимся только на моментах, относящихся к потоковым графам.

В каждом потоковом графе есть единственный task_group_context

Экземпляр потокового графа запускает все свои задачи на одной арене задач и использует для всех них единственный объект `task_group_context`. При создании объекта графа конструктору можно явно передать объект `task_group_context`:

```
tbb::task_group_context tgc;
tbb::flow::graph g{tgc};
```

Если же объект не передан, то будет автоматически создан объект по умолчанию.

Отмена потокового графа

Чтобы отменить потоковый граф, нужно отменить его `task_group_context`, как и в обобщенных TBB-алгоритмах.

```
tgc.cancel_group_excution();
```

И, как и в TBB-алгоритмах, уже начатые задачи выполняются до конца, а новые не запускаются. В приложении B описана также вспомогательная функция-член класса `graph`, которая позволяет непосредственно опросить состояние графа:

```
if (g.is_cancelled()) {
  std::cout << "My graph was cancelled!" << std::endl;
}
```

Если мы хотим отменить граф, но не располагаем ссылкой на его объект `task_group_context`, то можем получить ее от задачи:

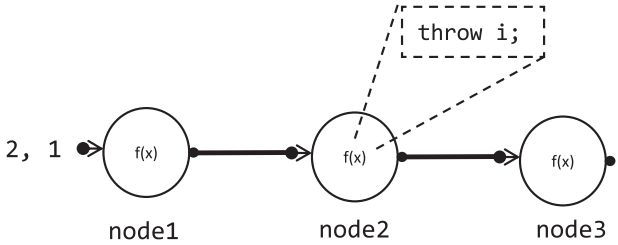
```
tbb::task::self().cancel_group_execution();
```

Сброс потокового графа после отмены

Если граф отменен, намеренно или в результате исключения, то его необходимо сбросить функцией `g.reset()`, прежде чем можно будет использовать снова. При этом заново инициализируется состояние графа: очищаются внутренние буферы, ребра приводятся в начальное состояние и т. д. Детали см. в приложении B.

Примеры обработки исключений

Чтобы понять, как работают исключения в потоковом графе, рассмотрим код на рис. 17.26. Здесь представлен небольшой граф с тремя узлами, который возбуждает исключение во втором узле, `node2`.



(a) Структура графа

```
void fig_17_26() {
  tbb::flow::graph g;

  tbb::flow::function_node<int, int> node1(g,
tbb::flow::serial,
    [](int i) { return i; }
  );
  tbb::flow::function_node<int, int> node2(g,
tbb::flow::serial,
    [](int i) {
      throw i;
      return i;
    }
  );
  tbb::flow::function_node<int, int> node3(g,
tbb::flow::serial,
    [](int i) { return i; }
  );
  tbb::flow::make_edge(node1,node2);
  tbb::flow::make_edge(node2,node3);
  node1.try_put(1);
  node2.try_put(2);
  g.wait_for_all();
}
```

(b) Реализация графа

Рис. 17.26 ❖ Потоковый граф, один из узлов которого возбуждает исключение

При выполнении этого примера мы получим исключение (надеемся, это не стало для вас сюрпризом):

```
terminate called after throwing an instance of 'int'
```

Поскольку мы не стали его обрабатывать, оно распространяется до самой внешней области видимости, и программа аварийно завершается. Конечно, мы можем изменить реализацию узла `node2`, обработав исключение в его теле, как показано на рис. 17.27.

```
tbb::flow::function_node<int, int> node2(g, tbb::flow::serial,
  [](int i) {
    try {
      throw i;
    } catch (int j) {
      std::cout << "Caught " << j << std::endl;
    }
    return i;
  }
);
```

Рис. 17.27 ❖ Потоковый граф,
один из узлов которого возбуждает исключение

После изменения пример доходит до конца и печатает сообщения «Caught» в случайном порядке:

```
Caught 2
Caught 1
```

Пока что ничего особо исключительного не видно (игра слов намеренная) – так всегда работают исключения.

Уникальная особенность обработки исключений, характерная только для потокового графа, состоит в том, что исключения можно перехватывать при обращении к функции графа `wait_for_all`, как показано на рис. 17.28.

```
try {
  g.wait_for_all();
} catch (int j) {
  std::cout << "Caught " << j << std::endl;
}
```

Рис. 17.28 ❖ Потоковый граф,
один из узлов которого возбуждает исключение

Если снова выполнить исходный пример на рис. 17.26, но окружить блоком try-catch вызов `wait_for_all`, то мы увидим только одно сообщение «Caught» (с цифрой 1 или 2):

```
Caught 2
```

Исключение, возбужденное в узле `node2`, не перехвачено в теле узла, поэтому распространяется до потока, который ждет завершения графа в функции `wait_for_all`. Если тело какого-то узла возбуждает исключение, то содержащий его граф отменяется. В данном случае мы не видим второго сообщения «Caught», потому что `node2` выполняется только один раз.

И конечно, если мы хотим повторно выполнить граф после обработки исключения, перехваченного в `wait_for_all`, то должны вызвать `g.reset()`, потому что граф был отменен.

Полезно: задавать приоритеты для графа, в котором используется task_group_context

Мы можем задать приоритеты для всех задач, запущенных графом, воспользовавшись объектом `task_group_context` этого графа, например:

```
if (auto t = g.root_task()) {
  t->group()->set_priority(tbb::priority_high);
}
```

А можно передать конструктору графа объект `task_group_context` с уже установленным приоритетом. В любом случае будут заданы приоритеты всех задач, связанных с графом. Можно создать один граф с высоким приоритетом, а другой с низким.

Незадолго до выхода этой книги из печати в TBB были добавлены относительные приоритеты функциональных узлов в качестве ознакомительной возможности. Это позволяет передать конструктору узла параметр, задающий его приоритет относительно других функциональных узлов. Этот интерфейс впервые появился в версии TBB 2019 Update 3. Интересующиеся читатели могут почерпнуть дополнительную информацию о нем в онлайновых заметках к версии и в документации.

Вредно: создавать ребро между узлами разных графов

Конструкторы всех узлов графа принимают ссылку на объект графа в качестве одного из аргументов. Вообще говоря, безопасно создавать ребра только между узлами одного и того же графа. Если ребром соединены узлы разных графов, то становится трудно рассуждать об их поведении, например: какие арены задач будут использоваться, сможет ли вызов `wait_for_all` корректно обнаружить завершение графа и т. д. Для оптимизации производительности библиотека TBB пользуется информацией о ребрах. Если два графа соединены ребром, то TBB будет беспрепятственно использовать его для целей оптимизации. Мы можем полагать, что создали два разных графа, но при наличии общих ребер TBB может выполнять их совместно неожиданными способами.

Для демонстрации такого неожиданного поведения мы написали класс `WhereAmIRunningBody`, показанный на рис. 17.29. Он печатает значения `max_concurrency` и `priority`, с помощью которых мы будем делать выводы о том, на какой арене задач и в каком контексте `task_group_context` выполняется задача, связанная с таким телом.

```
struct WhereAmIRunningBody {
  std::string node_name;
  WhereAmIRunningBody(const char *name) : node_name(name) {}

  int operator()(int i) {
    int P = tbb::this_task_arena::max_concurrency();
    std::string priority = "normal";

    if (tbb::task::self().group()->priority() == tbb::priority_high)
      priority = "high";

    std::cout << i << ":" << node_name
              << " executing in arena " << P
              << " with priority " << priority << std::endl;
    spinWaitForAtLeast(0.1);
    return i;
  }
};
```

Рис. 17.29 ❖ Класс тела, позволяющий делать выводы об арене задач и контексте `task_group_context`, которые используются при выполнении узла

В примере на рис. 17.30 класс `WhereAmIRunningBody` используется для демонстрации неожиданного поведения. Мы создали два узла: `g2_node` и `g4_node`. Конструктору узла `g2_node` передана ссылка на `g2`. Графу `g2` передана ссылка на `task_group_context` с приоритетом `priority_normal`, после чего для него вызвана функция `reset()` на арене задач со степенью конкурентности 2. Поэтому следует ожидать, что `g2_node` будет выполняться с обычным приоритетом на арене с двумя потоками, верно? Узел `g4_node` сконструирован так, что следует ожидать его выполнения с высоким приоритетом на арене с четырьмя потоками. Первая группа вызовов, в которую входят `g2_node.try_put(0)` и `g4_node.try_put(1)`, отвечает этим ожиданиям.

Но, проведя ребро из `g2_node` в `g4_node`, мы создали связь между узлами, принадлежащими разным графам. Второй набор вызовов, в который входит `g2_node.try_put(2)`, снова ведет к тому, что тело узла `g2_node` выполняется с обычным приоритетом на арене `a2`. Но при вызове `g4_node` TBB, стремясь сократить накладные расходы на планирование, применяет обход планировщика (см. главу 10), поскольку имеется ребро между `g2_node` и `g4_node`. В результате `g4_node` выполняется в том же потоке, что `g2_node`, но этот поток принадлежит арене `a2`, а не `a4`. Для него по-прежнему используется правильный контекст `task_group_context`, заданный при конструировании задач, но выполнение планируется не на той арене, которую мы ожидали.

```
2:g2_node executing in arena 2 with priority normal
2:g4_node executing in arena 2 with priority high
```

```
0:g2_node executing in arena 2 with priority normal
1:g4_node executing in arena 4 with priority high

void fig_17_30() {
  tbb::task_arena a2{2};
  tbb::task_group_context tcg2;
  tcg2.set_priority(tbb::priority_normal);
  tbb::flow::graph g2{tcg2};
  a2.execute([&]() {
    g2.reset();
  });

  tbb::task_arena a4{4};
  tbb::task_group_context tcg4;
  tcg4.set_priority(tbb::priority_high);
  tbb::flow::graph g4{tcg4};
  a4.execute([&]() {
    g4.reset();
  });

  tbb::flow::function_node<int, int>
  g2_node{g2, tbb::flow::serial, WhereAmIRunningBody("g2_node")};

  tbb::flow::function_node<int, int>
  g4_node{g4, tbb::flow::serial, WhereAmIRunningBody("g4_node")};

  g2_node.try_put(0);
  g2.wait_for_all();

  g4_node.try_put(1);
  g4.wait_for_all();

  tbb::flow::make_edge(g2_node,g4_node);
  g2_node.try_put(2);
  g2.wait_for_all();
  g4.wait_for_all();
}
```

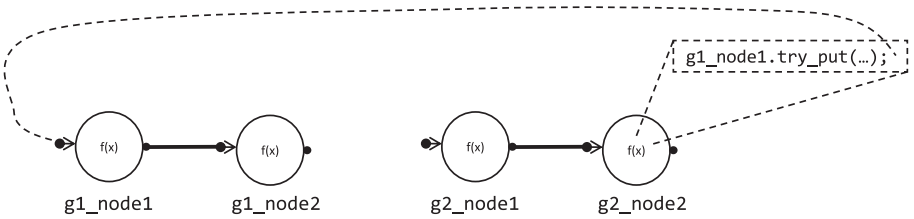
Рис. 17.30 ❖ Пример неожиданного поведения из-за перекрестного взаимодействия графов

Из этого простого примера видно, что ребро нарушает разделение графов. Если бы мы использовали арены `a2` и `a4` для управления количеством потоков, для изоляции работ или для привязки к потоку, то из-за наличия этого ребра все наши усилия пошли бы прахом. ***Не следует*** создавать ребер между разными графами.

Полезно: использовать try_put для передачи информации между графами

В предыдущем разделе мы пришли к выводу, что не следует создавать ребер между разными графами. Но что, если нам действительно нужно организовать

взаимодействие графов? Наименее опасный вариант – явно вызвать функцию try_put для отправки сообщения из узла одного графа в узел другого. Мы не создаем ребра, поэтому TBB не предпримет тайком никаких действий ради оптимизации взаимодействия между обоими узлами. Но даже в этом случае осмотрительность необходима, что демонстрирует пример на рис. 17.31.



(a) Структура графа

```
void fig_17_31() {
  tbb::flow::graph g1;
  tbb::flow::function_node<int, int> g1_node1{g1, tbb::flow::serial,
    [](int i) {
      std::cout << "g1_node1\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::function_node<int, int> g1_node2{g1, tbb::flow::serial,
    [](int i) {
      std::cout << "g1_node2\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::make_edge(g1_node1,g1_node2);

  tbb::flow::graph g2;
  tbb::flow::function_node<int, int> g2_node1{g2, tbb::flow::serial,
    [](int i) {
      std::cout << "g2_node1\n";
      spinWaitForAtLeast(i*0.1);
      return i;}};
  tbb::flow::function_node<int, int> g2_node2{g2, tbb::flow::serial,
    [&g1_node1](int i) {
      std::cout << "g2_node2\n";
      spinWaitForAtLeast(i*0.1);
      g1_node1.try_put(i);
      return i;}};
  tbb::flow::make_edge(g2_node1,g2_node2);

  g2_node1.try_put(1);
  g1.wait_for_all(); // возвращается сразу
  g2.wait_for_all(); // возвращается после g2_node1 и g2_node2
  std::cout << "At the end of the function\n";
  // мы пришли сюда до того, как g1 (запущенный g2) завершен
}
```

(b) Реализация графа

Рис. 17.31 ❖ Потоковый граф, посылающий сообщение другому потоковому графу

Здесь мы создаем граф g2, который отправляет сообщение графу g1, а затем ждет завершения g1 и g2. Только вот порядок ожидания неправилен!

Поскольку узел g2_node2 посылает сообщение узлу g1_node1, вызов g1.wait_for_all(), скорее всего, вернет управление сразу же, т. к. в g1 в момент вызова ничего не происходит. Затем мы вызываем функцию g2.wait_for_all(), которая вернет управление после завершения g2_node2. Как только этот вызов вернет управление, g2 уже завершен, но g1 только-только получил сообщение от g2_node2, и его узел g1_node1 только сейчас приступил к выполнению!

К счастью, если изменить порядок ожиданий, то все заработает, как надо:

```
g2.wait_for_all();
g1.wait_for_all();
```

Тем не менее мы видим, что явные вызовы try_put не всегда безопасны. Нужно очень внимательно относиться к взаимодействию графов между собой!

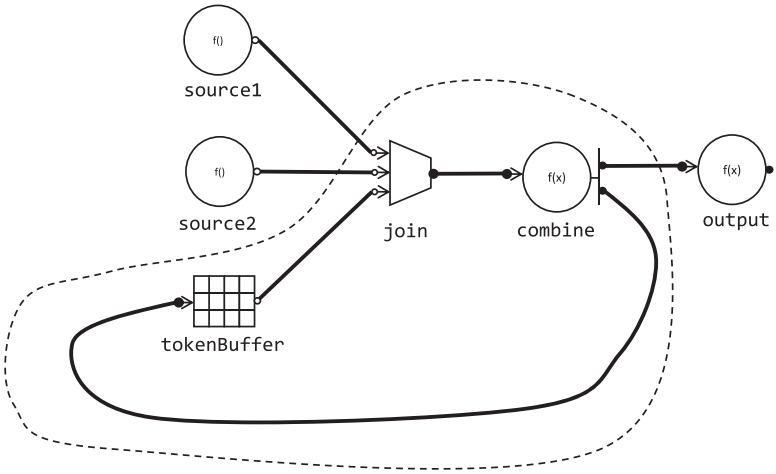
Полезно: использовать composite_node для инкапсуляции группы узлов

В двух предыдущих разделах мы предупредили, что взаимодействие между графами чревато ошибками. Часто более одного графа используют, потому что хотят логически отделить одни узлы от других. Инкапсулировать группу узлов удобно, если имеется общий паттерн, который приходится создавать многократно, или в одном большом плоском графе слишком много деталей.

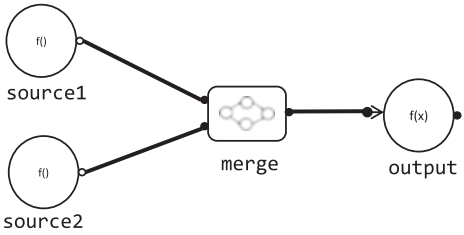
В обоих случаях можно использовать узел типа tbb::flow::composite_node, который позволяет инкапсулировать коллекцию узлов и использовать ее как полноправный узел графа. Ниже приведен интерфейс класса:

```
template< typename... InputTypes, typename... OutputTypes>
class composite_node <tbb::flow::tuple<InputTypes...>,
                      tbb::flow::tuple<OutputTypes...> > {
public:
  /* implementation defined */ input_ports_type;
  /* implementation defined */ output_ports_type;

  composite_node( graph &g );
  virtual ~composite_node();

  void set_external_ports(input_ports_type&& input_ports_tuple,
                          output_ports_type&& output_ports_tuple);
  input_ports_type& input_ports();
  output_ports_type& output_ports();
};
```

В отличие от других узлов, которые обсуждались в этой главе и в главе 3, чтобы воспользоваться этой функциональностью, мы должны создать новый класс, наследующий tbb::flow::composite_node. Например, рассмотрим потоковый граф на рис. 17.32(а). Он объединяет входные данные из источников source1 и source2, а для ограничения потребления памяти использует схему с передачей маркера.



(a) Граф с ненужными деталями на верхнем уровне



(b) Упрощенный граф с узлом `composite_node`

Рис. 17.32 ❖ Пример использования `composite_node`

Если такой паттерн передачи маркера часто используется в приложении или членами команды разработчиков, то имеет смысл инкапсулировать его, заведя отдельный тип узла, как показано на рис. 17.32(b). При этом сокрытие деталей также проясняет общую структуру приложения на верхнем уровне.

На рис. 17.33 показана реализация графа в случае, когда части, обведенные пунктирной линией, вынесены в один узел `merge`. Объект `merge` используется как любой другой узел потокового графа, его входные и выходные порты соединены ребрами с другими узлами. На рис. 17.34 показано, как узел `tbb::flow::composite_node` применяется для реализации класса `MergeNode`.

```
void fig_17_33() {
  tbb::flow::graph g;

  int src1_count = 0;
  tbb::flow::source_node<BigObjectPtr> source1{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src1_count < A_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src1_count++);
      return true;
    }
    return false;
  }, false};

  int src2_count = 0;
  tbb::flow::source_node<BigObjectPtr> source2{g,
  [&] (BigObjectPtr &m) -> bool {
    if (src2_count < A_LARGE_NUMBER) {
      m = std::make_shared<BigObject>(src2_count++);
      return true;
    }
    return false;
  }, false};

  MergeNode merge{g};

  tbb::flow::function_node<BigObjectPtr> output{g,
    tbb::flow::serial,
    [] (BigObjectPtr b) {
      std::cout << "Received id == " << b->getId()
                << " in final node" << std::endl;
    }
  };

  tbb::flow::make_edge(source1, tbb::flow::input_port<0>(merge));
  tbb::flow::make_edge(source2, tbb::flow::input_port<1>(merge));
  tbb::flow::make_edge(merge, output);

  bigObjectCount = 0;
  maxCount = 0;
  source1.activate();
  source2.activate();
  g.wait_for_all();
  std::cout << "maxCount == " << maxCount << std::endl;
}
```

Рис. 17.33 ❖ Создание потокового графа, в котором используется класс `MergeNode`, наследующий `tbb::flow::composite_node`

```
using BigObjectPtr = std::shared_ptr<BigObject>;
using CompositeType =
  tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                            std::tuple<BigObjectPtr>>;
using MFNode =
  tbb::flow::multifunction_node<std::tuple<BigObjectPtr,
                                           BigObjectPtr, token_t>,
                                std::tuple<BigObjectPtr, token_t>>;


class MergeNode : public CompositeType {
  tbb::flow::buffer_node<token_t> tokenBuffer;
  tbb::flow::join_node<std::tuple<BigObjectPtr, BigObjectPtr, token_t>,
                       tbb::flow::reserving> join;
  MFNode combine;

public:
  MergeNode(tbb::flow::graph &g) :
    CompositeType{g},
    tokenBuffer{g},
    join{g},
    combine{g, tbb::flow::unlimited,
      [] (const MFNode::input_type &in, MFNode::output_ports_type &p) {
        BigObjectPtr b0 = std::get<0>(in);
        BigObjectPtr b1 = std::get<1>(in);
        token_t t = std::get<2>(in);
        spinWaitForAtLeast(0.0001);
        b0->mergeIds(b0->getId(), b1->getId());
        std::get<0>(p).try_put(b0);
        std::get<1>(p).try_put(t);
      }}
  {
    tbb::flow::make_edge(tokenBuffer, tbb::flow::input_port<2>(join));
    tbb::flow::make_edge(join, combine);
    tbb::flow::make_edge(tbb::flow::output_port<1>(combine),
                         tokenBuffer);

    CompositeType::set_external_ports(
      CompositeType::input_ports_type(
        tbb::flow::input_port<0>(join),
        tbb::flow::input_port<1>(join)),
      CompositeType::output_ports_type(
        tbb::flow::output_port<0>(combine))
    );

    for (token_t i = 0; i < 3; ++i) tokenBuffer.try_put(i);
  }
};
```

Рис. 17.34 ❖ Реализация `MergeNode`

На рис. 17.34 класс MergeNode наследует типу CompositeType, псевдониму следующего типа:

```
tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                          std::tuple<BigObjectPtr>>;
```

Два аргумента шаблона означают, что MergeNode имеет два входных порта, получающих сообщения типа BigObjectPtr, и один выходной порт, отправляющий сообщения типа BigObjectPtr. В классе MergeNode имеется переменная-член для каждого инкапсулируемого узла: tokenBuffer, join и combine. Все переменные-члены инициализируются в списке инициализации конструктора MergeNode. В теле конструктора создаются внутренние ребра путем обращений к tbb::flow::make_edge. Вызов set_external_ports нужен для того, чтобы связать порты узлов-членов с внешними портами MergeNode. В данном случае первые два входных порта join становятся входами MergeNode, а выходной порт combine – выходом MergeNode. Наконец, поскольку этот узел реализует схему с передачей маркера, tokenBuffer заполняется маркерами.

Хотя поначалу создание типа, наследующего tbb::flow::composite_node, может напугать, использование этого интерфейса зачастую приводит к более понятному и допускающему повторное использование коду, особенно когда потоковые графы становятся все больше и сложнее.

Введение в Intel Advisor: Flow Graph Analyzer

Инструмент Flow Graph Analyzer (FGA) входит в состав Intel Parallel Studio XE 2019 и более поздние версии. Это часть инструмента Intel Advisor. Инструкции по скачиванию и установке выложены по адресу https://software.intel.com/en-us/articles/intel-advisor-xe-releasenotes.

FGA разрабатывался для поддержки проектирования, отладки, визуализации и анализа графов, созданных с помощью API потоковых графов в TBB. Однако многие возможности FGA полезны и для анализа общих графов вычислений вне зависимости от их происхождения. В настоящее время инструмент ограниченно поддерживается для других моделей параллельного программирования, включая OpenMP API.

В этой книге мы расскажем только о применении процессов проектирования и анализа к TBB. Мы также воспользуемся FGA для анализа некоторых примеров из этой главы. Однако все представленные выше оптимизации можно выполнить как с FGA, так и без него. Поэтому, если FGA вас не интересует, можете пропустить этот раздел. Впрочем, мы полагаем, что этот инструмент обладает несомненной ценностью, так что пропускать было бы ошибкой.

Процесс проектирования в FGA

В FGA мы можем графически спроектировать потоковый граф, проверить его правильность, оценить масштабируемость и, если довольны результатом, сгенерировать код на C++, в котором используются классы и функции из библиотеки TBB. FGA – не полная интегрированная среда разработки (IDE), как Mi-

crosoft Visual Studio, Eclipse или Xcode. Он позволяет начать проектирование потокового графа, но для завершения разработки придется выйти за пределы инструмента. Однако если в процессе проектирования соблюдать некоторые ограничения, как будет описано ниже, то итеративная разработка в конструкторе все же возможна.

На рис. 17.35 показан графический интерфейс FGA во время проектирования. Мы лишь вкратце остановимся на компонентах инструмента по мере описания типичного рабочего процесса, более полное описание имеется в документации по Flow Graph Analyzer.

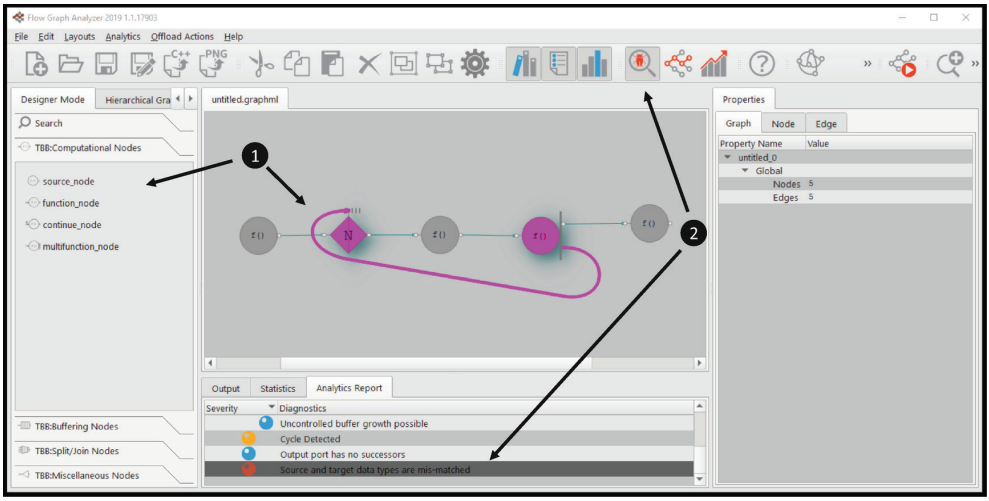


Рис. 17.35 ❖ Процесс проектирования в FGA

Как правило, проектирование начинается с пустого холста и проекта. Черный кружок с цифрой 1 на рис. 17.35 показывает, что мы выбираем узлы из палитры узлов, помещаем их на холст и соединяем, рисуя ребра между портами. Палитра содержит все типы узлов, имеющиеся в интерфейсе потокового графа в TBB, и с помощью всплывающих подсказок напоминает, для чего узел предназначен. Для каждого узла на холсте можно изменить его свойства, зависящие от типа; например, для узла типа `function_node` можно ввести код тела на C++, задать лимит конкурентности и т. д. Можно также задать оценку «веса», представляющую вычислительную сложность узла, чтобы впоследствии в процессе анализа масштабируемости посмотреть, достаточна ли производительность графа.

Нарисовав граф на холсте, мы можем выполнить проверку правил, т. е. проанализировать граф на предмет наличия типичных ошибок и антипаттернов. В результатах проверки правил, обозначенных кружком с цифрой 2 на рис. 17.35, отражены такие проблемы, как ненужная буферизация, несоответствие типов, подозрительные циклы в графах и т. д. На рис. 17.35 проверка правил обнаружила несовпадение типа входа узла `limiter_node` и типа выхода узла `multifunction_node`. В ответ мы можем, например, изменить тип выходного порта `multifunction_node`.

Исправив все ошибки, обнаруженные в ходе проверки правил, мы можем запустить анализ масштабируемости. В процессе этого анализа конструируется потоковый граф в памяти, и тела вычислительных узлов заменяются фиктивными телами, которые активно ожидают в течение времени, пропорционального «весу» узла. FGA прогоняет эту модель графа с разным количеством потоков и выводит таблицу ускорений, например:

Graph Name	Graph	Threads	Time(s)	Speedup
simple_g0	Results(4)			
	Scalability projection	1	4.00006	1
	Scalability projection	2	2.00003	2
	Scalability projection	3	2.00004	1.99999
	Scalability projection	4	1.01446	3.94305

Пользуясь этими средствами, мы можем итеративно уточнять проект графа. По ходу дела проект можно сохранять в формате GraphML (стандартный формат представления графов). Когда проект покажется нам удовлетворительным, мы сможем сгенерировать код на C++, выражающий наш проект в терминах интерфейса потокового графа TBB. Этот генератор кода правильнее было бы называть мастером, чем IDE, поскольку в нем нет прямой поддержки итеративной модели разработки кода. Не существует никакого способа импортировать изменения сгенерированного кода обратно в инструмент.

Советы по итеративной разработке с помощью FGA

Если мы хотим создать проект, который можно будет настраивать внутри FGA, то можем наложить на себя добровольные ограничения – писать тела узлов, перепоручая выполнение реализациям, сопровождаемым вне FGA. Это необходимо, потому что импортировать модифицированный код на C++ в FGA невозможно.

Например, чтобы упростить итеративную разработку, не следует размещать реализацию узла `function_node` прямо в его теле:

```
tbb::flow::function_node<int, std::string>
comp2str(g, tbb::flow::unlimited,
  [](int i) -> std::string {
    // полная реализация задана
    // в свойстве body узла в FGA
    int output_value = i*i;
    return std::to_string(output_value);
  }
);
```

Вместо этого следует специфицировать только интерфейс и перенаправить выполнение на реализацию, которую можно сопровождать отдельно:

```
tbb::flow::function_node<int, std::string>
comp2str(g, tbb::flow::unlimited,
  [](int i) -> std::string {
    // в свойстве body узла в FGA
    // задана перенаправленная реализация
    return comp2str_impl(i);
  }
);
```

Соблюдая такие самоограничения, мы сможем сопровождать проект графа и его представление на GraphML в FGA, т. е. итеративно изменять топологию и свойства узлов, не теряя изменения в теле узла, сделанные вне инструмента. Генерируя новую версию C++-кода с помощью FGA, мы просто включаем актуальный заголовок реализации, а тела узлов пользуются этой реализацией, сопровождаемой отдельно.

Конечно, Flow Graph Analyzer не заставляет нас действовать именно так, но это разумная практике в случае, когда мы хотим использовать FGA не просто как мастер генерации кода.

Процесс анализа в FGA

Процесс анализа в FGA не зависит от процесса проектирования. Конечно, мы можем проанализировать потоковый граф, спроектированный в FGA, но точно так же можно проанализировать потоковый граф, спроектированный и реализованный вне этого инструмента. Это возможно, потому что библиотека TBB оснащена средствами генерации событий, которые поступают сборщику трассы FGA во время выполнения. Трасса приложения TBB позволяет FGA реконструировать структуру графа и хронологию выполнения тел узлов – она ***не*** зависит от GraphML-файлов, созданных в процессе проектирования.

Чтобы использовать FGA для анализа приложения TBB, в котором имеется потоковый граф, нужно первым делом собрать трассу FGA. По умолчанию TBB не генерирует трассу, так что это действие необходимо явно активировать. Оснащение TBB средствами генерации трассы было ознакомительным средством до версии TBB 2019. Если вы пользуетесь более старой версией TBB, то придется выполнить дополнительные шаги. Отсылаем читателя к документации по FGA, где описано, как собирать трассы в разных версиях TBB и FGA.

При наличии трассы приложения процесс анализа в FGA включает действия, обозначенные пронумерованными кружками на рис. 17.36: (1) изучение представления производительности графа на древовидной карте, которая используется как индекс для отображения топологии графа; (2) выполнение алгоритма критического пути для нахождения критических путей в вычислении; (3) изучение хронологии и данные о конкурентности с целью лучше понять производительность в динамике. Анализ чаще всего является интерактивным процессом перехода между этими действиями по мере исследования производительности приложения.

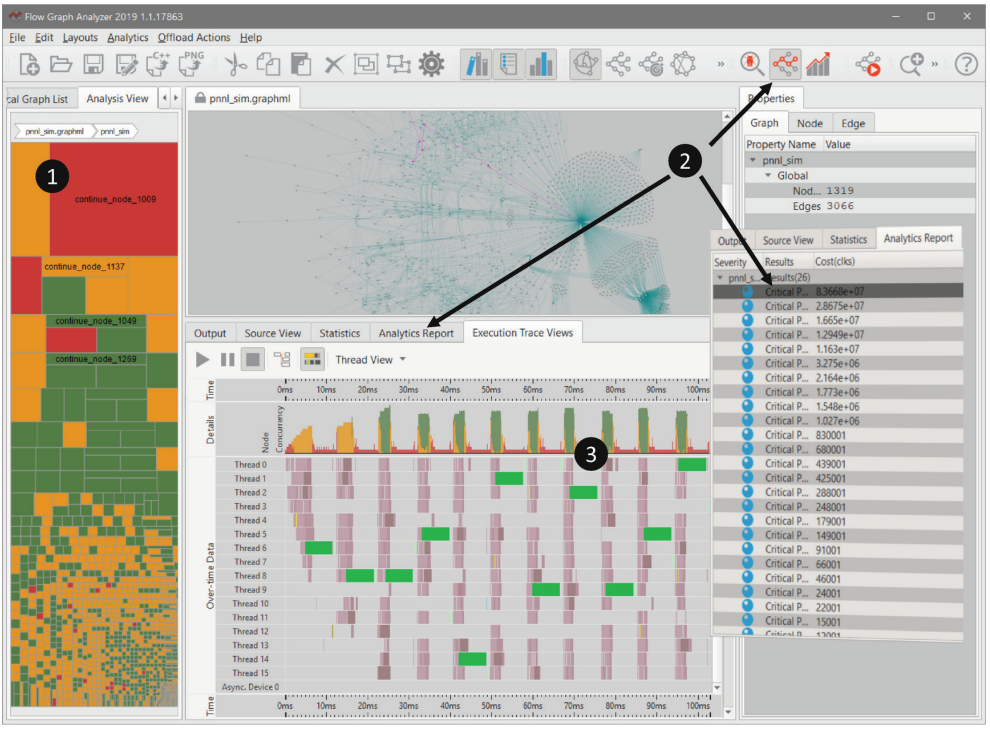


Рис. 17.36 ❖ Процесс анализа в FGA.
Результаты получены в системе с 16 ядрами

Представление в виде древовидной карты, обозначенное (1) на рис. 17.36, позволяет оценить общую работоспособность графа. Площадь каждого прямоугольника соответствует полному агрегированному времени ЦП, проведенному в узле, а его цвет показывает уровень конкурентности, наблюдавшийся при выполнении этого узла. Информация о конкурентности имеет четыре градации: плохая (красный), удовлетворительная (оранжевый), хорошая (зеленый) и перегружено (синий).

Узлы большой площади, помеченные как «плохие», – это горячие области, в которых средняя реальная конкурентность составляет от 0 % до 25 % доступной аппаратной конкурентности. Поэтому они являются первыми кандидатами на оптимизацию. Древовидная карта является также индексом в большой граф; щелчок по прямоугольнику подсвечивает узел в графе, а выбор подсвеченного узла, в свою очередь, помечает задачи из всех экземпляров этого узла в хронологическом представлении.

Холст с топологией графа синхронизирован с остальными представлениями. Выбор узла в древовидной карте, в хронологии или аналитическом отчете подсвечивает узел на холсте. Это позволяет быстро соотносить данные о производительности со структурой графа.

Одним из самых важных аналитических отчетов, предоставляемых FGA, является список критических путей в графе. Это особенно полезно, когда требуется проанализировать большой и сложный граф. Вычисление критических

путей дает список узлов на критических путях, показанный в области, помеченной кружком с цифрой 2 на рис. 17.36. В главе 3 мы говорили о том, что верхнюю границу ускорения для графа зависимостей легко вычислить, поделив полное агрегированное время, потраченное во всех узлах графа, на время, проведенное в самом длинном критическом пути, T_1/T_∞. Эту верхнюю границу можно использовать в качестве ожидаемого ускорения в приложении, выраженном в виде графа.

Представление хронологии и конкурентности, обозначенное кружком с цифрой 3 на рис. 17.36, показывает исходные трассы на «плавательных дорожках», соответствующих программным потокам. Пользуясь этой информацией, FGA вычисляет производные данные, в т. ч. среднюю конкурентность каждого узла и временную гистограмму конкурентности, для выполнения графа. Расположенная над плавательными дорожками, гистограмма показывает, сколько узлов активно в данный момент времени. Это позволяет быстро выявить промежутки времени с низкой конкурентностью. Щелчок по узлу в хронологическом представлении, попадающему в такие промежутки, позволяет разработчику найти в графе структуры, ведущие к таким узким местам.

Диагностика проблем производительности с помощью FGA

В этой главе мы обсудили ряд потенциальных проблем производительности, которые могут возникать при работе с потоковым графом. Сейчас мы кратко поговорим о том, как использовать FGA для изучения таких проблем в приложении, основанном на TBB.

Диагностика проблем зернистости

Как и в обобщенных циклических TBB-алгоритмах, мы должны помнить о задачах, которые слишком малы, чтобы получить выигрыш от распараллеливания. Но при этом нужно стремиться к тому, чтобы задач было достаточно для масштабирования рабочей нагрузки. В частности, в главе 3 отмечалось, что масштабируемость может быть ограничена последовательными узлами, если они становятся узким местом вычислений.

В примере хронологического представления в FGA на рис. 17.37 видно, что существует последовательная задача `m`, выделенная темным цветом, которая приводит к областям с низкой степенью конкурентности. Цвет говорит, что длительность задачи составляет примерно 1 мс – это выше порога эффективного планирования, но на хронологии задача выглядит как сериализованное бутылочное горлышко. Если возможно, мы должны разбить эту задачу на части, которые можно будет планировать параллельно, – либо на несколько независимых узлов, либо с помощью вложенного параллелизма.

С другой стороны, на рис. 17.37 имеются области, где более короткие задачи `n` выполняются параллельно. По цвету можно судить, что время их выполнения близко к порогу 1 мкс, поэтому мы видим в этой области просветы на хронологической шкале. Это означает, что накладными расходами на планирование нельзя пренебречь. В таком случае имеет смысл объединить узлы или, если возможно, задать облегченную политику выполнения, чтобы снизить непроизводительные затраты.

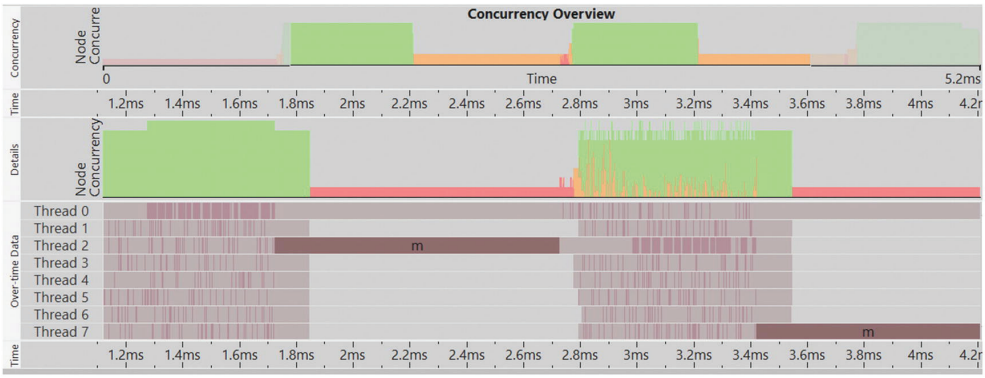
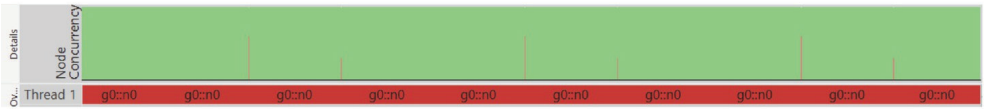


Рис. 17.37 ❖ В хронологическом представлении FGA цвет задачи обозначает время ее выполнения. Чем задача светлее, тем короче

Обнаружение медленного копирования

На рис. 17.38 показано, как можно обнаружить медленное копирование в FGA. Мы видим 100-миллисекундные участки на хронологии выполнения графа, изображенного на рис. 17.12, когда в сообщениях передаются сами объекты `BigObject` (рис. 17.38(a)) и указатели `shared_ptr<BigObject>` (рис. 17.38(b)). Чтобы конструирование действительно выглядело затратным, мы вставили в конструктор `BigObject` активное ожидание в течение 10 мс – тогда время конструирования `BigObject` и время выполнения тела узла `function_node` оказываются равны. На рис. 17.38(a) мы видим, что время копирования сообщения между узлами выглядит как просвет в хронологии. На рис. 17.38(b), где объекты передаются по указателю, время передачи сообщения пренебрежимо мало, так что никаких просветов не видно.

(a) Сообщениями являются объекты `BigObject`



(b) Сообщениями являются объекты `shared_ptr<BigObject>`

Рис. 17.38 ❖ Долгое копирование в FGA выглядит как просвет между выполнениями тела узла. Оба участка хронологии занимают приблизительно 100 мс

Когда FGA применяется для анализа приложений с потоковыми графами, просветы в хронологии свидетельствуют о неэффективности, которую нужно расследовать более тщательно. В этом разделе они были вызваны дорогостоя-

щим копированием сообщений, а в предыдущем – накладными расходами на планирование, сравнимыми со временем выполнения самой задачи. В обоих случаях наличие просветов подсказало нам, что нужно искать способы повысить производительность.

Диагностика шабашки

Выше в этой главе мы обсуждали выполнение графа с шабашниками на рис. 17.23 и его результат на рис. 17.24. FGA предлагает стековое представление хронологии выполнения, на котором легко обнаружить шабашников (рис. 17.39).

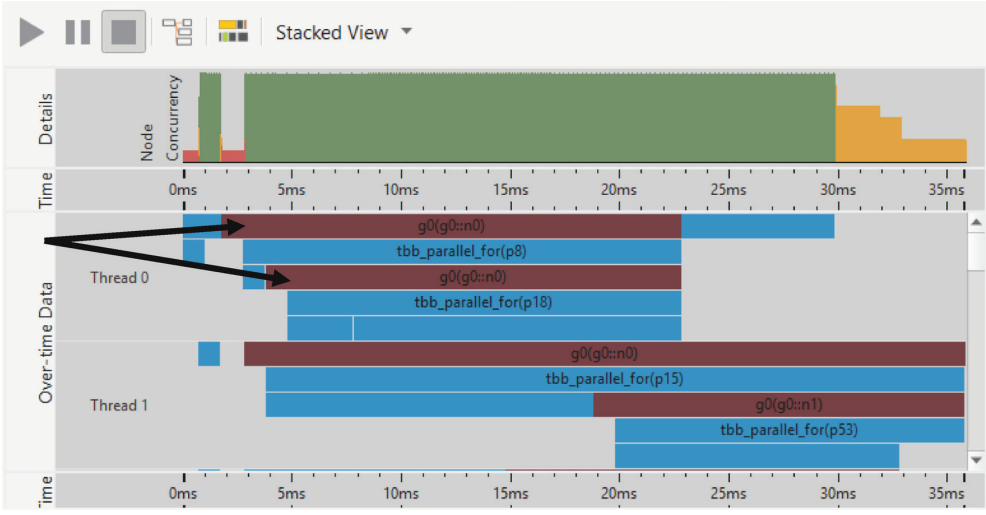


Рис. 17.39 ❖ Хронологии FGA, сгруппированные по узлу и региону. Мы видим, что поток 0 шабашничает, потому что конкурентно выполняет более одного параллельного региона

В стековом представлении видны все вложенные задачи, исполняемые потоком, в т. ч. те, что пришли из узлов потокового графа и из обобщенных параллельных TBB-алгоритмов. Поток, который выполняет два узла одновременно, – шабашник. Например, на рис. 17.39 поток 0 начинает исполнять другой экземпляр узла `n0` внутри уже имеющегося экземпляра `n0`. Из предыдущего обсуждения шабашки мы знаем, что это может произойти, если поток заимствует работу, ожидая завершения вложенного параллельного алгоритма. Стековое представление на рис. 17.39 позволяет легко понять, что в данном случае виновником является вложенный цикл `parallel_for` с меткой `p8`.

Применяя представления хронологии, предлагаемые FGA, мы можем обнаружить потоки-шабашники, просто заметив, что поток принимает участие в нескольких перекрывающихся регионах или узлах. Далее предстоит определить, является ли такая шабашка безобидной или ее нужно устранить, воспользовавшись средствами изоляции в TBB.

Резюме

API потокового графа – гибкий и мощный интерфейс для создания графов потоков данных и зависимостей. В этой главе мы обсудили некоторые дополнительные вопросы работы с высокоуровневым интерфейсом потоковых графов. Будучи реализован на основе задач TBB, он обладает всеми присущими им свойствами компонуемости и оптимизации. Мы рассмотрели, как эти средства можно использовать для оптимизации зернистости, эффективного использования кеша и памяти и организации достаточной степени параллелизма. Затем мы посоветовали, что стоит, а чего не стоит делать при работе с интерфейсами потокового графа. И наконец, мы дали краткий обзор инструмента Flow Graph Analyzer (FGA), входящего в состав Intel Parallel Studio XE, который поддерживает графическое проектирование и анализ потоковых графов в TBB.

Дополнительная информация

- *Michael Voss.* The Intel Threading Building Blocks Flow Graph // Dr. Dobb's. October 5, 2011. www.drdobbs.com/tools/the-intel-threading-building-blocksflow/231900177.
- *Vasanth Tovinkere, Pablo Reble, Farshad Akhbari, and Palanivel Guruvareddia.* Driving Code Performance with Intel Advisor's Flow Graph Analyzer // Parallel Universe Magazine. https://software.seek.intel.com/driving-code-performance.
- *Richard Friedman.* Intel Advisor's TBB Flow Graph Analyzer: Making Complex Layers of Parallelism More Manageable // Inside HPC. December 14, 2017. https://insidehpc.com/2017/12/intel-flow-graph-analyzer/.

Глава 18

Дополнение потоковых графов асинхронными узлами

Еще в 2005 году Герб Саттер написал статью «Время бесплатных завтраков кончилось»[1], в которой предупреждал о наступлении эры многоядерных процессоров и влиянии этого процесса на разработку программного обеспечения. Теперь разработчики, пекущиеся о производительности, уже не могут сидеть сложа руки и лениво ждать следующего поколения процессоров, а потом радостно наблюдать, как их приложения начинают работать быстрее. Эти дни безвозвратно миновали. Послание Герба заключается в том, разработчики, желающие извлечь максимум производительности из современных процессоров, должны освоить параллелизм. Дойдя до этого места, мы уже не сомневаемся в этом, и что с того? А то, что мы полагаем, что сегодня «завтраки сильно подорожали». Давайте разовьем эту тему.

Сравнительно недавно, под влиянием ограничений на энергопотребление, стали появляться более сложные процессоры. Сегодня нетрудно найти неоднородные системы, включающие один или несколько графических процессоров (GPU), программируемых пользователем вентильных матриц (ППВМ, англ. FPGA) или цифровых сигнальных процессоров (ЦСП, англ. DSP) наряду с одними или несколькими многоядерными центральными процессорами (ЦП, англ. CPU). И точно так же, как мы освоили параллелизм, чтобы выжать максимум из всех ядер ЦП, так теперь мы должны научиться передавать часть вычислений этим ускорителям. Да это же жутко сложно, скажете вы! Да, так и есть! Если последовательное программирование когда-то было «бесплатным завтраком», то параллельное программирование в гетерогенных средах стало, скорее, пиршеством в трехзведочном мишленовском ресторане – надо платить, но, боже, как вкусно!

А помогает ли TBB немного сэкономить на стоимости обеда? Разумеется! Да как вы могли сомневаться? В этой и следующей главе мы рассмотрим возможности, недавно включенные в библиотеку TBB с целью снова сделать завтраки

---

[1] *Herb Sutter*. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software // www.gotw.ca/publications/concurrency-ddj.htm.

доступными. Мы покажем, как передать вычисления асинхронным устройствам, и тем самым освоим гетерогенные вычисления. В этой главе мы усилим интерфейс потоковых графов в TBB новым типом узлов: `async_node`. А в следующей пойдем еще дальше и накачаем потоковые графы стероидами OpenCL.

Пример из асинхронного мира

Начнем с простейшего примера, в котором используется `async_node`. Мы покажем, почему этот узел полезен, а затем приведем более сложный пример, с которым будем работать в следующей главе.

Поскольку нет ничего проще программы «Hello World», мы предлагаем его аналог «Async World», основанный на API узла `async_node` в потоковом графе. Если у вас есть вопросы по поводу потоковых графов в TBB, прочитайте главу 3 для получения базовой информации, а затем используйте раздел «Потоковый граф» в приложении B как справочник. Потоковый граф, который мы построим в первом примере, изображен на рис. 18.1.

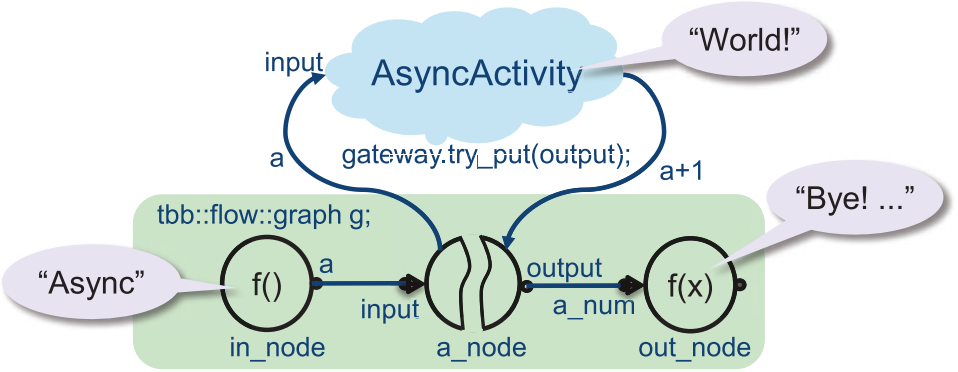


Рис. 18.1 ❖ Потоковый граф для примера «Async World»

Наша цель – отправить сообщение из узла `in_node` типа `source_node` асинхронному узлу `a_node`, но вместо того чтобы обрабатывать это сообщение в самом `a_node`, мы передадим его асинхронной операции, которая выполняется где-то в другом месте (в каком-то узле MPI, в графическом процессоре, поддерживающем OpenCL, в схеме ППВМ... ну, придумайте сами). По завершении асинхронной задачи потоковый граф снова получает управление, читает результат асинхронной операции и передает сообщение последующим узлам. В нашем очень простом примере «Async World» `in_node` просто печатает «Async» и передает значение `a=10` узлу `a_node`. Узел `a_node` получает `a=10` на входе и переправляет его операции `AsyncActivity`. В данном случае `AsyncActivity` – это класс, который увеличивает пришедшее в сообщении число на единицу и печатает «World!». Оба действия выполняются в новом потоке, который моделирует асинхронную операцию или устройство. И лишь когда `AsyncActivity` соблаговолит ответить, выдав `output=11`, узел `out_node` получит это значение, и программа завершится.

Код, представленный на рис. 18.2, содержит определение функции async_world(), в которой строится граф g с тремя узлами, показанный на рис. 18.1.

```
void async_world() {
  tbb::flow::graph g;
  bool n = false;

  //Исходный код
  tbb::flow::source_node<int> in_node{g,
    [&](int& a) {
      if (n) return false;
      std::cout << "Async ";
      a = 10;
      n = true;
      return true;
    },
    false
  };

  //Асинхронный узел
  AsyncActivity asyncAct;
  using activity_node_t = tbb::flow::async_node<int, int>;
  using gateway_t = activity_node_t::gateway_type;
  activity_node_t a_node{g, tbb::flow::unlimited,
    [&asyncAct](int const& input, gateway_t& gateway) {
      asyncAct.run(input, gateway);
    }
  };

  //Выходной узел
  tbb::flow::function_node<int> out_node{
    g, tbb::flow::unlimited,
    [](int const& a_num){
      std::cout << "Bye! Received: "<< a_num<< '\n';
    }
  };

  //Ребра:
  make_edge(in_node, a_node);
  make_edge(a_node, out_node);

  //Поехали!
  in_node.activate();
  g.wait_for_all();
}
```

Рис. 18.2 ❖ Построение потокового графа для примера «Async World»

Интерфейс узла типа source_node описан в первой строке таблицы на рис. В.37. В нашем примере создается узел in_node типа source_node. Аргументом лямбда-выражения, int& a, является выходное сообщение, которое будет отправлено его узлу-преемнику в графе (async_node). Когда узел source_node ак-

тивируется в конце функции async_world() обращением к in_node.activate(), лямбда-выражение выполняется только один раз, поскольку возвращает true уже при первом вызове (первоначально n=false и устанавливается в true внутри лямбда-выражения, которое возвращает true, только если n=true). В ходе этого единственного вызова сообщение a=10 отправляется следующему узлу графа. Последний аргумент in_node равен false, поэтому данный узел создается в режиме «спячки» и просыпается только при вызове in_node.activate() (в противном случае узел начал бы отправлять сообщения сразу после проведения выходного ребра).

Далее идет определение узла типа async_node. Его интерфейс выглядит так:

```
template<typename InType, typename OutType>
async_node(graph& g, size_t concurrency, Body body);
```

В нашем примере конструируется узел a_node:

```
using activity_node_t = tbb::flow::async_node<int, int>;
using gateway_t = activity_node_t::gateway_type;
activity_node_t a_node{g, unlimited, []… /*lambda*/ };
```

и в результате в графе g создается узел async_node<int, int> со степенью конкурентности unlimited. Аргумент unlimited означает, что мы разрешаем библиотеке запускать задачу сразу по приходе сообщения вне зависимости от того, сколько задач уже запущено. Если мы хотим, чтобы конкурентных выполнений a_node было не больше 4, то вместо unlimited следует передать значение 4.

Параметры шаблона <int, int> означают, что в узел a_node входит сообщение типа int и покидает его тоже сообщение типа int. Лямбда-выражение в конструкторе a_node выглядит следующим образом:

```
[&asyncAct](int const& input, gateway_t& gateway) {
  asyncAct.run(input, gateway);
}
```

Оно захватывает по ссылке объект asyncAct типа AsyncActivity и объявляет функтор, который должен выполняться для каждого сообщения, попадающего в a_node. У этого функтора два аргумента, input и gateway, передаваемых по ссылке. Однако минуточку, разве мы несколькими строчками выше не написали, что параметры шаблона <int, int> означают, что код ожидает на входе целое число и выдает на выходе целое число? Разве прототип функтора не должен иметь вид (const int& input) -> int? Да, для обычного узла function_node так и было бы, но мы-то имеем дело с async_node со всеми его особенностями. Здесь мы получаем на входе const int& input, как и ожидалось, но еще и второй аргумент, gateway_t& gateway, который нужен, для того чтобы подать выход операции AsyncActivity обратно в граф. Мы дойдем до этого трюка, когда будем объяснять, как устроен класс AsyncActivity. А пока, чтобы закончить описание этого узла, скажем лишь, что его основное назначение – передать работу AsyncActivity с помощью вызова asyncAct.run(input, gateway).

Выходной узел, out_node имеет тип function_node и в этом примере сконфигурирован как конечный узел, который не отправляет никаких сообщений:

```
tbb::flow::function_node<int> out_node{g, tbb::flow::unlimited,
  [](int const& a_num){
    std::cout << "Bye! Received: "<< a_num<< '\n';
  }
};
```

Этот узел получает целое число, поступившее от AsyncActivity через шлюз gateway, и заканчивает работу, напечатав «Bye!» и значение этого числа.

В последних строках примера на рис. 18.2 мы видим два вызова функции make_edge для создания соединений, изображенных на рис. 18.1, после чего граф наконец пробуждается функцией in_node.activate() и сразу же начинает ждать завершения обработки всех сообщений благодаря обращению к g.wait_for_all().

Тут-то и вступает в игру класс AsyncActivity, который реализует асинхронные вычисления и показан на рис. 18.3.

```
class AsyncActivity {
public:
  ~AsyncActivity() {
    asyncThread.join();
  }

  using node_t = tbb::flow::async_node<int, int>;
  using gateway_t = node_t::gateway_type;

  void run(int input, gateway_t& gateway) {
    gateway.reserve_wait();
    asyncThread = std::thread{
      [&,input]() {
        std::cout << "World! Input: " << input << '\n';
        int output = input + 1;
        gateway.try_put(output);
        gateway.release_wait();
      }
    };
  }
private:
   std::thread asyncThread;
};
```

Рис. 18.3 ❖ Реализация асинхронной операции

Открытая функция-член run (которая вызывается в функторе узла a_node с помощью asyncAct.run) сначала вызывает gateway.reserve_wait(), чтобы уведомить потоковый граф о том, что работа передана внешней операции и должна быть учтена функцией g.wait_for_all() в конце async_world(). Затем запускается асинхронный поток для выполнения лямбда-выражения, которое захватывает gateway по ссылке и целое число input по значению. Очень важно, что input передается по значению, потому что иначе объект ссылки – переменная a в source_node – мог бы быть уничтожен, до того как поток прочитает его значение (если source_node завершится раньше, чем asyncThread успеет прочитать a).

Лямбда-выражение в конструкторе потока сначала печатает сообщение «World», а потом присваивает output=11 (точнее, input+1). Этот результат передается обратно в потоковый граф посредством вызова функции-члена gateway.try_put(output). Наконец, с помощью функции gateway.release_wait() мы информируем граф о том, что AsyncActivity все сделала и ждать ее больше не нужно.

Примечание. Не требуется вызывать функцию reserve_wait() для каждого входного сообщения, полученного внешней операцией. Нужно лишь, чтобы каждому вызову reserve_wait() соответствовал вызов release_wait(). Заметим, что wait_for_all() не вернет управление, пока существуют вызовы reserve_wait() без соответствующих release_wait().

Программа напечатает такие строчки:

```
Async World! Input: 10
Bye! Received: 11
```

где «Async» печатается узлом in_node, «World! Input: 10» – асинхронной задачей, а последняя строчка – узлом out_node.

Зачем и когда использовать async_node?

Наверное, некоторые читатели самодовольно ухмыляются и думают: «Да не нужен мне async_node, чтобы это реализовать». В самом деле, почему бы не положиться на старый добрый function_node?

Например, a_node можно было бы реализовать, как на рис. 18.4, где function_node принимает целое input и возвращает другое целое, output. Соответствующее лямбда-выражение запускает поток asyncThread, который печатает и генерирует значение output, а затем ждет завершения этого потока в вызове asyncThread.join(), после чего радостно возвращает output.

```
tbb::flow::function_node<int, int> a_node{g,
  tbb::flow::unlimited,
  [&](const int& input) -> int {
    int output;
    std::thread asyncThread{[&,input]{
      std::cout << "World! Input: "<< input << '\n';
      output = input + 1;
    }};
   asyncThread.join(); // в этом месте рабочий поток блокируется
   return output;
}};
```

ОПАСНОСТЬ ОПАСНОСТЬ

Рис. 18.4 ❖ Простейшая реализация, которая создает асинхронный поток и ждет его завершения. А кто-то видит знак ОПАСНОСТЬ?

Если вы не принадлежали к числу ухмыляющихся читателей, то что скажете теперь? И вправду, что плохого в этой, гораздо более простой реализации? По-

чему не поступить так, чтобы разгрузить процессор, передать вычисления GPU или ППВМ и дождаться, пока ускоритель сделает свое дело?

Чтобы ответить на эти вопросы, нужно вернуться к одному из основополагающих принципов TBB – требованию компонуемости. TBB – компонуемая библиотека, потому что ее производительность не страдает, если разработчик – по прихоти или по необходимости – включит вложенный параллелизм внутрь других параллельных паттернов, какова бы ни была глубина вложенности. Одна из причин компонуемости TBB – тот факт, что добавление вложенных уровней параллелизма не увеличивает количество рабочих потоков. Это, в свою очередь, позволяет избежать перегруженности ядер и сопутствующих затрат, которые могут пагубно отразиться на производительности. Чтобы выжать максимум из оборудования, TBB обычно конфигурируется так, что количество рабочих потоков совпадает с количеством логических ядер. Различные TBB-алгоритмы (не важно, вложены они или нет) только добавляют легковесные задачи пользовательского уровня, чтобы загрузить потоки работой и тем самым полностью задействовать ядра. Однако, как мы предупреждали в главе 5, вызов блокирующей функции внутри пользовательской задачи блокирует не только эту задачу, но и исполняющий ее рабочий поток, которым управляет ОС. Если такое печальное событие произойдет и рабочий поток окажется заблокированным, то соответствующее ему ядро будет простаивать. И значит, мы не используем потенциал оборудования в полной мере!

В нашем простом примере на рис. 18.4 поток `asyncThread` будет использовать простаивающее ядро, пока выполняется задача, неподконтрольная потоковому графу. А что плохого в том, чтобы «скинуть» работу ускорителю (GPU/ППВМ/ЦСП – выбирайте по вкусу!) и подождать, пока он закончит? Если задача TBB вызывает блокирующую функцию из библиотеки OpenCL, CUDA или Thrust (это лишь немногие из существующих), то рабочий поток, исполняющий эту задачу, неизбежно будет блокирован.

До того как среди узлов потокового графа появился `async_node`, возможный, хотя и неидеальный выход из положения заключался в том, чтобы увеличить нагрузку на систему, создав один дополнительный поток. Для этого (см. главу 11) обычно выполняется такой код:

```
int cores = tbb::task_scheduler_init::default_num_threads();
tbb::task_scheduler_init(cores+1);
```

Это решение по-прежнему пригодно, если в программе нет потокового графа и мы просто хотим передать работу ускорителю, скажем, из `parallel_invoke` или с одного из этапов конвейера `parallel_pipeline`. Правда, мы должны понимать, что дополнительный поток будет блокирован большую часть времени в ожидании ускорителя. Однако у этого обходного пути есть недостаток: в течение некоторых периодов времени система будет перегружена (до и после передачи работы или даже на то время, пока драйвер ускорителя решает заблокировать[1] поток).

---

[1] Когда поток передает вычислительное ядро GPU, используя блокирующий вызов, драйвер необязательно сразу блокирует вызывающий поток. Например, некоторые драйверы GPU оставляют поток в состоянии активного ожидания, чтобы быстрее отвечать на легкие ядра. Но в конечном итоге поток блокируется, чтобы избежать бессмысленного потребления ресурсов во время обработки тяжелых ядер.

Для решения этой проблемы на помощь приходит `async_node`. Когда задача, запущенная узлом `async_node` (обычно его лямбда-выражение), завершается, рабочий поток, отвечавший за нее, переключается на другие ожидающие задачи потокового графа. Таким образом, рабочий поток не дает простаивать процессорному ядру. Важно помнить, что перед тем как завершить задачу, `async_node` должен уведомить потоковый граф, что выполняется асинхронная операция (вызвав `gateway.reserve_wait()`), а непосредственно перед передачей результата потоковому графу (с помощью `try_put()`) мы должны известить граф о том, что асинхронная операция завершилась, вызвав `gateway.release_wait()`. Все еще ухмыляетесь? Тогда расскажите нам, почему.

Более реалистичный пример

Тройственная функция из хорошо известного теста производительности STREAM[1] – это простая операция с массивами, которая вычисляет выражение `C = A + α*B`, где `A`, `B` и `C` – одномерные массивы. Она очень похожа на функцию `saxpy` из библиотеки BLAS 1, которая реализует операцию `A=A+α*B`, но записывает результат в другой вектор. Эта операция наглядно представлена на рис. 18.5.

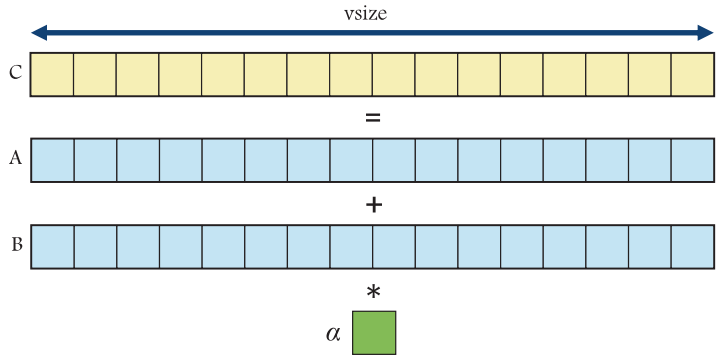


Рис. 18.5 ❖ Тройственная векторная операция, вычисляющая `C = A + α*B` ($c_i = a_i + \alpha * b_i, \forall i$)

Мы будем предполагать, что размеры массивов задаются переменной `vsize` и что во всех трех массивах хранятся числа с плавающей точкой одинарной точности. Дочитав до этого места, вы вряд ли испытаете трудности при распараллеливании этого естественно параллельного алгоритма. Давайте теперь обратимся к гетерогенной реализации.

У вас ведь есть интегрированный GPU, правда? Меня это не удивляет[2]! Согласно отчетам, более 95 % процессоров поставляются с интегрированным GPU, размещенным на той же плате, что многоядерный процессор. Неужели вы будете спать спокойно, запустив тройственный код всего на одном ядре ЦП?

---

1 *John McCalpin.* STREAM benchmark // www.cs.virginia.edu/stream/ref.html.

2 Песня Shania Twain – That Don't Impress Me Much. Альбом Come On Over, 1997.

Нет же, правда? Ядра ЦП не должны простаивать. И точно так же не должны простаивать ядра GPU. Во многих случаях мы можем задействовать потрясающие возможности GPU, чтобы дополнительно ускорить приложения.

На рис. 18.6 показано, как можно распределить вычисление тройственной функции между несколькими вычислительными устройствами.

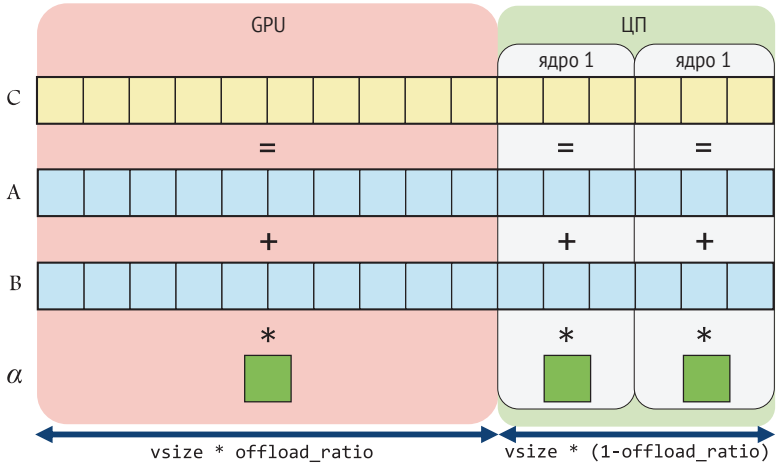


Рис. 18.6 ❖ Гетерогенная реализация вычисления тройственной функции

В нашей реализации переменная `offload_ratio` определяет, какая часть пространства итераций передается GPU. Остальные итерации параллельно выполняет ЦП. Понятно, что `0` ≤ `offload_ratio` ≤ `1`. Наш код будет основан на потоковом графе на рис. 18.7. Узел `in_node` типа `source_node` отправляет одну и ту же величину `offload_ratio` узлам `a_node` и `cpu_node`. Первый из них – узел типа `async_node`, который поручает вычисление соответствующего участка массивов процессору GPU с поддержкой OpenCL. Второй – обычный узел типа `function_node`, в который вложен алгоритм `parallel_for`, распределяющий участок массивов между имеющимися процессорными ядрами. Время выполнения на GPU, `Gtime`, и на ЦП, `Ctime`, возвращается соответствующим узлом, и обе величины преобразуются в кортеж в узле `join_node`. Наконец, в узле `out_node` эти времена распечатываются, а вычисленный гетерогенно массив `C` сравнивается с эталоном, полученным в результате простого последовательного выполнения цикла.

Примечание. Мы предпочитаем вводить новые идеи постепенно и старались следовать этому правилу, когда речь шла о самой библиотеке TBB. Но OpenCL – не предмет этой книги, поэтому мы вынуждены отступить от своих принципов и ограничиться лишь краткими комментариями по поводу конструкций OpenCL, встречающих в примерах ниже.

Для простоты в этом примере принимаются следующие предположения.

1) чтобы можно было применить стратегию буфера с нулевым копированием, которая уменьшает накладные расходы на перемещение данных между устройствами, будем предполагать, что установлен драйвер OpenCL 1.2 и что существует общая область памяти, видимая как ЦП, так

и GPU. Обычно так и обстоит дело в случае интегрированных GPU. В последних гетерогенных платах доступна версия OpenCL 2.0, и в таком случае мы будем пользоваться SVM (Shared Virtual Memory – разделяемая виртуальная память), как показано ниже;

2) чтобы уменьшить количество аргументов, передаваемых узлам потокового графа, и сделать код удобочитаемым, указатели на представления всех трех массивов, `A`, `B`, `C`, доступные ЦП и GPU, сделаны глобально видимыми. Переменная `vsize` также глобальная;
3) чтобы не останавливаться на вопросах, имеющих мало отношения к TBB, весь трафаретный код OpenCL инкапсулирован в одной функции `opencl_initialize()`. Эта функция получает сведения о платформе, выбирает устройство GPU `device`, создает контекст GPU `context` и очередь команд `queue`, читает исходный код ядра OpenCL, компилирует из него вычислительное ядро и инициализирует все три буфера, в которых хранятся GPU-представления массивов `A`, `B` и `C`. Поскольку классу `AsyncActivity` также нужна очередь команд и программные обработчики, соответствующие переменные `queue` и `program` тоже сделаны глобальными. Мы воспользовались C++-обертками OpenCL API, написанного на C. Точнее, мы взяли заголовок `cl2.hpp` с сайта https://github.com/KhronosGroup/OpenCL-CLHPP/.

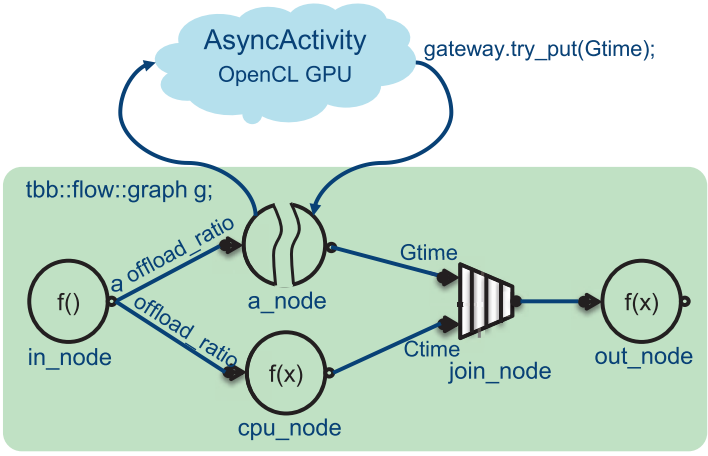


Рис. 18.7 ❖ Потоковый граф, реализующий гетерогенное вычисление тройственной функции

Начнем с функции `main`; на рис. 18.8 показано определение только двух первых узлов: `in_node` и `cpu_node`.

```
int main(int argc, const char* argv[]) {
  int nth = (argc>1) ? atoi(argv[1]): 4;
  vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  float ratio = (argc>3) ? atof(argv[3]) : 0.5;
  float alpha = 0.5;
  opencl_initialize(); // трафаретный код OpenCL
  tbb::task_scheduler_init init{nth};
  auto mp=tbb::global_control::max_allowed_parallelism;
  tbb::global_control gc(mp, nth+1); // еще один поток, спящий
  tbb::flow::graph g;

  bool n = false;
  tbb::flow::source_node<float> in_node{g,
    [&](float& offload_ratio) -> bool {
    if(n) return false;
    offload_ratio = ratio;
    n = true;
    return true;
  },false};
  tbb::flow::function_node<float, double> cpu_node{g,
    tbb::flow::unlimited,
    [&](float offload_ratio) -> double {
      auto t=tbb::tick_count::now();
      tbb::parallel_for(
        tbb::blocked_range<size_t>{
          static_cast<size_t>(ceil(vsize*offload_ratio)),
          static_cast<size_t>(vsize)
        },
        [&](const tbb::blocked_range<size_t>& r){
          for (size_t i = r.begin(); i < r.end(); ++i)
            Chost[i] = Ahost[i] + alpha * Bhost[i];
        }
      );
      return (tbb::tick_count::now() - t).seconds();
  }};
// … продолжение следует …
```

Рис. 18.8 ❖ Функция `main` программы гетерогенного вычисления тройственной функции; показаны первые два узла графа

Сначала мы читаем аргументы программы и инициализируем OpenCL, вызывая `opencl_initialize()`. Про эту функцию нам нужно только знать, что она инициализирует очередь команд GPU `queue` и OpenCL-программу `program`. Вопрос о начальном количестве потоков и о причинах инициализации объекта `global_control` будет рассмотрен в конце раздела. Исходный код вычислительного ядра GPU совсем прост:

```
kernel void triad(global float* A,
                  global float* B,
                  global float* C) {
  int i=get_global_id(0);
  float alpha = 0.5;
  C[i] = A[i] + alpha * B[i];
}
```

Здесь реализована тройственная операция `C = A + a*B` в предположении, что `a=0.5`, а массив чисел типа `float` хранится в глобальной памяти. В момент запуска ядра мы должны задать диапазон итераций, выполняемых на GPU, и внутренний планировщик GPU будет выбирать итерации из этого пространства командой `i=get_global_id(0)`. Для каждого такого `i` вычисление `C[i] = A[i] + alpha * B[i]` будет производиться параллельно различными вычислительными блоками GPU.

Внутри функции `opencl_initialize()` мы также выделяем память для трех буферов OpenCL и создаем соответствующие указатели, которые указывают на те же самые буферы со стороны ЦП (то, что называется представлением массивов на ЦП). В предположении, что имеется драйвер OpenCL 1.2, мы пользуемся конструктором `cl::Buffer`, который выделяет доступный GPU массив `Adevice`, соответствующий входному массиву `A`:

```
cl::Buffer Adevice;                    // Представление A на устройстве
Adevice = cl::Buffer{context, CL_MEM_READ_WRITE |
                     CL_MEM_ALLOC_HOST_PTR, sizeof(float)*vsize};
```

Флаг `CL_MEM_ALLOC_HOST_PTR` – ключ к буферу с нулевым копированием, поддерживаемому OpenCL, поскольку именно он говорит, что нужно выделить память, доступную хосту. Точно так же мы поступаем для GPU-представлений двух других массивов, `Bdevice` и `Cdevice`. Для получения указателей на представления этих буферов на ЦП служит функция OpenCL `enqueueMapBuffer`, которая используется следующим образом:

```
float* Ahost;                          // Хост-представление A
Ahost=(float*)queue.enqueueMapBuffer(Adevice, CL_TRUE, CL_MAP_WRITE |
CL_MAP_READ, 0, sizeof(float) * vsize, NULL, NULL, &err);
```

и возвращает указатель типа `float* Ahost`, с помощью которого можно читать и записывать ту же самую область памяти. Аналогичные обращения необходимы для получения указателей `Bhost` и `Chost`. В современных процессорах с интегрированным GPU этот вызов не влечет за собой копирования данных, потому-то стратегия и называется «буфер с нулевым копированием». Существуют и другие тонкости, связанные с OpenCL, например семантика и функциональность функции `clEnqueueUnmapMemObject()` и потенциальные проблемы, возникающие, когда ЦП и GPU пишут в разные участки одного массива, но они выходят за рамки этой книги.

Примечание. Если ваше устройство поддерживает OpenCL 2.0, то реализация упрощается, особенно если гетерогенный кристалл реализует так называемую SVM с мелкозернистым буфером. В этом случае можно выделить область памяти, которая не только видна ЦП и GPU, но также конкурентно обновляется и поддерживается в когерентном состоянии оборудованием. Чтобы узнать, поддерживается ли SVM с мелкозернистым буфером, нужно вызвать функцию `device.getInfo<CL_DEVICE_SVM_CAPABILITIES>()`.
Чтобы воспользоваться этим средством, мы можем в функции `opencl_initialize()` вызвать функцию `cl::SVMAllocator()`, передав ей в качестве аргумента шаблона `allocator` (распределитель памяти) конструктор `std::vector`. Это даст нам `std::vector A`, который одновременно является GPU- и ЦП-представлением данных:

```
using svmalloc_t = cl::SVMAllocator<float,
              cl::SVMTraitFine<cl::SVMTraitReadWrite<>>>;
svmalloc_t svmAlloc;
A=std::vector<float, svmalloc_t>{vsize, 0, svmAlloc};
```

`Ahost` и `Adevice` больше не нужны. Как всегда при работе с разделяемыми данными, ответственность за предотвращение гонки за данные лежит на нас. В нашем примере с этим все просто, потому что GPU и ЦП пишут в непересекающиеся участки массива `C`. Если же это условие не выполнено, то в некоторых случаях проблему удается решить с помощью массива атомарных переменных. Такое решение обычно называют платформенной или системной атомарностью, потому что элементы массива могут атомарно обновлять как ЦП, так и GPU. Эта возможность считается факультативной, и для ее использования необходимо создать `SVMAllocator` с характеристикой `cl::SVMTraitAtomic<>`.

Далее на рис. 18.8 идет объявление графа `g` и определение узла `in_node` типа `source_node`, похожее не то, что мы объясняли на рис. 18.2, но с тем отличием, что передается сообщение, содержащее значение `offload_ratio`.

Следующим в нашем примере является узел `cpu_node` типа `function_node`, который получает число типа `float` (на самом деле `offload_ratio`) и отправляет число типа `double` (время, потребовавшееся ЦП для вычислений). Внутри лямбда-выражения узла `cpu_node` вызывается `parallel_for`, и первым аргументом ему передается блочный диапазон типа:

```
tbb::blocked_range<size_t>{
                    static_cast<size_t>(ceil(vsize*offload_ratio)),
                    static_cast<size_t>(vsize)
                   }
```

означающий, что мы будем обходить только верхнюю часть массивов. Лямбда-выражение в этом алгоритме `parallel_for` параллельно вычисляет `Chost[i] = Ahost[i] + alpha * Bhost[i]` для различных участков пространства итераций, на которые автоматически разбивается диапазон.

На рис. 18.9, являющемся продолжением рис. 18.8, создается асинхронный узел `a_node`, который получает `float` (опять-таки значение `offload_ratio`) и отправляет время вычислений на GPU. Это делается асинхронно в лямбда-выражении `a_node`, где вызывается функция-член `run` объекта `asyncAct` типа `AsyncActivity` по аналогии с тем, что мы уже видели на рис. 18.2.

```
using async_node_t = tbb::flow::async_node<float, double>;
using gateway_t = async_node_t::gateway_type;
AsyncActivity asyncAct;
async_node_t a_node{g, tbb::flow::unlimited,
  [&asyncAct](const float& offload_ratio, gateway_t& gateway) {
    asyncAct.run(offload_ratio, gateway);
  }
};
using join_t = tbb::flow::join_node <std::tuple<double,double>,
                                     tbb::flow::queueing>;
join_t node_join{g};

tbb::flow::function_node<join_t::output_type> out_node{g,
  tbb::flow::unlimited,
  [&](const join_t::output_type& times){
    std::vector<float> CGold(vsize);
    std::transform(Ahost, Ahost + vsize, Bhost, CGold.begin(),
                   [&](float a, float b)->float{return a+alpha*b;});
    // сравнить эталон с результатом гетерогенного вычисления
    if (!std::equal(Chost, Chost+vsize, CGold.begin()))
       std::cout << "Error!!\n";
    std::cout<< "Time cpu: "<< std::get<1>(times)<< " sec."<< '\n';
    std::cout<< "Time gpu: "<< std::get<0>(times)<< " sec."<< '\n';
}};
// … продолжение следует …
```

Рис. 18.9 ❖ Функция `main` в программе гетерогенного вычисления тройственной функции, когда узлы графа определены, как показано выше

Тратить время на узел `join_node` мы не будем, т. к. этот тип уже был рассмотрен в главе 3. Достаточно сказать, что он передает следующему узлу кортеж, в который упаковано время вычислений на GPU и на ЦП.

Последним в графе является узел `out_node` типа `function_node`, который получает кортеж, содержащий оба времени выполнения. Прежде чем напечатать их, он проверяет, что массив `C` вычислен ЦП и GPU правильно. Для этого выделяется память для эталонного массива `CGold`, который затем последовательно вычисляется STL-алгоритмом `transform`. Если `Chost` совпадает с `CGold`, значит, все хорошо. Для реализации сравнения очень кстати придется STL-алгоритм `equal`.

На рис. 18.10 показано окончание функции `main()`, где создаются соединения между узлами, а затем с помощью активации узла `in_node` запускается выполнение графа. И в функции `g.wait_for_all()` мы ждем его завершения.

```
  tbb::flow::make_edge(in_node, a_node);
  tbb::flow::make_edge(in_node, cpu_node);
  tbb::flow::make_edge(a_node, tbb::flow::input_port<0>(node_join));
  tbb::flow::make_edge(cpu_node,tbb::flow::input_port<1>(node_join));
  tbb::flow::make_edge(node_join, out_node);

  auto gt = tbb::tick_count::now();
  in_node.activate();
  g.wait_for_all();
  return 0;
} // Конец main()!
```

Рис. 18.10 ❖ Последняя часть функции `main` программы вычисления тройственной функции – узлы соединяются ребрами, и граф запускается

Наконец, на рис. 18.11 приведена реализация класса `AsyncActivity`, функция-член которого `run` вызывается из `async_node`.

```
class AsyncActivity {
  tbb::task_arena a;
public:
  AsyncActivity() {
    a = tbb::task_arena{1,0};
  }
  using async_node_t = tbb::flow::async_node<float, double>;
  using gateway_t = async_node_t::gateway_type;
  void run(float offload_ratio, gateway_t& gateway){
    gateway.reserve_wait();
    a.enqueue([&,offload_ratio]()
      {
        auto t = tbb::tick_count::now();
        // Создать ядро тройственной функции, NDRange и запустить
        auto triad_kernel =
        cl::KernelFunctor<cl::Buffer&, cl::Buffer&, cl::Buffer&>
                          {program, "triad"};
        cl::EnqueueArgs q_args{
          queue,
          cl::NDRange{static_cast<size_t>(ceil(vsize*offload_ratio))}
        };
        triad_kernel(q_args, Adevice, Bdevice, Cdevice).wait();
        gateway.try_put((tbb::tick_count::now()-t).seconds());
        gateway.release_wait();
      });
    }
  };
```

Рис. 18.11 ❖ Реализация `AsyncActivity`, где и производится вызов вычислительного ядра на GPU

Вместо того чтобы запускать поток, как в классе AsyncActivity на рис. 18.3, мы пошли по другому, более замысловатому, но и более эффективному пути. Напомним, что мы отложили объяснение того, почему используется объект global_control на рис. 18.8. Там планировщик был инициализирован следующим образом:

```
tbb::task_scheduler_init init{nth};
auto mp=tbb::global_control::max_allowed_parallelism;
tbb::global_control gc(mp, nth+1); // еще один поток, спящий
```

Напомним (см. главу 11), что строка task_scheduler_init приводит к выполнению следующих действий:

- создается арена по умолчанию с nth слотами (один из них резервируется для мастер-потока);
- в глобальный пул потоков помещается nth - 1 рабочих потоков, которые будет занимать слоты на арене, как только на ней появится работа.

Но затем конструируется объект gc типа global_control, так что фактическое количество потоков в глобальном пуле увеличивается на единицу. Для этого дополнительного потока нет слота на арене по умолчанию, поэтому он засыпает.

Теперь класс AsyncActivity вместо запуска нового потока, как было раньше, будит спящий поток, что обычно требует меньше времени, особенно если AsyncActivity вызывается несколько раз. С этой целью конструктор класса инициализирует новую арену, a = tbb::task_arena{1,0}, на которой есть всего один слот для рабочего потока, потому что слоты для мастеров не резервируются. При вызове функции-члена run() новая задача ставится в очередь на этой арене путем обращения к a.enqueue(). Это приводит к активации спящего потока, который занимает слот на этой новой арене и доводит задачу до конца.

Затем задача, запущенная в этом объекте AsyncActivity, выполняет обычные действия, чтобы «скинуть» вычисления GPU. Во-первых, конструируется вычислительное ядро triad_kernel типа KernelFunctor с тремя аргументами cl::Buffer. Во-вторых, мы вызываем triad_kernel, передавая ему диапазон NDRange, вычисляемый как ceil(vsize*offload_ratio), и GPU-представления буферов Adevice, Bdevice и Cdevice.

При выполнении этого кода на процессоре Intel с интегрированным GPU печатаются такие строки:

```
Time cpu: 0.132203 sec.
Time gpu: 0.130705 sec.
```

Здесь vsize было равно 100 млн, и мы экспериментировали с offload_ratio, пока не нашли значение, при котором оба устройства вычисляли назначенные им участки массива примерно за одинаковое время.

Резюме

В этой главе мы познакомились с классом async_node, который дополняет возможности потокового графа для работы с асинхронными задачами, неподконтрольными графу. В первом простом примере «Async World» мы проиллюстри-

ровали применение этого класса и сопутствующего ему интерфейса `gateway`, который нужен для передачи сообщения от асинхронной задачи обратно потоковому графу. Затем мы обосновали необходимость такого расширения потоковых графов в TBB – это становится понятным, если принять во внимание, что блокировка задачи TBB приводит к блокировке рабочего потока. Узел типа `async_node` позволяет выполнять асинхронную работу вне потокового графа без блокировки рабочего потока TBB на время ожидания завершения асинхронной операции. И в заключение мы привели более реалистичный пример, в котором `async_node` используется, чтобы передать часть итераций цикла `parallel_for` на GPU. Мы надеемся, что все изложенное станет основой для разработки более сложных проектов с асинхронными компонентами. Ну а для тех, кто обычно ориентируется на GPU с поддержкой OpenCL, у нас есть хорошие новости: в следующей главе мы рассмотрим узел типа `opencl_node`, предлагающий более дружелюбный интерфейс для работы с GPU!

Дополнительная информация

Ниже приводятся дополнительные материалы по теме этой главы:

- *Herb Sutter*. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. URL: www.gotw.ca/publications/concurrency-ddj.htm.
- *John McCalpin*. STREAM benchmark. URL: www.cs.virginia.edu/stream/ref.html.
- *David Kaeli, Perhaad Mistri, Dana Schaa, Dong Ping Zhang*. Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann, 2015.

Глава 19

Накачанные потоковые графы: узлы OpenCL

Чувствуете ли вы, что `async_node` оставляет желать большего? Если да, эта глава для вас. В ней мы рассмотрим высокоуровневый класс потокового графа `opencl_node`, стремящийся скрыть аппаратные детали, а также нюансы программирования устройств с поддержкой OpenCL. Почему именно OpenCL? Причин много, назовем лишь некоторые: OpenCL – открытый стандарт, создаваемый усилиями членов большого консорциума. Он проектировался как платформенно независимый API и нацелен на гибкую эволюцию с появлением новых требований. Например, OpenCL первоначально был расширением языка C (а не C++), но в последней версии OpenCL 2.2 добавлена поддержка подмножества C++14, включая классы, лямбда-выражения, шаблоны и т. д.

Вам этого недостаточно? Ладно, тогда вот еще. Для нас причиной всех причин, гигантом на фоне карликов является количество и разнообразие платформ, на которых можно использовать OpenCL. В сегменте ноутбуков и настольных компьютеров более 95 % поставленных процессоров включают интегрированный GPU с поддержкой OpenCL (обычно производства Intel или AMD). В сегменте мобильных устройств сердцем большинства смартфонов и планшетов является система на кристалле (SoC), включающая GPU с поддержкой OpenCL (и да, в репозитории TBB есть двоичный код для Android тоже). Примеры сами по себе достаточно убедительны, но и это еще не все! На рынке встраиваемых устройств мы уже много лет можем купить и использовать гетерогенные платы с системами программирования на OpenCL для ППВМ (от компаний Intel-Altera и Xilinx). В сегменте серверов на момент написания этой книги Intel как раз начал продвигать центры обработки данных с PCI-платами ППВМ и процессором Intel Xeon Scalable Processor 6138P, который включает ППВМ на кристалле Intel-Altera Arria 10, и, конечно же, OpenCL является одной из поддерживаемых моделей программирования. Кроме того, OpenCL может работать на многих ЦП и других видах ускорителей, например Xeon Phi.

Но если OpenCL не отвечает вашим потребностям, то архитекторы TBB предусмотрели также возможность поддержки других моделей программирования.

Они абстрагировали низкоуровневые детали модели программирования ускорителей в модуль, названный *фабрикой*. На самом деле узел `opencl_node` – это результат конкретизации общего класса `streaming_node` конкретной фабрикой. Фабрика определяет необходимые методы для загрузки данных в ускоритель и выгрузки их оттуда, а также для запуска вычислительного ядра. Таким образом, `opencl_node` – плод брака между классом `streaming_node` и фабрикой OpenCL. Для поддержки новых моделей программирования нужно только разработать соответствующую фабрику.

Это довольно длинная глава, в которой рассматривается несколько понятий (`opencl_node`, `opencl_program`, `opencl_device`, аргументы и диапазон вычислительного ядра OpenCL, суббуферы и т. д.), поэтому кривая обучения будет крутой. Но мы начнем потихоньку, с равнины, и постепенно будем восходить к все более сложным классам и примерам (как всегда). Как показано на рис. 19.1, для начала мы разберем простой пример типа «Hello World», в котором используется узел `opencl_node`, а затем реализуем вычисление той же тройственной функции, что в предыдущей главе, но теперь уже с помощью новой высокоуровневой игрушки. Если хотите сохранить силы и не подниматься на вершину, то можете здесь и остановиться. А для опытных альпинистов в конце главы будет приведен краткий обзор более продвинутых средств, в т. ч. точная настройка диапазона OpenCL `NDRange` и спецификация ядра.

Рис. 19.1 ❖ Кривая обучения в этой главе

Пример «Hello OpenCL_Node»

На этот раз начнем с конца. Вот как выглядит результат нашего первого примера:

```
Hello OpenCL_Node
Bye! Received from: OPENCL_NODE
```

Эти две строки печатает потоковый граф, показанный на рис. 19.2, где выноски описывают, какая строка печатается каждым из трех узлов.

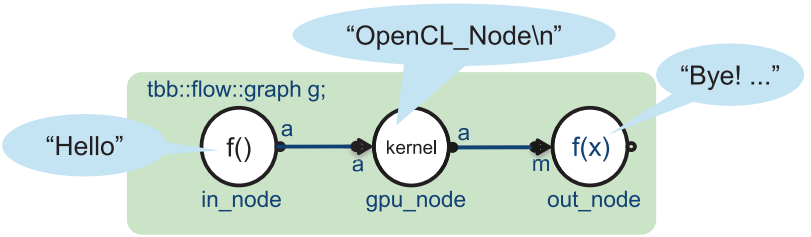


Рис. 19.2 ❖ Потоковый граф для примера «Hello OpenCL_Node»

Средний узел `gpu_node` типа `opencl_node` печатает строку «OpenCL_Node\n». Для этого ему предписано выполнять следующее ядро OpenCL, хранящееся в файле `hello.cl`:

```
kernel void cl_print(global char *str) {
  for ( ; *str; ++str ) {
    printf("%c", *str);
    if(*str>96 && *str<123) *str-=32; // a -> A, …
  }
}
```

Файл `hello.cl` содержит определение ядра `cl_print()`, которое будет выполняться специальным узлом потокового графа – типа `opencl_node`. Ядерная функция печатает массив символов, переданный во входном аргументе. Кроме того, чтобы заявить о своем присутствии, ядро также преобразует буквы нижнего регистра в верхний. Ключевое слово `global`, предшествующее объявлению аргумента `char *str`, означает, что массив символов должен храниться в глобальной памяти OpenCL. Сильно упрощая, можно сказать, что в данном случае строка хранится в области памяти, которую «как-то» могут читать и записывать как ЦП, так и GPU. В типичном случае интегрированного GPU глобальная память просто располагается в оперативной памяти. Это означает, что `opencl_node` должен получить в качестве аргумента массив символов. В нашем примере это массив «OpenCL_Node\n». Как вы, наверное, догадались, это сообщение приходит от первого узла, `in_node`. Да, указатель на строку (`a` на рис. 19.2) передается от `in_node` к `gpu_node`, и без всякого вмешательства пользователя строка, инициализированная на ЦП, оказывается в GPU. А какое сообщение достигает узла `out_node`? И на этот раз указатель `a`, который покидает `gpu_node` и прибывает в `out_node` под именем `m`. И вот наконец-то этот последний узел графа печатает «Bye! Received from: OPENCL_NODE», причем мы видим, во-первых, что строка изменилась, а во-вторых, что данные, обработанные на GPU, каким-то образом стали доступны ЦП. Теперь все мы жаждем увидеть детали реализации – и вот они, на рис. 19.3.

И всего-то! Отметим, что для конфигурирования узла GPU понадобилось всего три строки кода на C++. Ну, прелесть же, правда?

```
tbb::flow::graph g;
// Узел-источник:
bool n = false;
using buffer_t = tbb::flow::opencl_buffer<cl_char>;
tbb::flow::source_node<buffer_t> in_node{g, [&](buffer_t& a){
  if(n) return false;
  else {
    std::cout << "Hello ";
    char str[] = "OpenCL_Node\n";
    a = buffer_t{sizeof(str)};
    std::copy_n(str, sizeof(str), a.begin());
    n = true;
    return true;
  }
},false};

//Узел GPU:
tbb::flow::opencl_program<> program{std::string{"hello.cl"}};
tbb::flow::opencl_node<std::tuple<buffer_t>> gpu_node{g,
                                    program.get_kernel("cl_print")};
gpu_node.set_range({{1}});

//Выходной узел:
tbb::flow::function_node<buffer_t> out_node{g,
  tbb::flow::unlimited,
  [](buffer_t const& m){
    char *str = (char*) m.begin();
    std::cout << "Bye! Received from: " << str;
  }
};

//Создать ребра и запустить!
tbb::flow::make_edge(in_node, gpu_node);
tbb::flow::make_edge(gpu_node, out_node);
in_node.activate();
g.wait_for_all();
```

Рис. 19.3 ❖ Построение потокового графа для примера «Hello OpenCL_Node»

Ограничение ответственности. На момент написания этой главы последней была версия TBB 2019. В ней `opencl_node` все еще был ознакомительной возможностью, а это означает, что:

- реализация может измениться. Если вы используете ознакомительные средства в своем коде, то проверьте, по-прежнему ли они работают после перехода на новую версию TBB. В худшем случае ознакомительное средство может вообще исчезнуть!

- документация и поддержка могут быть ограничены. И действительно, документации по `opencl_node` и `streaming_node` в сети почти нет. Есть несколько статей в блоге[1] на эту тему, но им уже 3 года, и часть API с тех пор изменилась;
- его нужно включать явно (т. е. по умолчанию средство выключено). Чтобы можно было использовать `opencl_node` в программе, необходимо добавить следующие строки:

```
#define TBB_PREVIEW_FLOW_GRAPH_NODES 1
#define TBB_PREVIEW_FLOW_GRAPH_FEATURES 1
#include <tbb/flow_graph_opencl_node.h>
```

Преимущество этого заголовка, пусть и не очень существенное, в том, что необязательно включать `tbb/flow_graph.h` и заголовки OpenCL, потому что то и другое уже включено в `flow_graph_opencl_node.h`. На самом деле этот заголовочный файл наряду со статьями в блоге на данный момент является самым надежным источником информации о классах и функциях-членах, предоставляемых этим средством. Поэтому настоящую главу следует рассматривать как упрощенное введение в полторы тысячи строк кода, составляющих заголовок `opencl_node`.

А теперь рассмотрим узлы один за другим. Первый узел, `in_node`, уже знаком тем, кто изучал примеры в предыдущей главе. На всякий случай напомним, что (1) входным аргументом лямбда-выражения (`[&](buffer_t& a)`) является ссылка на сообщение, которое будет послано любому узлу, соединенному с данным; (2) узел `in_node` покидает только одно сообщение, потому что после первого вызова он возвращает `false`; (3) обращение к `in_node.activate()` пробуждает узел и активирует отправку этого единственного сообщения. Однако секундочку – появилось и кое-что новое, на что обязательно следует обратить внимание! Сообщение, покидающее `in_node`, должно оказаться в доступной GPU области памяти, поэтому аргумент `a` – не просто массив символов, а ссылка на `buffer_t`. Строка прямо перед определением `in_node` говорит, что `buffer_t` – это псевдоним шаблона `opencl_buffer`, конкретизированного типом символов OpenCL (`cl_char`):

```
using buffer_t = tbb::flow::opencl_buffer<cl_char>;
tbb::flow::source_node<buffer_t> in_node{g, [&](buffer_t& a){
…
```

Тип `opencl_buffer` – первый вспомогательный класс `opencl_node`, с которым мы встретились в этой главе, но далеко не последний. Это шаблонный класс, который абстрагирует строго типизированный линейный массив и инкапсулирует логику транзакций памяти между хостом и ускорителем. Мы выделяем память для `opencl_buffer<T>` с помощью конструктора класса, как в строке `a = buffer_t{sizeof(str)}`, или объявляя новый объект:

```
buffer_t a{sizeof(str)};
```

В обоих случаях создается буфер `opencl_buffer`, содержащий символы типа `cl_char`. Используемая нами версия фабрики OpenCL основана на OpenCL 1.2, в ней применяется стратегия буфера с нулевым копированием. Это означает, что конструктор `opencl_buffer` вызывает функцию OpenCL `clCreateBuffer`, пере-

---

[1] https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview.

давая в качестве одного из аргументов флаг CL_MEM_ALLOC_HOST_PTR. В предыдущей главе мы кратко упомянули, что буфер выделяется в пространстве GPU, но с помощью функции отображения (clEnqueueMapBuffer) можно получить указатель на память, доступную ЦП (ЦП-представление буфера). Чтобы вернуть управление буфером GPU, OpenCL предоставляет функцию обратного отображения (clEnqueueUnmapMemObject). В современных кристаллах с интегрированным GPU функции прямого и обратного отображений дешевы, потому что никакие данные не копируются, а в обязанности функций входит поддержание согласованности кешей CPU и GPU с копией, хранящейся в глобальной (оперативной) памяти. Для этого иногда требуется сбрасывать кеши CPU или GPU. Хорошая новость заключается в том, что нам до всей этой низкоуровневой возни нет никакого дела! Если будут разработаны новые фабрики с улучшенными возможностями или поддерживающие другие ускорители, то, чтобы ими воспользоваться, нам достаточно будет перекомпилировать свой исходный код. Допустим, что завтра Intel выкатит фабрику OpenCL 2.0, а наш ускоритель реализует SVM с мелкозернистым буфером. Тогда стоит нам воспользоваться фабрикой OpenCL 2.0 вместо 1.2, как мы совершенно бесплатно получим резкий прирост производительности (потому что операции прямого и обратного отображений становятся лишними, а когерентность кешей автоматически поддерживается оборудованием).

Простите, мы отвлеклись. Вернемся к делу. Мы объясняли, как работает узел типа source_node, в примере на рис. 19.3 (да-да, всего несколько абзацев назад). Этот узел, in_node, просто инициализирует массив символов str строкой "OpenCL_Node\n", выделяет память для буфера opencl_buffer a подходящего размера и копирует строку в этот буфер с помощью STL-алгоритма std::copy_n. Вот и все. Когда лямбда-выражение, переданное этому узлу source_node, завершится, сообщение со ссылкой на opencl_buffer отправится от узла in_node к узлу gpu_node.

Теперь вспомним строки, необходимые для конфигурирования gpu_node:

```
tbb::flow::opencl_program<> program{std::string{"hello.cl"}};
tbb::flow::opencl_node<std::tuple<buffer_t>> gpu_node{g,
                                    program.get_kernel("cl_print")};
gpu_node.set_range({{1}});
```

В первой строке используется второй из рассматриваемых в этой главе вспомогательных классов для opencl_node. Здесь создается объект program, конструктору которого передается имя файла hello.cl, содержащего вычислительное ядро OpenCL, cl_print. Существуют и другие конструкторы opencl_program на случай, если мы захотим передать заранее откомпилированное ядро или его SPIR-версию (представление на промежуточном языке OpenCL). Но чтобы не отвлекаться и сосредоточиться на примере, мы рассмотрим эти альтернативы позже.

Во второй строке создается узел gpu_node типа opencl_node<tuple<buffer_t>>. Это означает, что gpu_node получает сообщение типа buffer_t и по завершении отправляет сообщение того же типа buffer_t. А зачем заводить кортеж для одного аргумента (порта)? Дело в том, что opencl_node проектировался для приема нескольких сообщений от предшествующих узлов и отправки нескольких сообщений последующим узлам графа, и все эти сообщения собраны в кортеж. В настоящее время не существует частного случая интерфейса для одного вхо-

да и выхода, поэтому приходится создавать кортеж из одного элемента. Что касается соответствия между портами opencl_node и аргументами ядра, то по умолчанию opencl_node связывает первый входной порт с первым аргументом, второй порт со вторым аргументом и т. д. Другие возможности будут рассмотрены ниже.

А так ли нужно отправлять исходящее сообщение для каждого входящего? Тип opencl_node призван поддерживать максимальную связанность (один выходной порт на каждый входной), но если входов меньше, чем выходов, или наоборот, то соответствующие порты всегда можно оставить несвязанными. А правда ли, что вход и выход должны быть одного типа? Для текущей версии фабрики – да. Если входной порт 0 имеет тип T, то и выходной порт 0 будет иметь тип T (в кортеже, задающем типы аргументов, вход и выход не различаются).

Примечание. Главная причина таких проектных решений заключается в том, что каждый порт opencl_node потенциально может быть отображен на любой аргумент ядра OpenCL. Для аргумента вида «in-out» совпадение типов входного и выходного портов, конечно, имеет смысл. Для аргумента вида «out» все равно нужно передать объект, в который будет произведена запись, так что тип входа по-прежнему должен совпадать с типом выхода, иначе opencl_node должен был бы сам создавать объекты, чего он не делает. И наконец, для аргумента вида «in» наличие соответствующего ему выхода позволяет переправлять значение, т. е. передавать его без изменения последующим узлам. Поэтому самым практичным решением было сделать все аргументы вида «in-out». Мы полагаем, что это разумно, если рассматривать кортеж узла OpenCL как список аргументов и иметь возможность присоединять ребра к любому аргументу, чтобы установить или получить значение соответственно до или после выполнения. Для аргумента вида «in» соответствующее выходное значение не изменяется. Для аргумента вида «out» мы предоставляем место, в которое можно записать выходное значение и затем получить его. А аргументу вида «in-out» мы отправляем исходное значение, а потом получаем модифицированное.

Напомним, что узел OpenCL – ознакомительное средство. Разработчики TBB очень хотят получать отзывы о таких средствах – именно поэтому они вообще существуют. Они собирают все мнения – что хорошо, что плохо, – чтобы тратить время и силы на совершенствование именно тех частей библиотеки, которые пользуются максимальным спросом. Ознакомительный узел OpenCL сочтен достойным того, чтобы предложить его пользователям и получить обратную связь. Если у нас есть обоснованное мнение по поводу того, что добавить, мы должны высказать его и быть услышанными!

Конструктор класса opencl_node принимает в качестве аргументов объект потокового графа g и описатель ядерной функции, которая должна находиться в файле программы OpenCL. Поскольку файл hello.cl включает ядерную функцию cl_print, мы используем такую функцию-член: program.get_kernel("cl_print").

Это означает, что в одном исходном файле OpenCL может быть несколько ядерных функций, назначаемых разным узлам opencl_node. А должны ли мы ограничиваться только одним программным файлом? Необязательно. Мы можем создать столько объектов opencl_program, сколько пожелаем, если ядра разнесены по нескольким исходным файлам.

Наконец, в третьей строке мы видим вызов `gpu_node.set_range({{1}})`. Эта функция-член класса `opencl_node` определяет пространство итераций, которое будет обходить GPU. В терминологии OpenCL оно называется `NDRange`, но сейчас не станем заострять внимание на таких деталях. Пока что поверим на слово, что вызов `set_range({{1}})` приводит к тому, что тело ядра выполняется только один раз.

Итак, с узлами типа `source_node` (`in_node`) и `opencl_node` (`gpu_node`) мы разобрались, остался обычный узел `out_node` типа `function_node`. Ему соответствует такой код:

```
tbb::flow::function_node<buffer_t> out_node{g,
  tbb::flow::unlimited,
  [](buffer_t const& m){
    char *str = (char*) m.begin();
    std::cout << "Bye! Received from: " << str;
  }
};
```

Мы видим, что `out_node` получает сообщение `m` типа `buffer_t`. Поскольку на самом деле `buffer_t` – это псевдоним `opencl_buffer<cl_char>`, вызов `m.begin()` возвращает видимый ЦП указатель на строку, который был сначала инициализирован в `in_node`, а затем изменен ядром на GPU. Последний наш узел просто печатает эту строку и завершается.

Оставшаяся часть примера – обычная логика построения потокового графа: создание ребер между узлами, пробуждение узла-источника и ожидание завершения обработки всех сообщений (в данном случае только одного). Ничего нового.

Но прежде чем преодолевать первый серьезный подъем, еще раз повторим только что пройденное и постараемся лучше понять, что происходит с сообщением `a`, которое появилось на свет на ЦП, было отправлено GPU и там модифицировано, а затем передано последнему узлу, где мы смогли наблюдать эффект выполнения GPU-ядра. Надеемся, что в этом поможет рис. 19.4.

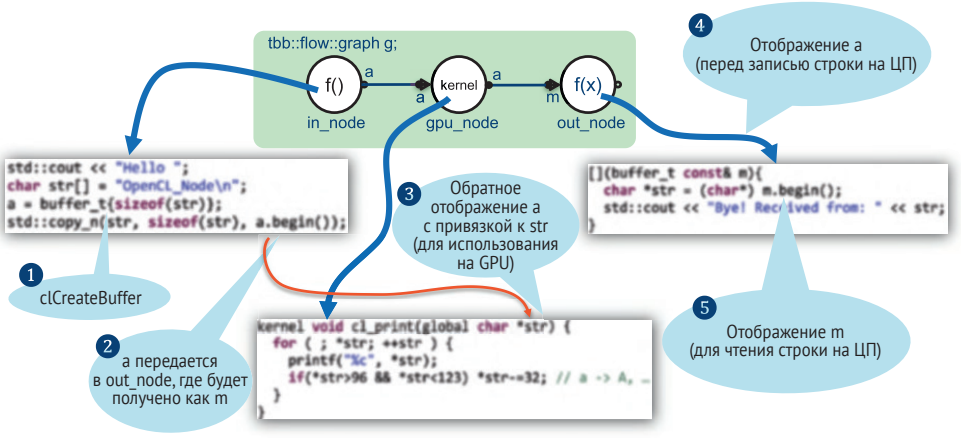


Рис. 19.4 ❖ Детальное описание операций над сообщением в примере

На этом рисунке предполагается, что фабрика OpenCL основана на версии 1.2 этого стандарта. В таком случае сообщение a создается как opencl_buffer в пространстве памяти GPU, но может быть также записано на ЦП, если сначала получить доступный ЦП итератор с помощью a.begin(). Ссылка на a – это сообщение, которое покидает узел in_node и приходит в порт 0 узла gpu_node (а это значит, что сообщение – ссылка на a может покинуть узел-родоначальник только через порт 0). Порт 0 узла gpu_node привязан к первому аргументу ядерной функции, имеющему совместимый тип (opencl_buffer<cl_char> можно привести к типу char *). Ядро может обращаться к строке, не опасаясь проблем с когерентностью кешей, потому что перед запуском ядра фабрика OpenCL озаботилась обратным отображением буфера. Наконец, ссылка на буфер достигает узла out_node, где строка снова подвергается отображению и печатается на ЦП.

Прежде чем идти дальше, мы хотим подчеркнуть, как здорово, что нам не приходится вручную писать весь трафаретный код OpenCL (определение платформы, устройства, контекст, очереди команд, чтение и компиляция ядра, задание аргументов ядра и его запуск, освобождение ресурсов OpenCL и т. д.). Все это скрыто под капотом благодаря фабрике OpenCL. Кроме того, как уже было сказано, после обновления фабрики наш код, возможно, будет работать быстрее или получит возможность работать с другими ускорителями, а от нас не потребуется вносить какие-либо модификации (ну разве что по мелочи).

Где исполняется наше ядро?

Пока все хорошо, верно? Но раз уж мы заговорили о трафаретном коде OpenCL, где же рычаг, который управляет тем, на каком устройстве исполняются наши узлы opencl_node? В предыдущем примере мы сказали, что gpu_node исполняет заданное вычислительное ядро на GPU. Ну а где же еще? А что, если мы соврали? Тревожно как-то, да? Ну что ж, посмотрим сначала, имеются ли на нашей машине еще какие-то устройства с поддержкой OpenCL. Будем надеяться, что такое устройство всего одно и это GPU, но руку на отсечение я бы не дал! Нужно все разузнать, но что-то не очень хочется писать старомодный код на чистом OpenCL. По счастью, фабрика OpenCL в TBB вручает нам еще два ценных вспомогательных класса (всего мы уже насчитали четыре). Это классы opencl_device и opencl_device_list. Для начала воспользуемся ими вне контекста потокового графа, как показано на рис. 19.5.

Сначала мы инициализируем объект devices типа opencl_device_list, вызвав функцию available_devices(). Эта функция возвращает итерируемый контейнер, содержащий все устройства с поддержкой OpenCL на первой платформе. Да, только на первой доступной платформе[1]. Затем вы извлекаем из списка первое устройство opencl_device d и запрашиваем у него имя платформы,

[1] Напомним, что это ознакомительное средство. Если вам такой гибкости недостаточно, то было бы правильно отправить компании Intel запрос со словами о том, что вы считаете идею узла OpenCL полезной, но хотелось бы снять имеющиеся ограничения.

профиль, версию и название поставщика. Эти атрибуты одинаковы для всех устройств на одной платформе.

```
const tbb::flow::opencl_device_list& devices =
        tbb::flow::interface10::opencl_info::available_devices();
tbb::flow::opencl_device d = *devices.cbegin();
std::cout << "Platform: "         << d.platform_name()    << '\n';
std::cout << "Platform profile: " << d.platform_profile() << '\n';
std::cout << "Platform version: " << d.platform_version() << '\n';
std::cout << "Platform vendor: "  << d.platform_vendor()  << '\n';
for (auto d : devices) {
  std::cout << "Device: "           << d.name() << '\n';
  std::cout << "  Major version: "  << d.major_version();
  std::cout << ";  Minor version: " << d.minor_version() << '\n';
  std::cout << "  Device type: ";
  switch (d.type()) {
    case CL_DEVICE_TYPE_GPU:
      std::cout << "GPU" << '\n';
      break;
    case CL_DEVICE_TYPE_CPU:
      std::cout << "CPU" << '\n';
      break;
    default:
      std::cout << "Unknown" << '\n';
  }
}
```

Рис. 19.5 ❖ Простой код для опроса платформы OpenCL и имеющихся устройств

Далее с помощью функции `for(opencl_device d:devices)` мы обходим весь список устройств и для каждого из них печатаем имя, основную и дополнительную версии и тип. Информация об основной и дополнительной версиях уже была получена от функции `d.platform_version()`, но она возвращала строку, тогда как вызовы `d.major_version()` и `d.minor_version()` возвращают целые числа. Результат выполнения этой программы на компьютере MacBook, на котором выполнялся и предыдущий пример, показан на рис. 19.6.

Примечание. Функция `available_devices()` на самом деле не является открытой, потому-то мы и вынуждены использовать длинную цепочку пространств имен:

`tbb::flow::interface10::opencl_info::available_devices()`.

Мы обратили внимание, что в исходном файле `flow_graph_opencl_node.h` прямо перед реализацией этой функции-члена находится комментарий

`// TODO: подумать о том, чтобы включить пространство имен opencl_info в открытый API`

Поскольку это ознакомительное средство, интерфейс еще не устаканился. Вспомните об этом, если приведенное выше «подумать» в итоге станет явью.

```
Platform: Apple
Platform profile: FULL_PROFILE
Platform version: OpenCL 1.2 (Feb 22 2019 20:16:07)
Platform vendor: Apple
Device: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
  Major version: 1;  Minor version: 2
  Device type: CPU
Device: Intel(R) HD Graphics 530
  Major version: 1;  Minor version: 2
  Device type: GPU
Device: AMD Radeon Pro 450 Compute Engine
  Major version: 1;  Minor version: 2
  Device type: GPU
```

Рис. 19.6 ❖ Результат выполнения программы рис. 19.5 на компьютере MacBook Pro

Может показаться удивительным, что на ноутбуке имеется аж три устройства, поддерживающих OpenCL! Одно из них – сам процессор Intel, а два других – GPU, первый интегрирован в Intel Core i7, а второй – дискретный AMD GPU. Напомним, что OpenCL – переносимый язык программирования, на котором можно писать код и для ЦП. И глядите-ка – первое устройство с поддержкой OpenCL – это не GPU, а 4-ядерный процессор Intel. Ну а теперь как вы считаете, где исполнялось ядро нашего первого примера? Вы правы, на первом устройстве. По умолчанию фабрика OpenCL выбирает первое доступное устройство, не важно, ЦП это или GPU. Итак… мы-таки солгали!!! Ядро исполнялось на ЦП, который замаскировался под ускоритель OpenCL. А что, если мы солгали не только здесь, но и в других местах книги? Подумайте об этом… это еще ужаснее (если только вы не начали чтение прямо с этой главы).

Ладно, исправим этот мелкий недочет. Чтобы спасти положение, у фабрики OpenCL есть два дополнительных средства: фильтр устройств и селектор устройств. Фильтры устройств позволяют на этапе инициализации `opencl_factory` задать множество устройств, на которых разрешено выполнять ядро. Все прошедшие фильтр устройства должны принадлежать одной и той же платформе OpenCL. По умолчанию используется класс фильтра `default_device_filter`, который автоматически находит все доступные устройства на первой платформе OpenCL и возвращает содержащий их список `opencl_device_list`. Со своей стороны селектор устройств, как следует из самого названия, отбирает одно устройство из этого списка `opencl_device_list`. Можно использовать разные селекторы для разных экземпляров `opencl_node`. Выбор производится при каждом выполнении ядра, поэтому может быть так, что узел `opencl_node` будет работать на разных устройствах при разных вызовах. Селектор по умолчанию `default_device_selector` отложенно выбирает и возвращает первое устройство из списка доступных, построенных фильтром устройств. Чтобы наш узел `gpu_node` исполнялся на настоящем GPU, нужно вместо строки

```
… gpu_node{g, program.get_kernel("cl_print")};
```

написать строку

```
… gpu_node{g, program.get_kernel("cl_print"), gpu_selector};
```

где `gpu_selector` – объект написанного нами класса `gpu_device_selector`:

```
gpu_device_selector gpu_selector;
```

а сам этот класс показан на рис. 19.7.

```
class gpu_device_selector{
public:
  template<typename DeviceFilter> tbb::flow::opencl_device
    operator()(tbb::flow::opencl_factory<DeviceFilter>& f) {
      auto it = std::find_if(
        f.devices().cbegin(), f.devices().cend(),
        [](const tbb::flow::opencl_device& d) {
          if (d.type() == CL_DEVICE_TYPE_GPU){
            std::cout << "Found GPU!" << '\n';
            return true;
          }
          return false;
        });

      if (it == f.devices().cend()){
        std::cout << "No GPU found!." << '\n';
        return *f.devices().cbegin(); //Вернуть первый
      }

      std::cout << "Running on "<< it->name() << '\n';
      return *it;
    }
};
```

Рис. 19.7 ❖ Наш первый селектор устройств

По соглашению (формально говоря, «концепции») третьим аргументом конструктора `opencl_node` является функтор (объект класса, в котором определена функция-член `operator()`), возвращающий устройство. Но в качестве тела функтора можно использовать встроенное лямбда-выражение. Функция `operator()` принимает ссылку на объект `f` класса `opencl_factory` и возвращает `opencl_device`. Применяя STL-алгоритм `find_if`, мы находим первый итератор `it` контейнера `devices()`, удовлетворяющий условию `it->type()==CL_DEVICE_TYPE_GPU`. Чтобы не писать лишнего, мы объявили его как `auto it` и поручили компилятору найти истинный тип, а именно

```
tbb::flow::opencl_device_list::const_iterator it = ...
```

На случай, когда GPU-устройства вообще нет, мы включили строку, которая возвращает самое первое устройство (хотя бы одно должно быть, какой же смысл в платформе без единого устройства?). Функтор завершает работу, печатая имя выбранного устройства и возвращая его. На нашем ноутбуке результат выглядел бы так:

```
Hello Found GPU!
Running OpenCL code on Interl(R) HD Graphics 530
OpenCL_Node
Bye! Received from: OPENCL_NODE
```

Заметим, что новые сообщения печатаются функтором селектора устройств узла `gpu_node`, который работает в момент активации этого узла. То есть сначала `in_node` печатает свое сообщение «Hello» и передает сообщение узлу `gpu_node`, который сначала ищет устройство (и печатает при этом слова, набранные полужирным шрифтом), а затем исполняет ядро. Вот на что стоит обратить внимание: узел `opencl_node` потокового графа обычно активируется несколько раз, поэтому лучше бы реализовать максимально простой селектор устройств.

Например, если лямбда-выражение в алгоритме `std::find_if` может не печатать сообщение «Found GPU!», то его можно еще упростить:

```
auto it = std::find_if(f.devices().cbegin(), f.devices().cend(),
  [](const tbb::flow::opencl_device& d) {
    return  d.type() == CL_DEVICE_TYPE_GPU;
  });
```

Если нам не нравится, что код явно содержит класс `gpu_device_selector`, то можем заменить функтор лямбда-выражением. Правда, это не совсем тривиально, потому что функция `operator()` в этом классе, как мы помним, шаблонная:

```
template<typename DeviceFilter> tbb::flow::opencl_device
        operator()(tbb::flow::opencl_factory<DeviceFilter>& f) {…}
```

Самый простой (из известных нам) способ реализации такого лямбда-выражения – воспользоваться полиморфными лямбдами, которые были введены в стандарте C++14. Не забудьте, что код на рис. 19.8 нужно компилировать с флагом `std=c++14`.

```
tbb::flow::opencl_node<std::tuple<buffer_t>> gpu_node{g,
  program.get_kernel("cl_print"),
  [] (auto& f) { //полиморфные лямбды из C++14
    std::cout << "Available devices:\n";
    int i = 0;
    std::for_each(f.devices().cbegin(), f.devices().cend(),
     [&](const tbb::flow::opencl_device& d) {
     std::cout << i++ << ".- Device: " << d.name() << std::endl;
    });
    tbb::flow::opencl_device d = *(++f.devices().cbegin()); //ВНИМАНИЕ!
    std::cout << "Running on " << d.name() << '\n';
    return d;
  }};
```

Рис. 19.8 ❖ Использование лямбда-выражения вместо функтора для выбора устройства

Обратите внимание на аргумент **`(auto& f)`** лямбда-выражения вместо `(opencl_factory<DeviceFilter>& f)`, который использовался в варианте с функтором.

Этот код обходит контейнер devices(), а затем возвращает второе устройство в списке, что приводит к такой распечатке:

```
Available devices:
0.- Device: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
1.- Device: Intel(R) HD Graphics 530
2.- Device: AMD Radeon Pro 450 Compute Engine
Running on Intel(R) HD Graphics 530
```

Теперь мы знаем свой список устройств, и в предположении, что хотим использовать интегрированный GPU, лучше изменить лямбда-выражение, сделав его быстрее:

```
tbb::flow::opencl_node<std::tuple<buffer_t>> gpu_node{g,
    program.get_kernel("cl_print"),
    [] (auto& f) { //полиморфные лямбды из C++14
         return *(++f.devices().cbegin()); //ВНИМАНИЕ!
    }};
```

Еще быстрее было бы запомнить opencl_device, возвращенное после первого обращения к селектору устройств. На рис. 19.9 приведен набросок модификации класса gpu_device_selector, представленного на рис. 19.7.

```
class gpu_device_selector {
  bool first_time = true;
  tbb::flow::opencl_device device;
public:
  template<typename DeviceFilter>
  tbb::flow::opencl_device operator()
   (tbb::flow::opencl_factory<DeviceFilter>& f) {
     if(first_time){
       device = *(++f.devices().cbegin());
       first_time = false;
     }
     return device;
   }
};
```

Рис. 19.9 ❖ Класс селектора устройств, который запоминает устройство opencl_device, найденное при первом обращении

Теперь в этом классе появилась переменная-член device типа opencl_device. При первом вызове operator() мы обходим список устройств f.devices() в поисках подходящего устройства (в данном случае оно будет вторым). После этого найденное устройство запоминается в переменной device для использования в будущем. Заметим, что нужно еще позаботиться о предотвращении гонки за данные, если этот оператор может одновременно вызываться из разных потоков.

Мы надеемся, что вы сохраните в тайне то, как плохо написан код примеров на рис. 19.8 и 19.9. Здесь мы «зашили» в код, что интересующее нас устрой-

ство второе в списке. Это так для нашей тестовой машины, но может оказаться совершенно иначе на других платформах. Более того, если в контейнере `f.devices()` имеется всего одно устройство, то разыменование `*(++f.devices().cbegin())` вообще приведет к ошибке сегментации. И это еще один пример компромисса между производительностью и переносимостью. Если мы не знаем, где будет выполняться наш код, а временем выбора устройства можно пренебречь по сравнению со временем вычислений, то лучше оставить вариант на рис. 19.7 (закомментировав печать).

Возвращаясь к более реалистичному примеру из главы 18

Вы еще не забыли вычисление тройственной функции векторов из предыдущей главы? Речь идет о простой операции над массивами вида `C = A + a*B`, где `A`, `B` и `C` – одномерные массивы, содержащие `vsize` чисел типа `float`, а `a` – скаляр, которому мы присвоили значение 0.5 (имеем право). Рисунок 19.10 – напоминание о том, как вычисление тройственной функции распределяется между GPU и ЦП в зависимости от переменной `offload_ratio`.

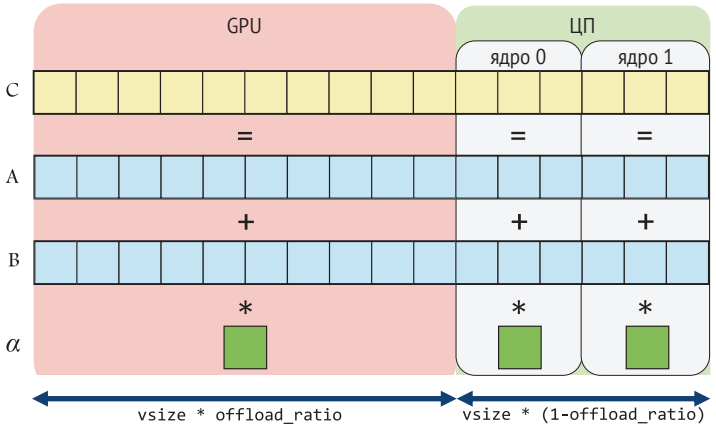


Рис. 19.10 ❖ Гетерогенная реализация тройственной функции

Повторную реализацию мы предприняли с двумя целями. Во-первых, еще раз посетив нашего старого знакомого, но теперь с точки зрения `opencl_node`, мы сможем лучше оценить преимущества этого высокоуровневого средства потоковых графов TBB. Во-вторых, выход за пределы примера «Hello OpenCL_Node» позволит глубже познакомиться с более продвинутыми применениями класса `opencl_node` и его вспомогательных классов. На рис. 19.11 показана схема потокового графа, который мы собираемся реализовать.

Как и в предыдущих примерах, узел типа `source_node` (`in_node`) просто запускает выполнение потокового графа, в данном случае передавая единственное сообщение, содержащее значение `offload_ratio`. Следом за ним идет узел типа

multifunction_node (dispatch_node). Отличительной особенностью этого вида узлов является гибкость при отправке сообщений последующим узлам графа. Мы видим, что dispatch_node имеет пять выходных портов: первые четыре связывают его с gpu_node, а последний – с cpu_node. Узел gpu_node имеет тип opencl_node и конфигурируется ядром вычисления тройственной функции на GPU, которое в качестве входных аргументов ожидает «GPU-представления» массивов A, B и C (как и в предыдущей главе, они называются Adevice, Bdevice и Cdevice). Однако у gpu_node имеется дополнительный порт для приема количества порученных ему итераций, который зависит от offload_ratio и который мы будем называть NDRange, придерживаясь терминологии OpenCL. Узел cpu_node – обычный функциональный узел, который получает «ЦП-представление» всех трех массивов и offload_ratio, чтобы ЦП мог выполнить свою часть сделки. У cpu_node имеется единственный входной порт, поэтому dispatch_node должен упаковать четыре переменные, необходимые ЦП, в кортеж. Оба узла, gpu_node и cpu_node, передают свое представление результирующего массива C узлу join_node, который строит содержащий их кортеж и передает его узлу out_node. Этот последний узел проверяет правильность вычисления и печатает время выполнения. А теперь давайте без лишних слов приступим к реализации, начав с определений типов данных и выделения буферов (рис. 19.12).

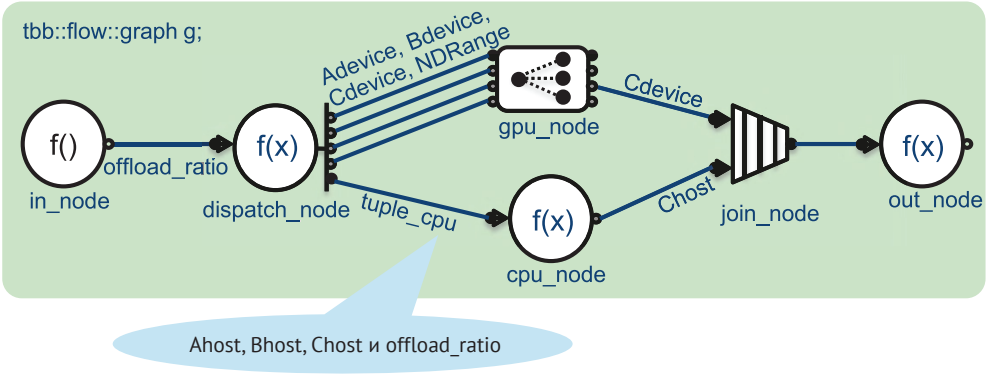


Рис. 19.11 ❖ Потоковый граф для реализации тройственной функции – теперь с использованием opencl_node

Начиная с этого момента buffer_f будет псевдонимом шаблона opencl_buffer, конкретизированного типом cl_float (аналог обычного типа float в OpenCL). Мы выделяем буферы Adevice, Bdevice и Cdevice этого типа под «GPU-представления» наших трех массивов. В классе opencl_buffer имеется также функция-член data(), с которой мы здесь встречаемся впервые. Эта функция возвращает доступный ЦП указатель на буфер GPU и берет на себя заботу об отображении этого буфера, чтобы ЦП мог обратиться к нему. Это позволяет инициализировать указатели Ahost, Bhost и Chost. Применяя STL-алгоритм generate, мы инициализируем массивы A и B случайными числами от 0 до 255, используя в качестве генератора вихрь Мерсенна (как в главе 5).

```
tbb::flow::graph g;

using buffer_f = tbb::flow::opencl_buffer<cl_float>;
buffer_f Adevice{vsize};
buffer_f Bdevice{vsize};
buffer_f Cdevice{vsize};
float* Ahost = Adevice.data();
float* Bhost = Bdevice.data();
float* Chost = Cdevice.data();

// Инициализировать генератор случайных чисел
std::random_device seed;     // Начальное значение генератора
std::mt19937 mte{seed()};    // Алгоритм вихря Мерсенна
std::uniform_int_distribution<> uniform{0, 256};
// Инициализировать A и B
std::generate(Ahost, Ahost+vsize, [&]{return uniform(mte);});
std::generate(Bhost, Bhost+vsize, [&]{return uniform(mte);});
```

Рис. 19.12 ❖ Определения типов данных и выделение буферов в примере вычисления тройственной функции

На рис. 19.13 определены первые два узла графа, `in_node` и `dispatch_node`.

В этой части алгоритма нет ничего сложного. Наш старый приятель `in_node` однократно отправляет `offload_ratio=0.5` узлу `dispatch_node`, имеющему такой тип:

```
multifunction_node<float,
            tuple<buffer_f, buffer_f, buffer_f, NDRange, tuple_cpu>>
```

Это означает, что он получает `float` (`offload_ratio`) и имеет пять выходных портов, через которые отправляются сообщения, чьи типы определены соответствующими элементами кортежа. Кортеж инкапсулирует типы данных всех пяти выходных портов данного многофункционального узла: три `buffer_f` (в данном случае `opencl_buffer`) для трех массивов, `NDRange` и `tuple_cpu`, в котором упакована вся информация для `cpu_node`.

Два входных аргумента лямбда-выражения, образующего тело `dispatch_node`, имеют вид:

```
[&](const float& offload_ratio, mfn_t::output_ports_type& ports ){…
```

Это входное сообщение (`offload_ratio`) и описатель (`ports`), который дает доступ к каждому из пяти выходных портов. Теперь воспользуемся функцией `get<port_number>(ports).try_put(message)`, чтобы отправить сообщение в порт с номером `port_number`. Четыре обращения к этой функции – все, что нужно для отправки информации, которую ждет GPU. Заметим, что в последнем вызове отправляется одномерный массив, содержащий всего один элемент `ceil(vsize*offload_ratio)`, который описывает пространство итераций на GPU. Единственное сообщение начинает путешествие на ЦП через последний порт с помощью вызова `get<4>(ports).try_put(cpu_vectors)`. Предварительно мы упа-

ковали ЦП-представление всех трех векторов и информацию об их разбиении (ceil(vsize*offload_ratio)) в кортеж cpu_vectors.

```
bool n = false;
tbb::flow::source_node<float> in_node{g, [&](float& offload_ratio) {
  if(n) return false;
  offload_ratio = 0.5;
  n = true;
  return true;
},false};

using NDRange = std::array<size_t, 1> ; //одномерный массив элементов типа size_t
using tuple_cpu = std::tuple<float*, float*, float*, size_t>;
using mfn_t = tbb::flow::multifunction_node<float,
    std::tuple<buffer_f, buffer_f, buffer_f, NDRange, tuple_cpu>>;
mfn_t dispatch_node{g,
  tbb::flow::unlimited,
  [&](const float& offload_ratio, mfn_t::output_ports_type& ports ){
    t0_p =tbb::tick_count::now();
    // Сообщения для GPU
    std::get<0>(ports).try_put(Adevice);
    std::get<1>(ports).try_put(Bdevice);
    std::get<2>(ports).try_put(Cdevice);
    std::get<3>(ports).try_put(
                 {static_cast<size_t>(ceil(vsize*offload_ratio))});

    tuple_cpu cpu_vectors;
    std::get<0>(cpu_vectors) = Ahost;
    std::get<1>(cpu_vectors) = Bhost;
    std::get<2>(cpu_vectors) = Chost;
    std::get<3>(cpu_vectors) = ceil(vsize*offload_ratio);
    // Сообщение для ЦП
    std::get<4>(ports).try_put(cpu_vectors);
  }};
```

Рис. 19.13 ❖ Первые два узла, in_node и dispatch_node, в примере вычисления тройственной функции

Нет вопросов? Уверены? Мы не хотим, чтобы что-то осталось непонятным. Нет? Ну ладно. Перейдем к реализации следующих двух узлов, где происходит самое главное – собственно вычисление. См. рис. 19.14.

Хотя cpu_node на рис. 19.14 объявлен вторым, мы рассмотрим его первым, потому что он требует меньше пояснений. Параметры шаблона <tuple_cpu, float*> означают, что узел получает кортеж tuple_cpu и отправляет указатель на float. Входной аргумент лямбда-выражения cpu_vectors используется в его теле, чтобы распаковать указатели на три вектора и переменную start (в нее записывается значение ceil(vsize*offload_ratio), уже вычисленную узлом dispatch_node). Располагая этой информацией, parallel_for производит вычисле-

ние тройственной функции в диапазоне blocked_range<size_t>(start, vsize), который соответствует второй части пространства итераций.

```
gpu_device_selector gpu_selector;
tbb::flow::opencl_program<> program{std::string{"triad.cl"}};
using tuple_gpu = std::tuple<buffer_f, buffer_f, buffer_f, NDRange>;
tbb::flow::opencl_node<tuple_gpu> gpu_node{g,
                                           program.get_kernel("triad"),
                                           gpu_selector};
gpu_node.set_args(tbb::flow::port_ref<0, 2>, alpha);
gpu_node.set_range(tbb::flow::port_ref<3>); //NDRange -> последний порт


tbb::flow::function_node<tuple_cpu, float *> cpu_node{g,
  tbb::flow::unlimited,
  [&](tuple_cpu cpu_vectors) {
    float* Ahost = std::get<0>(cpu_vectors);
    float* Bhost = std::get<1>(cpu_vectors);
    float* Chost = std::get<2>(cpu_vectors);
    size_t start = std::get<3>(cpu_vectors);
    // Параллельное выполнение на ЦП
    parallel_for(tbb::blocked_range<size_t>{start, vsize},
       [&](const tbb::blocked_range<size_t>& r){
         for (size_t i = r.begin(); i < r.end(); ++i)
           Chost[i] = Ahost[i] + 0.5 * Bhost[i];
       });
    return Chost;
}};
```

Рис. 19.14 ❖ Узлы, которые на своих плечах несут весь груз вычисления тройственной функции: gpu_node и cpu_node

Как уже было сказано, GPU отвечает за первую часть пространства итераций, которая в этом контексте определена как NDRange=[0, ceil(vsize*offload_ratio)]. Исходный код ядра GPU такой же, как в предыдущей главе, оно просто получает все три массива и вычисляет тройственную операцию для каждого i в диапазоне NDRange:

```
kernel void triad(global float* A, global float* B, global float* C){
  int i = get_global_id(0);
  C[i] = A[i] + 0.5 * B[i];
}
```

Эти строки ядра находятся в файле triad.cl, отсюда и строка

```
tbb::flow::opencl_program<> program{std::string{"triad.cl"}};
```

в начале рис. 19.14. Написанный нами тип tuple_gpu упаковывает все три буфера buffer_f и NDRange. Поэтому узел gpu_node объявлен так:

```
tbb::flow::opencl_node<tuple_gpu> gpu_node{g,
                                           program.get_kernel("triad"),
                                           gpu_selector};
```

Здесь из программного файла выбирается ядро "triad" и задается наш любимый селектор устройств gpu_selector.

Дальше идет интересная деталь конфигурации. В узел gpu_node приходят четыре сообщения, и ранее мы упомянули, что «opencl_node связывает первый входной порт с первым аргументом, второй порт со вторым аргументом и т. д.». Но секундочку! У ядра-то только три аргумента! Мы что, опять «соврамши»?! Нет, на этот раз нет. Мы ведь оговорились тогда, что это поведение по умолчанию и его можно изменить. И вот как.

Вызов gpu_node.set_args(port_ref<0,2>) говорит, что сообщения, приходящие в порты 0, 1 и 2, следует связывать с тремя входными аргументами ядра (A, B и C). А как быть с NDRange? В нашем первом примере «Hello OpenCL_Node» на рис. 19.3 встречалось только обращение gpu_node.set_range({{1}}), которое задает наименьший возможный диапазон NDRange с постоянным значением 1. Но в более сложном примере, рассматриваемом сейчас, NDRange переменный и приходит из узла dispatch_node. Мы можем связать третий порт узла, в который приходит NDRange, с функцией set_range(), что и сделали в строке gpu_node.set_range(port_ref<3>). Это означает, что мы можем передать set_range() постоянный или переменный NDRange, пришедший в порт. Функция-член set_args() должна обеспечивать такую же гибкость, верно? Мы знаем, как связать аргументы ядра с портами opencl_node, но зачастую аргументы ядра должны задаваться только один раз, а не при каждом вызове.

Предположим, к примеру, что наше ядро получает значение a, которое теперь является аргументом, заданным пользователем (а не «зашитым» значением 0.5, как раньше):

```
kernel void triad(global float* A, global float* B,
                  global float* C, float alpha){
  int i = get_global_id(0);
  C[i] = A[i] + alpha * B[i];
}
```

Тогда можно написать: gpu_node.set_args(port_ref<0,2>, 0.5f), т. е. первые три аргумента ядра связываются с данными, пришедшими в порты 0, 1, 2, а четвертый аргумент – с... 0.5 (о нет! только не надо зашивать снова! Да ладно, вообще-то никто не мешает нам передать переменную alpha, которой раньше было присвоено значение... 0.5).

Теперь перейдем к последним двум узлам, node_join и out_node, которые определены, как показано на рис. 19.15.

Как видно, узел node_join получает buffer_f (от gpu_node) и указатель на float (от cpu_node). Он и нужен-то только для объединения этих двух сообщений в кортеж, который переправляется следующему узлу. А следующим является узел out_node типа function_node, который получает сообщение типа join_t::output_type и никаких выходных сообщений не отправляет. Заметим, что join_t – тип узла node_join, поэтому join_t::output_type – псевдоним tuple<buffer_f, float*>. И действительно, входной аргумент лямбда-выражения, m, имеет такой тип. Для

распаковки кортежа m удобно воспользоваться конструкцией std::tie(Cdevice, Chost) = m, которая в точности эквивалентна двум предложениям:

```
Cdevice = std::get<0>(m);
Chost = std::get<1>(m);
```

```
using join_t = tbb::flow::join_node<std::tuple<buffer_f, float*>,
                                    tbb::flow::queueing>;
join_t  node_join{g};

tbb::flow::function_node<join_t::output_type>  out_node{g,
  tbb::flow::unlimited,
  [&](const join_t::output_type& m){
    std::tie (Cdevice, Chost) = m; //распаковать кортеж m

    // Последовательное выполнение
    std::vector<float> CGold(vsize);
    std::transform(Ahost, Ahost + vsize, Bhost, CGold.begin(),
              [&](float a, float b)->float{return a+0.5*b;});

    // Проверить правильность
    if ( ! std::equal(Chost, Chost+vsize, CGold.begin()))
      std::cout << "Error!!\n";
}};
```

Рис. 19.15 ❖ Последние два узла, node_join и out_node, графа гетерогенного вычисления тройственной операции над векторами

В следующих строках тела узла out_node проверяется, что гетерогенное вычисление правильно, для чего сначала последовательно вычисляется эталонный результат тройственной операции CGold, а потом сравнивается с Chost с помощью алгоритма std::equal. Поскольку Chost, Cdevice.data() и Cdevice.begin() на самом деле указывают на один и тот же буфер, следующие три сравнения эквивалентны:

```
std::equal (Chost, Chost+vsize, CGold.begin())
std::equal (Cdevice.begin(), Cdevice.end(), CGold.begin())
std::equal (Cdevice.data(), Cdevice.data()+vsize, CGold.begin())
```

Пора привести наш код в исполнение. На рис. 19.16 добавлены вызовы make_edge и запускается выполнение потокового графа.

Заметим, что хотя все четыре входных порта gpu_node соединены с предшествующим узлом dispatch_node, только порт 2 gpu_node соединен с node_join. Через этот порт передается результирующий буфер Cdevice, так что только о нем нам и нужно беспокоиться. Остальные три позабытых порта не будут чувствовать себя обиженными.

Потребовалось некоторое время, чтобы объяснить пример до конца, но одна вещь все-таки осталась. Как все это соотносится с версией на основе async_node, представленной в предыдущей главе? В нее входил трафаретный код OpenCL, скрытый в функции OpenCL_Initialize(), но все равно необходимый, потому что

он давал доступ к контексту, очереди команд и обработчикам ядра. Версия на основе async_node насчитывает 287 строк (не считая комментариев и пустых строк) при использовании заголовка OpenCL cl.h или 193 строки при использовании обертки этого заголовка на C++ cl.hpp. Новая версия на основе opencl_node меньше – в ней всего 144 строки.

```
// in_node -> dispatch_node
tbb::flow::make_edge(in_node, dispatch_node);
// dispatch_node -> gpu_node
tbb::flow::make_edge(tbb::flow::output_port<0>(dispatch_node),
                     tbb::flow::input_port<0>(gpu_node));
tbb::flow::make_edge(tbb::flow::output_port<1>(dispatch_node),
                     tbb::flow::input_port<1>(gpu_node));
tbb::flow::make_edge(tbb::flow::output_port<2>(dispatch_node),
                     tbb::flow::input_port<2>(gpu_node));
tbb::flow::make_edge(tbb::flow::output_port<3>(dispatch_node),
                     tbb::flow::input_port<3>(gpu_node));
// dispatch_node -> cpu_node
tbb::flow::make_edge(tbb::flow::output_port<4>(dispatch_node),
                     cpu_node);
// gpu_node -> node_join
tbb::flow::make_edge(tbb::flow::output_port<2>(gpu_node),
                     tbb::flow::input_port<0>(node_join));
// cpu_node -> node_join
tbb::flow::make_edge(cpu_node,tbb::flow::input_port<1>(node_join));
// node_join -> out_node
tbb::flow::make_edge(node_join, out_node);

tbb::tick_count t = tbb::tick_count::now();
in_node.activate();
g.wait_for_all();
```

Рис. 19.16 ❖ Последняя часть функции main, в которой соединяются узлы и запускается потоковый граф

Дьявол кроется в деталях

Те, кто раньше разрабатывал программы на OpenCL, знают, что, используя библиотеку OpenCL напрямую, можно «наслаждаться» значительной свободой. Но, на первый взгляд, эта гибкость никак не проявляется в узле opencl_node. Как определить многомерный NDRange? Как задать локальный размер в дополнении к глобальному размеру NDRange? И как предоставить предкомпилированное ядро вместо исходного кода на OpenCL? Быть может, проблема в том, что мы рассмотрели не все рычаги конфигурирования? Давайте ответим на эти вопросы.

Основные функции OpenCL, необходимые для запуска ядра, – это clSetKernelArg (clSetKernelArgSVMPointer, если используются указатели на разделяемую

виртуальную память, доступную в OpenCL 2.x) и `clEnqueueNDRangeKernel`. Эти функции вызываются фабрикой OpenCL, и мы можем контролировать передаваемые им аргументы. Для иллюстрации того, как функции-члены `opencl_node` и вспомогательные функции транслируются в низкоуровневые вызовы OpenCL, заглянем внутрь узла `opencl_node` на рис. 19.17.

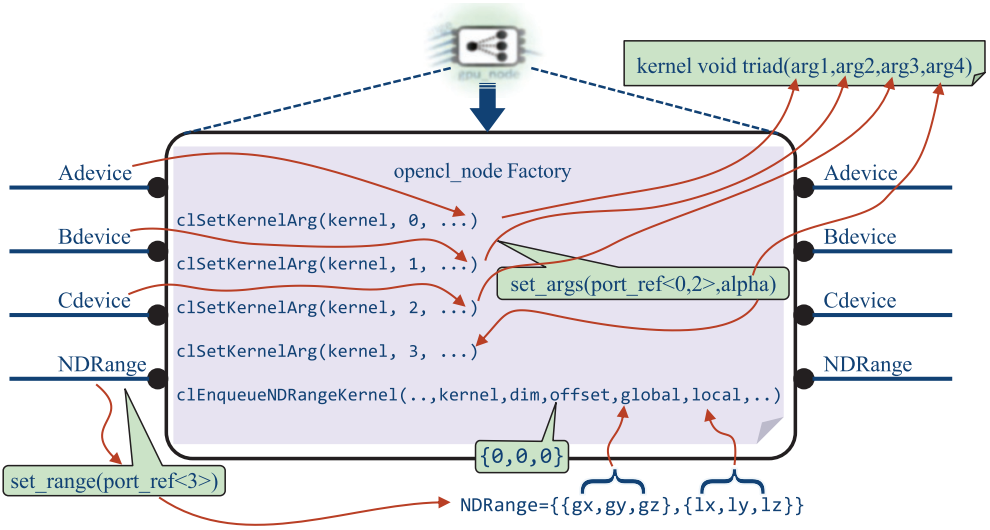


Рис. 19.17 ❖ Внутреннее устройство `opencl_node` и соответствие между функциями `opencl_node` и низкоуровневыми вызовами OpenCL

На этом рисунке используется узел `gpu_node` из примера с тройственной функцией, где мы сконфигурировали `opencl_node`, так что он принимает три буфера `opencl_buffer` и один диапазон `NDRange` (всего четыре порта, через которые сообщения входят в узел и покидают его). Выше мы объяснили, что благодаря вызову `gpu_node.set_args(port_ref<0,2>, alpha)` мы ясно говорим, что первые три входных порта (0, 1 и 2), в которые приходят векторы `A`, `B` и `C`, должны быть привязаны к трем аргументам ядра, а последний аргумент ядра (коэффициент `a`) статически связывается с переменной `alpha`, которая не приходит ни от одного из предшествующих узлов графа. Теперь у нас имеется вся информация, необходимая, чтобы выполнить четыре вызова `clSetKernelArg()`, показанных на рис. 19.17, а они, в свою очередь, вершат магию, благодаря которой эти четыре аргумента оказываются входами в ядерную OpenCL-функцию `void triad(...)`.

Теперь посмотрим, как конфигурируется обращение к `clEnqueueNDRangeKernel`. Это одна из самых сложных функций в OpenCL, она требует девяти аргументов, перечисленных на рис. 19.18. Однако эта книга – не учебник по OpenCL, и в данной главе достаточно обсудить только пять аргументов, со второго по шестой. Аргумент `kernel` мы обсудим ниже, а для понимания остальных четырех нужно глубже познакомиться с одним из фундаментальных понятий OpenCL: `NDRange`.

```
cl_int clEnqueueNDRangeKernel (
  cl_command_queue command_queue,
  cl_kernel kernel,
  cl_uint dim,
  const size_t *offset, //global_work_offset
  const size_t *global, //global_work_size
  const size_t *local,  //local_work_size
  cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list,
  cl_event *event)
```

Рис. 19.18 ❖ Сигнатура OpenCL-функции `clEnqueueNDRangeKernel`

Концепция NDRange

Объект `NDRange` определяет пространство итераций, состоящее из независимых элементов работы. Потенциально это пространство трехмерное, но может быть двумерным или одномерным. В нашем примере с тройственной функцией пространство `NDRange` одномерное. Аргумент `dim` функции `clEnqueueNDrangeKernel` на рис. 19.17 и 19.18 должен принимать соответствующее значение – 1, 2 или 3 – и правильно устанавливается вызовом функции `gpu_node.set_range()`. В примере на рис. 19.17 этот вызов `set_range()` говорит, что информация `NDRange` поступает в порт 3 узла `gpu_node` от предшествующего узла графа. Информация `NDRange` должна содержаться в одном или, факультативно, в двух контейнерах, предоставляющих функции-члены `begin()` и `end()`. Такие функции-члены имеются во многих типах C++, в т. ч. `std::initializer_list`, `std::vector`, `std::array` и `std::list`. Если задан только один контейнер, то `opencl_node` просто устанавливает аргумент `global_work_size` функции `clEnqueueNDRangeKernel()` (на рис. 19.17 и 19.18 ему соответствует переменная `global`). Если же задан также второй контейнер, то `opencl_node` устанавливает еще аргумент `local_work_size` (ему на рис. 19.17 и 19.18 соответствует переменная `local`).

Примечание. Как было сказано, аргумент `NDRange` `global_work_size` определяет пространство параллельных итераций, которые будут выполнены ускорителем. Каждая точка этого пространства в OpenCL называется элементом работы (если вы знакомы с CUDA, то это эквивалент потока CUDA). Следовательно, элементы работы могут обрабатываться параллельно разными вычислительными блоками (compute units – CU) ускорителя, и соответствующее вычисление определяется кодом ядра, т. е. если наша ядерная функция содержит код `C[i]=A[i]+B[i]`, то это выражение будет применено к каждому элементу работы `i` одномерного пространства итераций.

Элементы работы объединяются в так называемые группы работ (или блоки в терминологии CUDA). В силу архитектурных особенностей реализации элементы работы, принадлежащие одной группе, теснее связаны между собой. Например, GPU гарантирует, что группа работ будет запланирована одному вычислительному блоку. Отсюда следует, что мы можем синхронизировать элементы работы в одной группе с барьером OpenCL, и эти элементы будут разделять область памяти, связанную с одним CU, которая называется «локальной памятью» и работает быстрее глобальной.

Аргумент `local_work_size` задает размер группы работ. Драйвер OpenCL умеет автоматически вычислять рекомендованную величину `local_work_size`, если она не задана явно. Однако если мы хотим, чтобы группа работ имела четко определенный размер, то должны задать аргумент `local_work_size`.

Несколько примеров помогут во всем разобраться. Пусть имеются двумерные массивы `A`, `B` и `C` размерности `h×w`, и мы хотим выполнить операцию над матрицами `C=A+B`. Хотя матрицы двумерные, в OpenCL они передаются ядру в виде указателя на линеаризованный по строкам одномерный массив `cl_mem`. Это не мешает нам вычислить одномерный индекс по двумерному, так что ядро выглядит следующим образом:

```
x = get_global_id(0); // столбец
y = get_global_id(1); // строка
C[y*w+x] = A[y*w+x] + B[y*w+x]; // строка x ширина + столбец
```

хотя есть и более заковыристый способ выразить то же самое с помощью типа `int2`:

```
int2 gId = (int2)(get_global_id(0), get_global_id(1));
C[gId.y*w+gId.x] = A[gId.y*w+gId.x] + B[gId.y*w+gId.x];
```

Чтобы получить больше информации о том, что происходит с каждым элементом работы во время выполнения ядра, добавим диагностическую печать:

```
kernel void cl_print(global float* A,
                     global float* B,
                     global float* C,
                     int w) {
  int2 gId =    (int2)(get_global_id(0),   get_global_id(1));
  int2 lId =    (int2)(get_local_id(0),    get_local_id(1));
  int2 grId =   (int2)(get_group_id (0),   get_group_id(1));
  int2 gSize =  (int2)(get_global_size(0), get_global_size(1));
  int2 lSize =  (int2)(get_local_size(0),  get_local_size(1));
  int2 numGrp = (int2)(get_num_groups (0), get_num_groups(1));
  if (gId.x == 0 && gId.y==0)
     printf("gSize.x=%d, gSize.y=%d, lSize.x=%d, lSize.y=%d,
             numGrp.x=%d, numGrp.y=%d\n",
             gSize.x, gSize.y, lSize.x, lSize.y, numGrp.x,
             numGrp);
  printf("gId.x=%d, gId.y=%d, lId.x=%d, lId.y=%d, grId.x=%d,
          grId.y=%d\n\n",
          gId.x, gId.y, lId.x, lId.y, grId.x, grId.y);
  C[gId.y*w+gId.x] = A[gId.y*w+gId.x] + B[gId.y*w+gId.x];
}
```

Рис. 19.19 ❖ Пример ядра, которое складывает две матрицы и печатает информацию об элементах работы

В первых трех переменных, `gId`, `lId` и `grId`, хранится глобальный идентификатор, локальный идентификатор и идентификатор группы каждого элемента работы соответственно в направлениях `x` и `y`. В следующих трех

переменных, gSize, lSize и numGrp, хранится глобальный размер, локальный размер и количество групп работ. Первое условие if удовлетворяется только для элемента работы с глобальным идентификатором (0,0). Поэтому только для него печатаются размеры и количество групп, поскольку они для всех элементов работы одинаковы. Второе предложение printf выполняется для каждого элемента работы и печатает его идентификаторы: глобальный, локальный и группы. Результат показан на рис. 19.20 для случая, когда функция clEnqueueNDrangeKernel вызывалась с аргументами dim = 2, global = {4,4} и local = {2,2}.

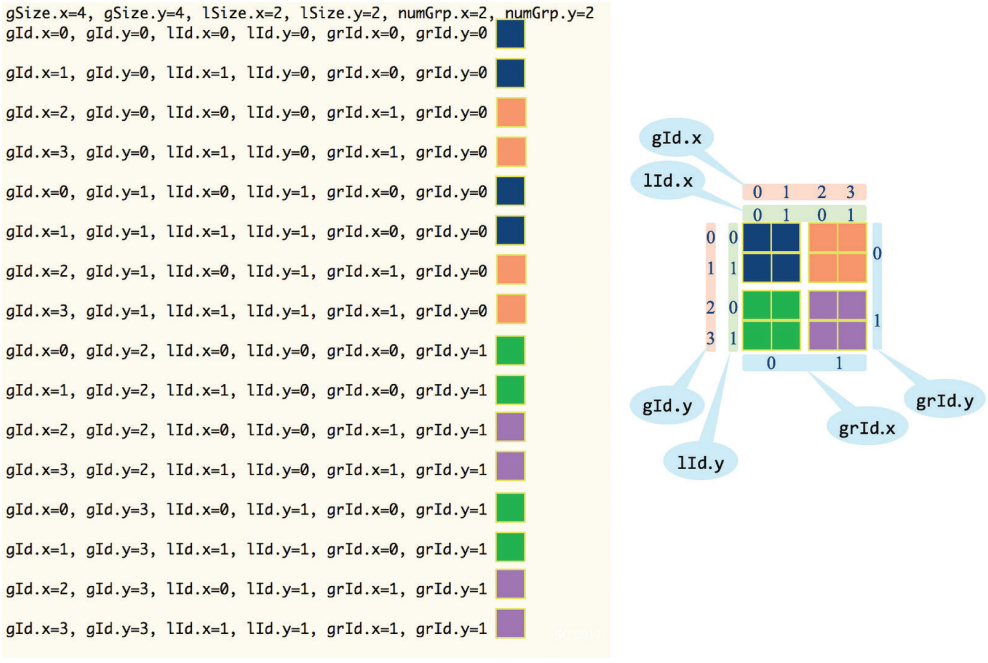


Рис. 19.20 ❖ Вывод ядра на рис. 19.19, сконфигурированного с параметрами dim=2, global={4,4}, local={2,2} --set_range({{4, 4}, {2, 2}})--

На этом рисунке каждый элемент работы изображен цветным квадратиком. Всего имеется 16 элементов работы, организованных в виде сетки 4×4 и разбитых на 4 группы, которым сопоставлены разные цвета. Поскольку локальный размер равен {2,2}, каждая группа работ образует подпространство размера 2×2. Неудивительно, что количество групп равно 4, но чтобы придать этой главе некий формализм, мы введем инварианты, в выполнении которых легко убедиться:

```
numGrp.x = gSize.x/lSize.x
0 <= gId.x < gSize
0 <= lId.x < lSize
gId.x = grId * lSize.x + lId.x
```

и аналогично для координаты .y (и даже .z в трехмерном пространстве).

А теперь вопрос: как задать глобальный и локальный размеры для узла opencl_node? В примерах выше мы просто вызывали gpu_node.set_range({{<num>}}). Это транслируется в параметры dim=1, global={<num>} и local=NULL, что дает одномерный диапазон NDRange, для которого локальный размер устанавливает драйвер OpenCL.

В общем случае нам нужно задать global={gx, gy, gz} и local={lx, ly, lz}. Проще всего сделать это так:

```
gpu_node.set_range({{gx, gy, gz},{lx, ly, lz}});
```

Но, как было сказано, нам подходит любой контейнер, который можно обойти с помощью функции-члена begin(). Более замысловатый способ выразить то же самое выглядит так:

```
std::list<int> list_global = {gx, gy, gz};
std::vector<int> vector_local = {lx, ly, lz};
gpu_node.set_range({list_global, vector_local});
```

Получающийся диапазон имеет столько измерений, сколько элементов в контейнере, а размер по каждому измерению определяется значением соответствующего элемента. Надо только не забывать, что размерности глобального и локального контейнеров должны быть одинаковы.

Чтобы было интереснее, мы должны добавить управляющий код, который сможет запустить ядро на рис. 19.19. Самый краткий из известных нам способов – построить граф с одним узлом opencl_node, как показано на рис. 19.21.

Каково? Всего несколько строчек – у нас есть OpenCL-код для сложения двух матриц A и B. Заметим, что узел gpu_node типа opencl_node имеет всего один порт, port<0>, связанный с третьим аргументом ядра, матрицей C, в котором передается результат вычисления, выполненного ядром. Входные матрицы A и B и ширина матрицы w передаются напрямую с помощью функции-члена set_args. Отметим также, что узел opencl_node должен иметь по крайней мере один порт и активируется, только если сообщение приходит именно в этот порт. Альтернативно gpu_node можно было бы реализовать так:

```
tbb::flow::opencl_node<std::tuple<buffer_f,tbb::flow::opencl_range>>
            gpu_node{g, program.get_kernel("cl_print"), s};
gpu_node.set_range(tbb::flow::port_ref<1>);
gpu_node.set_args(Adevice, Bdevice, tbb::flow::port_ref<0>, w);
tbb::flow::input_port<0>(gpu_node).try_put(Cdevice);
tbb::flow::input_port<1>(gpu_node).try_put(
                    tbb::flow::opencl_range({{w, h}, {w/2, h/2}}));
```

Здесь gpu_node получает Cdevice через port<0>, NDRange через port<1>, а остальные аргументы ядра задаются с помощью функции-члена set_range(). Сообщения, приходящие и уходящие через порт port<1> узла gpu_node, имеют тип tbb::flow::opencl_range (очередной вспомогательный класс для opencl_node!), и мы с помощью try_put() передаем объект opencl_range, инициализированный двумя контейнерами.

```
int h = (argc>1) ? atoi(argv[1]) : 4;
int w = (argc>2) ? atoi(argv[2]) : 4;
size_t vsize = h * w;

tbb::flow::graph g;

using buffer_f = tbb::flow::opencl_buffer<cl_float>;
buffer_f Adevice{vsize};
buffer_f Bdevice{vsize};
buffer_f Cdevice{vsize};
float* Ahost = Adevice.data();
float* Bhost = Bdevice.data();

// Инициализировать A и B
std::iota(Ahost, Ahost+vsize, 0); // 0, 1, 2, 3, ...
std::iota(Bhost, Bhost+vsize, 0); // 0, 1, 2, 3, ...

//Узел GPU:
gpu_device_selector s;
tbb::flow::opencl_program<> program{std::string{"fig_19_19.cl"}};
tbb::flow::opencl_node<std::tuple<buffer_f>> gpu_node{g,
                      program.get_kernel("cl_print"), s};
gpu_node.set_range({{w, h}, {w/2, h/2}});
gpu_node.set_args(Adevice, Bdevice, tbb::flow::port_ref<0>, w);
tbb::flow::input_port<0>(gpu_node).try_put(Cdevice);
g.wait_for_all();
```

Рис. 19.21 ❖ Тестирование `opencl_node` в изоляции

Поиграем со смещением

Мы не рассмотрели еще два аргумента функции `clEnqueueNDRangeKernel` (см. рис. 19.18). Один из них – смещение `offset`, которое позволяет пропустить первые элементы работы в начале пространства итераций. В текущей реализации фабрики OpenCL зашито смещение `{0,0,0}`. Но ничего страшного. Есть два способа обойти это ограничение.

Первый – передать смещение ядру и прибавить его к глобальному идентификатору перед обращением к элементам массива по индексу. Например, для операции сложения одномерных массивов `C = A + B` можно написать что-то вроде

```
kernel void vector-add(global float* A, global float* B,
                       global float* C, int offset){
  int i = get_global_id(0) + offset;
  C[i] = A[i] + B[i];
}
```

И конечно, мы можем адаптировать `NDRange`, чтобы избежать выхода за границы массивов. Это решение работает, пусть оно и не особенно элегантное. А какое тогда элегантное? Для достижения того же результата можно исполь-

зовать класс opencl_subbuffer. Например, если мы хотим сложить только некоторые участки векторов A и B, то можем упростить ядро vector-add:

```
kernel void vector-add(global float* A, global float* B,
                       global float* C){
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

но тогда должны передавать следующие аргументы функции-члену set_args():

```
Adevice.subbuffer(offset, size)
```

и аналогично для Bdevice и Cdevice. Еще один способ создания суббуфера Cdevice – вызвать функцию

```
tbb::flow::opencl_subbuffer<cl_float>(Cdevice, offset, size)
```

Задание ядра OpenCL

Вот мы и дошли до аргумента kernel (рис. 19.18). До сих пор мы задавали ядро с помощью исходных файлов OpenCL. В последнем примере на рис. 19.21 мы снова воспользовались классом opencl_program:

```
tbb::flow::opencl_program<> program{std::string{"fig_19_19.cl"}};
```

что эквивалентно более явному конструктору:

```
tbb::flow::opencl_program<> program{opencl_program_type::SOURCE,
                                    std::string{"fig_19_19.cl"}};
```

Именно так ядро обычно и передается. С одной стороны, при этом приходится компилировать ядро во время выполнения, но с другой – обеспечивается переносимость, потому что исходный код будет компилироваться (только один раз при конструировании opencl_program) для любого имеющегося устройства. Фабрика OpenCL использует для этой цели функции clCreateProgramWithSource и clBuildProgram.

Если мы уверены, что код не придется переносить на другую платформу или если в производственной версии мы хотим выжать всю производительность до последней капли, то можем откомпилировать ядро заранее. При работе с инструментами Intel OpenCL для этого нужно выполнить команду

```
ioc64 -cmd=build -input=my_kernel.cl -ir=my_kernel.clbin
      -bo="-cl-std=CL2.0" -device=gpu
```

которая генерирует файл my_kernel.clbin. Теперь объект программы можно создать быстрее:

```
opencl_program<> program{opencl_program_type::PRECOMPILED,
                         std::string{"my_kernel.clbin"}};
```

Если конструктору opencl_program передан файл такого типа, то фабрика пользуется функцией clCreateProgramWithBinary. Есть еще одна возможность – передать промежуточное SPIR-представление ядра, указав тип opencl_program_type::SPIR. Для генерации SPIR-представления служит команда

```
ioc64 -cmd=build -input=my_kernel.cl -spir64=my_kernel.spir
      -bo="-cl-std=CL1.2"
```

В обоих случаях компилятор ioc64 предоставляет полезную информацию. Так, последняя команда выводит:

```
Using build options: -cl-std=CL1.2
OpenCL Intel(R) Graphics device was found!
Device name: Intel(R) HD Graphics
Device version: OpenCL 2.0
Device vendor: Intel(R) Corporation
Device profile: FULL_PROFILE
fcl build 1 succeeded.
bcl build succeeded.
my_kernel info:
      Maximum work-group size: 256
      Compiler work-group size: (0, 0, 0)
      Local memory size: 0
      Preferred multiple of work-group size: 32
      Minimum amount of private memory: 0
Build succeeded!
```

В числе прочего это говорит нам о том, что максимальный размер группы работ равен 256, а размер группы работ для этого ядра предпочтительнее выбирать кратным 32.

Еще о выборе устройства

В предыдущем разделе мы выяснили, что ноутбук, на котором мы экспериментируем, содержит два GPU. Рассмотрим пример, в котором оба они используются в одном потоковом графе. На рис. 19.22 два узла `opencl_node` соединены так, что первый вычисляет `C = A + B` и отправляет `C` следующему, который вычисляет `C = C - B`. По завершении работы обоих узлов мы в обычном узле типа `function_node` проверяем, что `C == A`. Размер массива равен `rows × cols`.

Мы уже знаем, что на нашем ноутбуке список устройств `f.devices()` содержит три устройства, причем только второе и третье действительно являются GPU. Таким образом, для указания на каждый GPU можно без опаски использовать `f.devices().begin()` `+1` и `+2`, как показано в обведенных рамкой предложениях на рис. 19.22. Помимо того что каждый `opencl_node` настроен на свой GPU, для них еще и заданы разные ядра в программном файле `fig_19_23.cl`: `cl_add` и `cl_sub`. Информация передается от `gpu_node1` к `gpu_node2` в виде объекта `opencl_buffer Cdevice`. Внутри фабрики OpenCL перемещение данных минимизировано, и если, например, к `opencl_buffer` будут обращаться два последовательных узла `opencl_node`, отображенных на один и тот же GPU, то данные, выделенные на GPU, не будут перемещаться на ЦП, пока первый же ЦП-узел графа не попытается обратиться к соответствующему буферу (с помощью функций-членов `opencl_buffer.begin()` или `opencl_buffer.data()`).

```
tbb::flow::graph g;
// Инициализировать A и B
…
tbb::flow::opencl_program<> program{std::string{"fig_19_23.cl"}};
using tuple_gpu = std::tuple<buffer_f>;

//Узел GPU 1:
tbb::flow::opencl_node<tuple_gpu> gpu_node1{g,
  program.get_kernel("cl_add"),
  [](auto& f){
    auto d = *(f.devices().begin() + 1);
    std::cout << "Running gpu_node1 on " << d.name()<< '\n';
    return d;
  }};
gpu_node1.set_range({{cols, rows}});
gpu_node1.set_args(Adevice, Bdevice, tbb::flow::port_ref<0>);

//Узел GPU 2:
tbb::flow::opencl_node<tuple_gpu> gpu_node2{g,
  program.get_kernel("cl_sub"),
  [](auto& f){
    auto d = *(f.devices().begin() + 2);
    std::cout << "Running gpu_node2 on " << d.name()<< '\n';
    return d;
  }};
gpu_node2.set_range({{cols, rows}});
gpu_node2.set_args(Adevice, Bdevice, tbb::flow::port_ref<0>);

//Выходной узел:
tbb::flow::function_node<buffer_f> out_node{g, tbb::flow::unlimited,
  [&](buffer_f const& Cdevice){
    float* Chost = Cdevice.data();
    if (! std::equal(Chost, Chost+vsize, Ahost))
      std::cout << "Errors in the heterogeneous computation.\n";
  }
};
make_edge(tbb::flow::output_port<0>(gpu_node1),
          tbb::flow::input_port<0>(gpu_node2));
make_edge(tbb::flow::output_port<0>(gpu_node2), out_node);
tbb::flow::input_port<0>(gpu_node1).try_put(Cdevice);
g.wait_for_all();
```

Рис. 19.22 ❖ Пример с двумя узлами `opencl_node`, сконфигурированными для работы с разными GPU

На рис. 19.23 представлена программа `fig_19_23.cl`, включающая два ядра, на которые ссылается приведенный выше код. Теперь вместо передачи ширины строки в четвертом аргументе мы используем переменную-член `gSz.x`, которая содержит то же самое значение.

```
kernel void cl_add(global float* A,global float* B,global float* C){
  int2 gId = (int2)(get_global_id(0), get_global_id(1));
  int2 gSz = (int2)(get_global_size(0), get_global_size(1));

  if(gId.x == 0 && gId.y==0)
    printf("gSz.x=%d, gSz.y=%d\n",gSz.x,gSz.y);
  C[gId.y*gSz.x+gId.x]=A[gId.y*gSz.x+gId.x]+B[gId.y*gSz.x+gId.x];
}

kernel void cl_sub(global float* A,global float* B,global float* C){
  int2 gId = (int2)(get_global_id(0), get_global_id(1));
  int2 gSz = (int2)(get_global_size(0), get_global_size(1));

  if(gId.x == 0 && gId.y==0)
    printf("gSz.x=%d, gSz.y=%d\n",gSz.x,gSz.y);
  C[gId.y*gSz.x+gId.x]=C[gId.y*gSz.x+gId.x]-B[gId.y*gSz.x+gId.x];
}
```

Рис. 19.23 ❖ Содержимое файла `fig_19_23.cl`, на котором показаны два ядра, вызываемых из разных узлов `opencl_node`

На нашем ноутбуке программа на рис. 19.22 напечатала:

```
Running gpu_node1 on Intel(R) HD Graphics 530
Running gpu_node2 on AMD Radeon Pro 450 Compute Engine
gSz.x=4, gSz.y=4
gSz.x=4, gSz.y=4
```

Можно также завести один узел `opencl_node`, изменяющий OpenCL-устройство, которому «скидывается» работа при каждом вызове. В примере на рис. 19.24 показан узел `opencl_node`, который вызывается трижды, и каждый раз для выполнения простого ядра используется другое устройство.

В этом коде используется атомарная переменная `device_num`, инициализированная значением 0. При каждом обращении к `gpu_node` возвращается другое устройство, причем они перебираются циклически (на данной платформе устройств три). Если задано такое ядро:

```
kernel void cl_inc(global int* A) {
  int gId = get_global_id(0);
  A[gId]++;
  if(gId == 0) printf("A[0]=%d\n", A[0]);
}
```

то будет напечатано:

```
Iteration: 0
Iteration: 1
Iteration: 2
Running on Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Running on Intel(R) HD Graphics 530
Running on AMD Radeon Pro 450 Compute Engine
A[0]=1
A[0]=2
A[0]=3
```

Откуда можно сделать вывод, что элементы массива Adevice увеличивались трижды в каждом из трех последовательных вызовов gpu_node и что соответствующее ядро выполнялось тремя разными OpenCL-устройствами.

```
tbb::flow::graph g;
using buffer_f = tbb::flow::opencl_buffer<cl_int>;
buffer_f Adevice{vsize};
int* Ahost = Adevice.data();
std::iota(Ahost, Ahost+vsize, 0); // 0, 1, 2, 3, ...

//Узел GPU
tbb::flow::opencl_program<> program{std::string{"fig_19_24.cl"}};
tbb::atomic<int> device_num{0};
tbb::flow::opencl_node<std::tuple<buffer_f>> gpu_node{g,
  program.get_kernel("cl_inc"),
  [&device_num](auto& f){
    auto d=*(f.devices().begin()+device_num++ % f.devices().size());
    std::cout << "Running on " << d.name() << '\n';
    return d;
  }};
gpu_node.set_range({{static_cast<int>(vsize)}});
gpu_node.set_args(tbb::flow::port_ref<0>);
for(int i=0; i<3; i++){
  std::cout << "Iteration: " << i << '\n';
  tbb::flow::input_port<0>(gpu_node).try_put(Adevice);
}

g.wait_for_all();
std::vector<int> AGold(vsize);
std::iota(AGold.begin(), AGold.end(), 3); // 3, 4, 5, 6, ...
if (! std::equal(Adevice.begin(), Adevice.end(), AGold.begin()))
  std::cout << "Errors in the heterogeneous computation.\n";
```

Рис. 19.24 ❖ Один opencl_node может изменять целевой ускоритель при каждом вызове

Предупреждение по поводу порядка

И еще один подводный камень, о котором следует знать, связан с порядком прихода сообщений в узел opencl_node, если он обслуживает несколько узлов. Например, на рис. 19.25 показан потоковый граф g, включающий узел gpu_node, в который поступают сообщения от двух функциональных узлов, filler0 и filler1. Каждый из них отправляет 1000 буферов b по 10 целых чисел в каждом вида {i,i,i,…,i}, где i изменяется от 1 до 1000. Принимающий gpu_node получает эти сообщения как b1 и b2 и вызывает простейшее OpenCL-ядро:

```
kernel void mul(global int* b1, global int* b2){
  const int index = get_global_id(0);
  b1[index] *= b2[index];
}
```

Это ядро вычисляет `b1[i]=b1[i]*b2[i]`. Если `b1` и `b2` равны (`{1,1,1,…}`, `{2,2,2,…}` и т. д.), то мы должны получить на выходе 1000 буферов, содержащих квадраты целых чисел: `{1,1,1,…}`, затем `{4,4,4,…}` и т. д. Правильно? Уверены? Не хотелось бы оказаться лгунами, поэтому на всякий случай проверим это предположение в последнем узле графа `checker`.

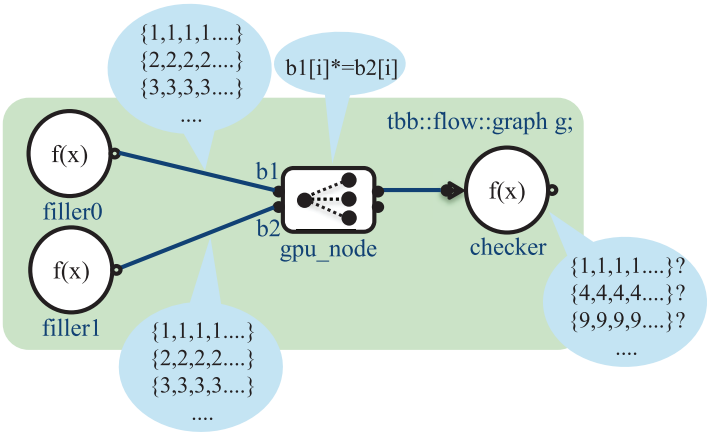


Рис. 19.25 ❖ Два функциональных узла, отправляющих узлу `opencl_node` буферы, которые должны перемножаться на GPU

На рис. 19.26 приведен код, реализующий этот граф. Мы согласны с Джорджем Бернардом Шоу в том, что «Наказание лжеца совершенно не в том, что ему никто не верит, а в том, что он сам никому не может верить». Уж мы-то на лжи собаку съели, поэтому включили в код конструкцию `try-catch` специально для того, чтобы хватать шулеров за руку!

Сначала мы определяем `buffer_i` как `opencl_buffer`, содержащий целые числа. Оба узла `filler` получают целое `i` и заполняют `buffer_i` десятью копиями `i`, после чего отправляют буфер узлу `gpu_node`. Три строки, где конфигурируется `opencl_node`, нам уже знакомы и не требуют пояснений. Последний узел осуществляет проверку и возбуждает исключение, если хотя бы одно из чисел в буфере, обработанном на GPU, не является полным квадратом. Затем мы проводим ребра и запускаем цикл на 1000 итераций. Вот и наступил момент истины – печатается сообщение:

```
Liar!!: 42 is not a square of any integer number
```

Нас поймали за руку! Очевидно, GPU вычислял `6*7`, а не `6*6` или `7*7`. Но почему? А потому, что мы не предприняли никаких мер, гарантирующих, что сообщения, приходящие узлу `gpu_node`, правильно объединены в пары. Напомним, что тела «заполнителей» `filler` исполняются задачами, и мы не можем делать предположений о порядке их выполнения.

```
using buffer_i = tbb::flow::opencl_buffer<int>;
try {
  constexpr int n = 10;
  tbb::flow::graph g;
  tbb::flow::function_node<int, buffer_i> filler0{g,
    tbb::flow::unlimited,
    [n](int i){
      buffer_i b{n};
      std::fill(b.begin(), b.end(), i);
      return b;
    }};

  tbb::flow::function_node<int, buffer_i> filler1 = filler0;

  tbb::flow::opencl_program<> program{std::string{"mul.cl"}};
  tbb::flow::opencl_node<std::tuple<buffer_i, buffer_i>> gpu_node{g,
                                        program.get_kernel("mul")};
  gpu_node.set_range({{ n }});

  tbb::flow::function_node<buffer_i> checker{g,
    tbb::flow::serial,
    [](const buffer_i& b){
      for (int v : b){
        int r = static_cast<int>(std::sqrt(v) + .5);
        if (r*r != v)
        throw std::runtime_error(std::to_string(v) +
                        " is not a square of any integer number" );
      }
    }};

  tbb::flow::make_edge(filler0, tbb::flow::input_port<0>(gpu_node));
  tbb::flow::make_edge(filler1, tbb::flow::input_port<1>(gpu_node));
  tbb::flow::make_edge(tbb::flow::output_port<0>(gpu_node), checker);

  for (int i = 1; i<=1000; ++i){
    filler0.try_put(i);
    filler1.try_put(i);
  }
  g.wait_for_all();
}
catch (std::exception& e){
  std::cerr << "Liar!!: " << e.what() << std::endl;
}
```

Рис. 19.26 ❖ Исходный код, соответствующий графу на рис. 19.25

По счастью, opencl_node содержит удобный механизм сравнения ключей, который спасает положение. Мы воспользовались им на рис. 19.27.

```
// было: using buffer_i = tbb::flow::opencl_buffer<int>;
class buffer_i : public tbb::flow::opencl_buffer<int>  {
  int my_key;
public:
  buffer_i() {}
  buffer_i(size_t n, int k) : tbb::flow::opencl_buffer<int>{n},
                              my_key{k} {}
  int key() const {return my_key;}
};
…
    tbb::flow::function_node<int, buffer_i> filler0{g,
      tbb::flow::unlimited,
      [n](int i){
        buffer_i b{n, i};
        std::fill(b.begin(), b.end(), i);
        return b;
      }};

    tbb::flow::function_node<int, buffer_i> filler1 = filler0;

    tbb::flow::opencl_program<> program{std::string{"mul.cl"}};
    tbb::flow::opencl_node<std::tuple<buffer_i, buffer_i>,
                           tbb::flow::key_matching<int> >
                           gpu_node{g, program.get_kernel("mul")};
    gpu_node.set_range({{ n }});
…
```

Рис. 19.27 ❖ Исправление кода рис. 19.26

Теперь `buffer_i` стал новым классом, который наследует `opencl_buffer<cl_int>` и добавляет переменную-член `my_key` и функцию-член `key()`, возвращающую ее. У заполнителей сейчас другой конструктор (`buffer_i b{N,i}`), но важнее, что `opencl_node` принимает второй аргумент шаблона (`key_matching<int>`). Это означает, что `opencl_node` должен вызывать функцию `key()` и ждать, когда во все входные порты придут сообщения с одинаковым ключом. Готово! Если теперь выполнить код с этими незначительными изменениями, то мы увидим, что обвинения в лжесвидетельстве с нас сняты!

Резюме

В этой главе мы рассказали об узлах типа `opencl_node` потокового графа в TBB. Мы начали с простого примера «Hello OpenCL_Node», где свели знакомство с `opencl_node`, рассмотрев простейшие возможности этого класса. Затем мы углубились во вспомогательные классы, в т. ч. `opencl_device_list` – контейнер объектов `opencl_device`, а также рассмотрели фильтры устройств и селекторы устройств. Чтобы проиллюстрировать другие вспомогательные классы, а заодно разобрать более сложный пример, мы реализовали вычисление тройственной операции над векторами, поручив узлу `opencl_node` выполнить одну часть вычислений, а другую – одновременно выполняя на ЦП. Это позволило нам ввести в рассмотрение вспомогательный класс `opencl_buffer`, а также функции-

члены set_range и set_args класса opencl_node. Для изложения концепции NDRange задания глобального и локального размеров OpenCL понадобился почти целый раздел, где мы попутно объяснили, как использовать класс opencl_subbuffer, и поведали о различных способах задания программы ядра (предварительно откомпилированного или в виде промежуточного SPIR-представления). Затем мы привели два примера для иллюстрации отображения разных узлов opencl_node на разные устройства и того, как можно изменить устройство, которому opencl_node скидывает работу, при каждом вызове. Наконец, мы описали, как избежать проблем с упорядочением в случае, когда opencl_node получает сообщения от разных узлов.

И последнее. Возможно, мы все-таки в чем-то солгали, но это не наша вина. На момент написания этой книги opencl_node все еще считался ознакомительным средством, а значит, окончательная версия может отличаться. После трех лет разработки существенные изменения вряд ли будут, но твердо обещать мы не можем. Если в будущей версии такие изменения все-таки появятся, обещаем переработать эту главу! Вы нам верите?

Дополнительная информация

Ниже приведены дополнительные рекомендуемые материалы по теме этой главы.

- *Alexei Katranov.* Opencl_node overview // Серия статей в блоге Intel Developer Zone: https://software.intel.com/en-us/blogs/2015/12/09/opencl-node-overview.
- *David Kaeli, Perhaad Mistri, Dana Schaa, Dong Ping Zhang.* Heterogeneous Computing with OpenCL 2.0. Morgan Kaufmann, 2015.

Значок альпиниста на рис. 19.1 принадлежит Скотту де Йонге (Scott de Jonge) и опубликован на сайте www.flaticon.com.

Глава 20

TBB в системах с архитектурой NUMA

Знающие программисты, пекущиеся о производительности, понимают важность учета локальности. Когда слышишь о локальности, первое, что приходит на ум, – локальность кеша, но часто для тяжелых приложений, работающих в больших системах с архитектурой разделяемой памяти, следует рассматривать также локальность при неравномерном доступе к памяти (Non-Uniform Memory Access – NUMA). Вы, конечно, знаете, что идея NUMA в том, что память организована в виде разных банков, и некоторые ядра быстрее обращаются к «близким» банкам, чем к «далеким». Формально узел NUMA – это группа, состоящая из процессорных ядер, кешей и локальной памяти, в которой время доступа всех ядер к локальным разделяемым кешам и памяти одинаково. Время доступа из одного узла NUMA к другому может быть гораздо больше. Возникает ряд вопросов, например: как выделять память для структур данных программы на разных узлах NUMA, и где работают потоки, которые эти структуры обрабатывают (близко к данным они находятся или далеко)? В этой главе мы ответим на поставленные вопросы и, что важнее, объясним, как воспользоваться локальностью NUMA в параллельном приложении TBB.

Настройка производительности в NUMA-системах сводится к четырем действиям: (1) определение топологии платформы; (2) определение затрат на доступ к памяти из разных узлов системы; (3) контроль над местами хранения данных (размещение данных); (4) контроль над местами выполнения работы (привязка к процессорам).

Чтобы вы не испытали разочарования в дальнейшем (уж если расстраиваться, то прямо сейчас!), сразу скажем: в настоящее время TBB не предлагает высокоуровневых средств, позволяющих задействовать локальность NUMA. Иными словами, из четырех перечисленных выше действий TBB готова както помочь только в четвертом: мы можем рассчитывать на классы `task_arena` (см. главу 12) и локальный `task_sheduler_observer` (см. главу 13), чтобы указать, какие потоки должны быть ограничены узлом NUMA. Для всех остальных действий, в т. ч. для привязки потоков к узлам NUMA (а это существенная часть четвертого действия), придется использовать либо низкоуровневые системно зависимые вызовы, либо высокоуровневые сторонние библиотеки и инструменты. Так что даже в этой посвященной TBB книге эта последняя глава не только о TBB. Наша цель – скрупулезно рассмотреть вопрос о реализации кода

для задействования локальности NUMA, пусть даже большая часть требуемых действий напрямую не связана с TBB.

Предупредив читателя, перейдем к организации этой главы. В целом разделы следуют в том же порядке, что и перечисленные выше действия. В первом разделе описываются инструменты, с помощью которых можно определить топологию платформы и узнать, сколько имеется узлов NUMA. Если больше одного, то можно переходить к следующему разделу. В нем мы используем тест производительности, чтобы получить представление о том, на какое ускорение можно рассчитывать, если воспользоваться локальностью NUMA на данной платформе. Если ожидаемый выигрыш впечатляет, то можно задуматься о том, как задействовать локальность в своем коде (а не только в простом тесте). Если мы понимаем, что локальность данных может принести плоды в нашем приложении, то можем заняться проблемой по существу: решить вопросы размещения данных и привязки к процессорам. Располагая знаниями о платформе и классами `task_arena` и `task_scheduler_observer`, мы реализуем простое приложение, в котором используется локальность NUMA, и оценим ускорение по сравнению с эталонной реализацией. Весь процесс показан на рис. 20.1. В заключение мы очертим более продвинутые и общие подходы, которые стоило бы рассмотреть в более сложных приложениях.

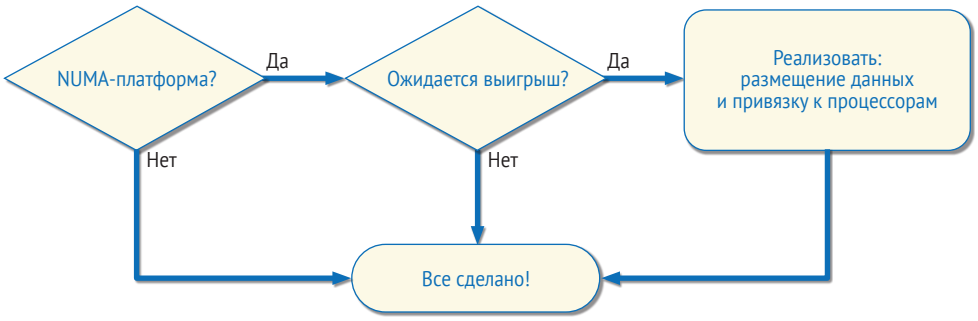


Рис. 20.1 ❖ Действия, необходимые, чтобы задействовать локальность NUMA

Примечание. Для тех, кому интересно, почему в текущей версии TBB нет высокоуровневой поддержки NUMA, укажем несколько причин. Во-первых, это действительно трудная задача, сильно зависящая от конкретного распараллеливаемого приложения и архитектуры системы, в которой оно будет работать. Поскольку универсального решения не существует, выбор оптимальных размещения данных и привязки к процессорам остается на усмотрение разработчиков. Во-вторых, архитекторы и разработчики TBB всегда старались избегать решений, зависящих от оборудования, потому что они могут войти в противоречие с переносимостью кода и средствами компонуемости. Библиотека создавалась не только для высокопроизводительных вычислений (HPC), когда у приложения обычно имеется монопольный доступ ко всей высокопроизводительной платформе (или ее разделу). TBB должна делать все возможное также в разделяемых средах, где работает несколько приложений или процессов. Привязка потоков к процессорным ядрам, а памяти – к узлам NUMA во многих случаях ведет к неоптимальному использованию системной архитектуры. Мы неоднократно демонстрировали, что ручная привязка к потокам – неудачная идея в любом приложении или системе, где имеется

хоть какая-то динамика. Мы настоятельно рекомендуем отказаться от такого подхода, если нет абсолютной уверенности в том, что производительность конкретного приложения на конкретной параллельной платформе удастся увеличить, а о переносимости (или дополнительных усилиях, необходимых, чтобы сделать приложение, знающее о NUMA, переносимым) можно не думать.

Учитывая, что в основе параллельных TBB-алгоритмов и планировщика с заимствованием работ лежат задачи, размещение задач на ядрах, близких к локальной памяти, может показаться трудной проблемой. Но это не должно отпугнуть таких бравых и бесстрашных программистров, как мы. Вперед, на приступ!

Определение топологии платформы

«Знай своего врага и знай себя, и ты сможешь провести тысячу битв без поражений», – писал Сунь Цзы в «Искусстве войны». Цитата из книги, которой уже тысяча лет, рекомендует сначала дотошно разобраться, с чем мы имеем дело, и только потом предпринимать какие-то действия. Для определения архитектуры NUMA имеются инструменты. В этой главе мы будем пользоваться программами `hwloc` и `likwid`[1], чтобы собрать информацию об архитектуре и выполнении кода. Пакет программ `hwloc` предлагает переносимый способ опрашивать топологию системы и применять некоторые средства управления NUMA, в т. ч. размещение данных и привязку к процессорам. А `likwid` – еще один пакет для сбора информации о топологии оборудования – можно использовать для получения счетчиков производительности оборудования, а также как источник полезных микротестов производительности для получения характеристик системы. Можно также анализировать производительность своего кода с помощью программы VTune. Хотя `likwid` существует только для Linux, `hwloc` и VTune легко установить также в Windows и MacOS. Но, поскольку платформы с разделяемой памятью, на которых мы будем иллюстрировать код, работают под управлением Linux, мы будем предполагать именно эту ОС, если явно не оговорено противное.

Поскольку для настройки NUMA нужно очень хорошо понимать используемую платформу, начнем с перечисления характеристик двух машин, на которых будем работать в этой главе. Будем называть эти машины **yuca** (от названия растения «юкка») и **aloe** (название растения «алоэ вера»). Для начала соберем базовую информацию о машинах. В Linux для этой цели можно воспользоваться командой `lscpu`, результат которой показан на рис. 20.2.

Сразу же видно, что yuca имеет 64 логических ядра, пронумерованных от 0 до 63, по два логических ядра на одно физическое (т. е. реализован гипертрединг, или одновременная многопоточность, англ. SMT), по восемь физических ядер на сокет, а всего четыре сокета, которые называются также узлами, или доменами, NUMA. С другой стороны, aloe имеет 32 физических ядра, гипертрединг выключен (в каждом физическом ядре работает только один поток),

[1] www.open-mpi.org/projects/hwloc и https://github.com/RRZE-HPC/likwid.

в каждом сокете 16 физических ядер, а всего сокетов (узлов NUMA) два. В конце выдачи `lscpu` показаны узлы NUMA и идентификаторы логических ядер, включенных в состав каждого узла, но картина прояснится, если воспользоваться утилитой `lstopo` из пакета `hwloc`. На рис. 20.3 показано содержимое PDF-файла, который был сгенерирован на yuca в результате выполнения команды `lstopo --no-io yuca.pdf` (флаг `--no-io` отменяет включение информации о топологии устройств ввода/вывода).

yuca

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	64
On-line CPU(s) list:	0-63
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	4
NUMA node(s):	4
Vendor ID:	GenuineIntel
CPU family:	6
Model:	46
Stepping:	6
CPU MHz:	1064.000
BogoMIPS:	3999.02
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	18432K
NUMA node0 CPU(s):	0-7,32-39
NUMA node1 CPU(s):	8-15,40-47
NUMA node2 CPU(s):	16-23,48-55
NUMA node3 CPU(s):	24-31,56-63

aloe

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	32
On-line CPU(s) list:	0-31
Thread(s) per core:	1
Core(s) per socket:	16
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Stepping:	2
CPU MHz:	1200.000
BogoMIPS:	4601.10
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	40960K
NUMA node0 CPU(s):	0-15
NUMA node1 CPU(s):	16-31

Рис. 20.2 ❖ Вывод lscpu на машинах yuca и aloe

Из этого рисунка можно получить ясное представление об организации NUMA на машине yuca. Каждый из четырех узлов NUMA содержит восемь физических ядер, которые ОС видит как 16 логических (или аппаратных) потоков. Заметим, что идентификаторы логических ядер зависят от архитектуры, прошивки (конфигурации BIOS в случае ПК) и версии ОС, поэтому по номеру мы ничего сказать не можем. В данной конкретной конфигурации yuca логические ядра 0 и 32 принадлежат одному физическому ядру. Теперь мы лучше понимаем смысл последних четырех строк выдачи `lscpu` на yuca:

```
NUMA node0 CPU(s):   0-7,32-39
NUMA node1 CPU(s):   8-15,40-47
NUMA node2 CPU(s):   16-23,48-55
NUMA node3 CPU(s):   24-31,56-63
```

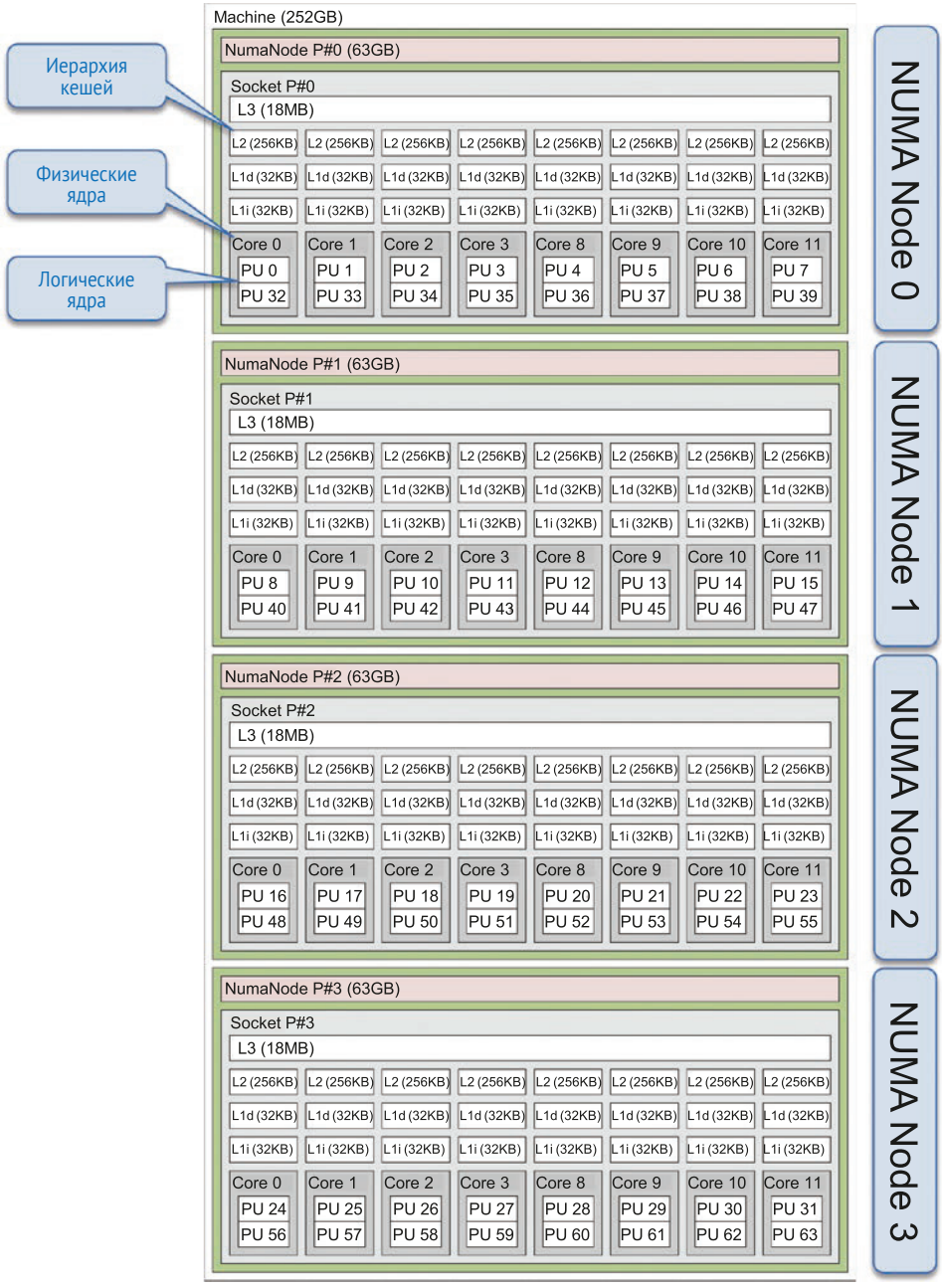


Рис. 20.3 ❖ Результат выполнения `lstopo` на машине yuca

На yuca каждый узел NUMA имеет 63 ГБ локальной памяти, всего 252 ГБ. На машине aloe объем памяти тоже равен 252 ГБ, но разделена она только между двумя узлами NUMA. На рис. 20.4 показана немного отредактированная выдача `lstopo` на машине aloe.

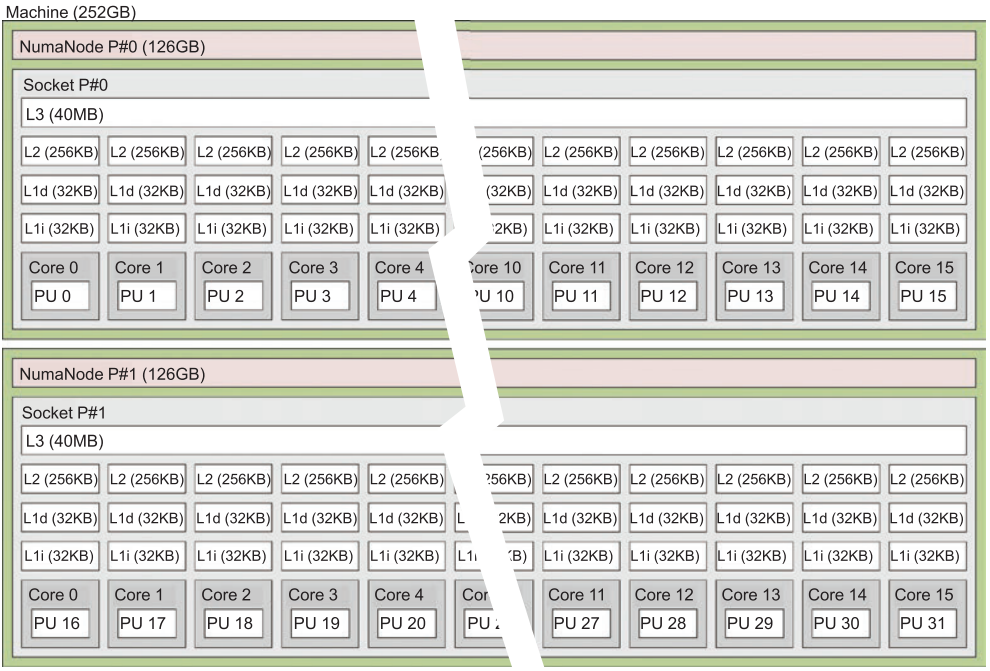


Рис. 20.4 ❖ Результат выполнения `lstopo` на машине aloe

Мы видим, что на aloe каждое физическое ядро содержит одно логическое, и нумеруются они от 0 до 15 в первом домене и от 16 до 31 во втором.

Каковы затраты на доступ к памяти

Теперь, зная топологию платформы, оценим количественно накладные расходы вследствие нелокального доступа в предположении, что мы уже умеем контролировать размещение данных и привязку к процессору. Фактически мы и так контролируем оба этих аспекта в имеющихся тестах производительности, например `likwid-bench`, входящем в состав инструмента `likwid`. Применяя этот тест, мы можем запустить тройственную операцию STREAM (см. две предыдущие главы) командой

```
likwid-bench -t stream -i 1 -w S0:12GB:16-0:S0,1:S0,2:S0
```

которая выполняет одну итерацию (`-i 1`) теста stream, сконфигурированного с помощью аргумента `-w`, интерпретируемого следующим образом:

- S0: потоки привязаны к узлу 0 NUMA;
- 12 GB: три массива, участвующих в тройственной операции, занимают всего 12 ГБ (по 4 ГБ на массив);
- 16: в вычислении участвуют 16 потоков, каждый из которых обрабатывает порцию, содержащую 31 250 000 чисел типа `double` (4000 млн байт / 8 байт на `double` / 16 потоков);
- 0:S0,1:S0,2:S0: все три массива размещены в узле 0 NUMA.

На машине yuca эта команда сообщает, что пропускная способность составляет 8219 МБ/с. Но было бы безрассудно изменить размещение всех трех массивов, перенеся их, например, на узел 1 NUMA (задав параметр `0:S1,1:S1,2:S1`) и оставив 16 вычислительных потоков на узле 0. Неудивительно, что в итоге пропускная способность сократилась до 5110 МБ/с, т. е. мы потеряли 38 % по сравнению с ситуацией, когда имела место локальность NUMA. Аналогичные результаты получаются для других конфигураций, в которых обсчитываются локальные данные (данные размещены на тех же ядрах, к каким привязаны потоки), и конфигураций, где локальности нет (данные размещены на ядрах без привязки потоков). На yuca все нелокальные конфигурации приводят к одинаковой потере пропускной способности, но существуют другие топологии NUMA, в которых величина штрафа зависит от того, где размещены данные и где работают потоки.

На машине aloe имеется только два узла NUMA, 0 и 1. Если вычисления производятся в том же домене, где находятся данные, то пропускная способность составляет 38 671 МБ/с, тогда как в противном случае мы получаем лишь 20 489 МБ/с (пропускная способность упала на 47 %, почти вдвое). Уверены, что вы, любезный читатель, теперь горите желанием узнать о том, как повысить производительность, воодушевленные шансом воспользоваться локальностью NUMA в собственных проектах!

Базовый пример

На рис. 20.5 показана параллельная версия вычисления тройственной операции с помощью одного лишь алгоритма `parallel_for`.

В последних двух строках этого кода, еще не оптимизированного для NUMA, печатается время работы и достигнутая пропускная способность. Для вычисления последней подсчитывается общее число прочитанных и записанных байтов – `vsize` × 8 байт на каждое `double` × 3 операции доступа на каждый элемент массива (два чтения, одна запись), и эта величина делится на время выполнения и на миллион (чтобы привести к единице измерения МБ в секунду). На yuca при обработке массива, содержащего 10^9 элементов, в 32 потока получаем:

```
./fig_20_05 32 1000000000
Time: 2.23835 seconds; Bandwidth: 10722.2MB/s
```

а на aloe:

```
./fig_20_05 32 1000000000
Time: 0.621695 seconds; Bandwidth: 38604.2MB/s
```

Заметим, что пропускную способность, полученную в результате вычисления тройственной операции, не следует сравнивать с предыдущими результатами `likwid-bench`. Теперь мы используем 32 потока (а не 16), которые, по усмотрению планировщика ОС, могут работать на любом ядре (необязательно на одном узле NUMA). Да и массивы ОС размещает в соответствии с собственной стратегией. В Linux по умолчанию применяется стратегия[1] «локального

[1] Мы можем узнать, какая стратегия NUMA используется, командой `numactl -show`.

выделения», в которой поток, выделяющий память, решает, где размещать данные: в локальной памяти, если места хватает, в противном случае в удаленной. Иногда эту стратегию называют «первое прикосновение», потому что размещение данных производится не в момент выделения памяти, а в момент первого обращения. Это означает, что некоторый поток может выделить область, но страничный отказ генерирует поток, первым обращающийся к этой области, и в реальности страница выделяется в памяти, локальной для этого потока. В примере на рис. 20.5 один и тот же поток выделяет память для массивов и инициализирует их, поэтому рабочие потоки `parallel_for`, работающие на одном и том же узле NUMA, будут обращаться к памяти быстрее. И последнее различие связано с тем, что `likwid-bench` реализует вычисление тройственной операции на языке ассемблера, что не дает компилятору применить дальнейшие оптимизации.

```
int main(int argc, const char* argv[]) {

  int nth = (argc>1) ? atoi(argv[1]) : 4;
  size_t vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  float alpha = 0.5;
  tbb::task_scheduler_init init{nth};

  std::unique_ptr<double[]> A{new double[vsize]};
  std::unique_ptr<double[]> B{new double[vsize]};
  std::unique_ptr<double[]> C{new double[vsize]};

  for(size_t i = 0; i < vsize; i++){
    A[i] = B[i] = i;
  }

  auto t=tbb::tick_count::now();
  tbb::parallel_for(tbb::blocked_range<size_t>{0, vsize},
    [&](const tbb::blocked_range<size_t>& r){
      for (size_t i = r.begin(); i < r.end(); ++i)
        C[i] = A[i] + alpha * B[i];
    });
  double ts = (tbb::tick_count::now() - t).seconds();

  std::cout << "Time: " << ts << " seconds; ";
  std::cout << "Bandwidth: "  << vsize*24/ts/1000000.0 << "MB/s\n";
  return 0;
}
```

Рис. 20.5 ❖ Базовый код, который мы постараемся оценить и улучшить

Мастерство размещения данных и привязки к процессору

Привязка данных и вычислений – дело отнюдь не тривиальное. Главным образом потому, что зависит от операционной системы и ее вызовов. В Linux низкоуровневый интерфейс предоставляет библиотека `libnuma`[1], в которую входят функции для управления стратегиями размещения данных и привязки к процессору, реализованными в ядре ОС. Альтернативу более высокого уровня дает команда `numactl`[2], которая решает ту же проблему, но менее гибко.

Однако сводить на нет переносимость TBB-приложения, соединив его с системно зависимой библиотекой NUMA, – не лучшая идея. Переносимую и широко используемую альтернативу дает уже упомянутая выше библиотека `hwloc`. В настоящее время TBB не предлагает собственный API для управления локальностью NUMA, но, как мы увидим ниже, можно принять кое-какие меры, чтобы задачи TBB обращались к локальным данным, когда это возможно. На момент написания книги ручное управление размещением данных и привязкой к процессору приходится осуществлять посредством сторонней библиотеки, и в этой главе мы без ограничения общности прибегнем к услугам `hwloc`. Существуют версии этой библиотеки для Windows, MacOS и Linux (на самом деле в Linux `hwloc` реализована поверх `numactl/libnuma`).

В примере на рис. 20.6 мы запрашиваем количество узлов NUMA, а затем выделяем память для данных на каждом узле, чтобы впоследствии создать по одному потоку на узел и связать его с соответствующим доменом. Используется версия библиотеки `hwloc 2.0.1`.

Общий аргумент всех функций семейства `hwloc` – топология объекта, в нашем примере – `topo`. Этот объект сначала инициализируется, а затем в него загружается информация о платформе. После этого мы можем получать сведения из структуры данных `topo`, например с помощью функции `hwloc_get_nbobjs_by_type`, которая возвращает количество узлов NUMA, если второй аргумент равен `HWLOC_OBJ_NUMANODE` (возможны и другие значения, например `HWLOC_OBJ_CORE` или `HWLOC_OBJ_PU` – соответственно количество логических ядер и обрабатывающих блоков). Это количество узлов NUMA хранится в переменной `num_nodes`.

Далее создается массив, содержащий `num_nodes` указателей на `double`, который инициализируется в функции `alloc_mem_per_node`. В результате обращения к функции `alloc_thr_per_node` создается `num_nodes` потоков, каждый из которых привязан к соответствующему узлу NUMA. Обе функции показаны на рис. 20.7 и 20.8 соответственно. И в конце примера мы освобождаем выделенную память и структуру данных `topo`.

---

1 http://man7.org/linux/man-pages/man3/numa.3.html.

2 http://man7.org/linux/man-pages/man8/numactl.8.html.

```
#include <hwloc.h>
…
int main(void)
{
  hwloc_topology_t topo;
  hwloc_topology_init(&topo);
  hwloc_topology_load(topo);

  //* Напечатать количество узлов NUMA.
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::cout << "There are " << num_nodes << " NUMA node(s)\n";

  double ** data = new double*[num_nodes];
  //* Выделить память на каждом узле NUMA
  long size = 1024*1024;
  alloc_mem_per_node(topo, data, size);

  sout_t sout;
  //* Один мастер-поток на узел NUMA
  alloc_thr_per_node(topo, sout);

  for (auto &s : sout) {
    std::cout << s.str();
  }
  //* Освободить выделенную память и топологию
  for(int i = 0; i < num_nodes; i++){
    hwloc_free(topo, data[i], size);
  }
  hwloc_topology_destroy(topo);
  delete [] data;
  return 0;
}
```

Рис. 20.6 ❖ Использование hwloc
для выделения памяти и привязки потоков к каждому узлу NUMA

На рис. 20.7 приведена реализация функции alloc_mem_per_node. Основная операция в ней – вызов функции hwloc_get_obj_by_type, которая возвращает описатель i-го объекта узла NUMA numa_node, если второй и третий аргументы равны HWLOC_OBJ_NUMANODE и i соответственно. У объекта numa_node имеется несколько атрибутов, в т. ч. numa_node->cpuset (битовая маска, показывающая, какие логические ядра входят в состав узла) и numa_node->nodeset (аналогичная битовая маска, идентифицирующая сам узел). Функция hwloc_bitmap_asprintf полезна для преобразования этой информации в строки, это будет показано ниже в выдаче программы. Пользуясь битовой маской nodeset, мы можем выделить память в узле с помощью функции hwloc_alloc_membind.

```
void alloc_mem_per_node(hwloc_topology_t topo,
                        double **data,
                        long size){
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  for(int i = 0; i < num_nodes; i++){ //для каждого узла NUMA
    hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                                  HWLOC_OBJ_NUMANODE, i);
    char *s;
    hwloc_bitmap_asprintf(&s, numa_node->cpuset);

    std::cout<<"NUMA node " << i << " has cpu bitmask: " << s <<'\n';
    free(s);
    hwloc_bitmap_asprintf(&s, numa_node->nodeset);
    std::cout << "Allocate data on node " << i
              << " with node bitmask " << s << '\n';
    free(s);

    data[i] = (double *) hwloc_alloc_membind(topo,
      size*sizeof(double), numa_node->nodeset,
      HWLOC_MEMBIND_BIND, HWLOC_MEMBIND_BYNODESET);
  }
}
```

Рис. 20.7 ❖ Функция выделяет память для массива указателей на double, по одному для каждого узла NUMA

Ниже показано, что было напечатано на машине yuca в момент возврата управления из этой функции.

```
There are 4 NUMA node(s)
NUMA node 0 has cpu bitmask: 0x000000ff,0x000000ff
Allocate data on node 0 with node bitmask 0x00000001
NUMA node 1 has cpu bitmask: 0x0000ff00,0x0000ff00
Allocate data on node 1 with node bitmask 0x00000002
NUMA node 2 has cpu bitmask: 0x00ff0000,0x00ff0000
Allocate data on node 2 with node bitmask 0x00000004
NUMA node 3 has cpu bitmask: 0xff000000,0xff000000
Allocate data on node 3 with node bitmask 0x00000008
```

В частности, мы видим маски cpuset и nodeset для каждого узла NUMA. Взглянув еще раз на рис. 20.3, мы увидим, что в узле 0 имеется 8 ядер с 16 логическими ядрами, пронумерованными от 0 до 7 и от 32 до 39; они представлены битовой маской 0x000000ff, 0x000000ff. Обратите внимание на запятую, разделяющую два набора логических ядер, размещенных в восьми физических. Для сравнения с платформой, на которой гипертрединг выключен, приведем соответствующий результат на машине aloe:

```
There are 2 NUMA node(s)
NUMA node 0 has cpu bitmask: 0x0000ffff
```

```
Allocate data on node 0 with node bitmask 0x00000001
NUMA node 1 has cpu bitmask: 0xffff0000
Allocate data on node 1 with node bitmask 0x00000002
```

На рис. 20.8 показана функция `alloc_thr_per_node`, которая запускает столько потоков, сколько имеется узлов NUMA, а затем привязывает их, применяя маску `cpuset`.

```
using sout_t = tbb::enumerable_thread_specific<std::stringstream>;
void alloc_thr_per_node(hwloc_topology_t topo, sout_t& sout){
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::vector<std::thread> vth;
  for(int i = 0; i < num_nodes; i++){
    vth.push_back(std::thread{[i, num_nodes, &topo, &sout]()
    {
      int err;
      sout.local() << "I'm masterThread: " << i << " out of "
                   << num_nodes << '\n';
      sout.local() << "Before: Thread: " << i
                   << " with tid " << std::this_thread::get_id()
                   << " on core " << sched_getcpu() << '\n';

      hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                          HWLOC_OBJ_NUMANODE,i);
      err = hwloc_set_cpubind(topo, numa_node->cpuset,
                              HWLOC_CPUBIND_THREAD);
      assert(!err);
      sout.local() << "After: Thread: " << i
                   << " with tid " << std::this_thread::get_id()
                   << " on core " << sched_getcpu() << '\n';
    }});
  }
  for(auto &th: vth) th.join();
}
```

Рис. 20.8 ❖ Функция создает и привязывает поток к узлу NUMA

Эта функция также запрашивает количество узлов NUMA, `num_nodes`, чтобы затем выполнить ровно столько итераций цикла, создающего потоки. В лямбда-выражении, выполняемом каждым потоком, мы вызываем `hwloc_set_cpubind`, чтобы привязать поток к конкретному узлу NUMA, пользуясь маской `numa_node->cpuset`. Чтобы убедиться в правильности привязки, мы печатаем идентификатор потока (получая его с помощью `std::this_thread::get_id`) и идентификатор логического ядра, на котором работает поток (его возвращает `sched_getcpu`). Ниже, а также на рис. 20.9 показан результат, полученный на машине yuca.

```
Before: Thread 0 with tid 873342720 on core 33
After: Thread 0 with tid 873342720 on core 33
Before: Thread 1 with tid 864950016 on core 2
After: Thread 1 with tid 864950016 on core 8
```

```
Before: Thread 2 with tid 856557312 on core 33
After: Thread 2 with tid 856557312 on core 16
Before: Thread 3 with tid 848164608 on core 5
After: Thread 3 with tid 848164608 on core 24
```

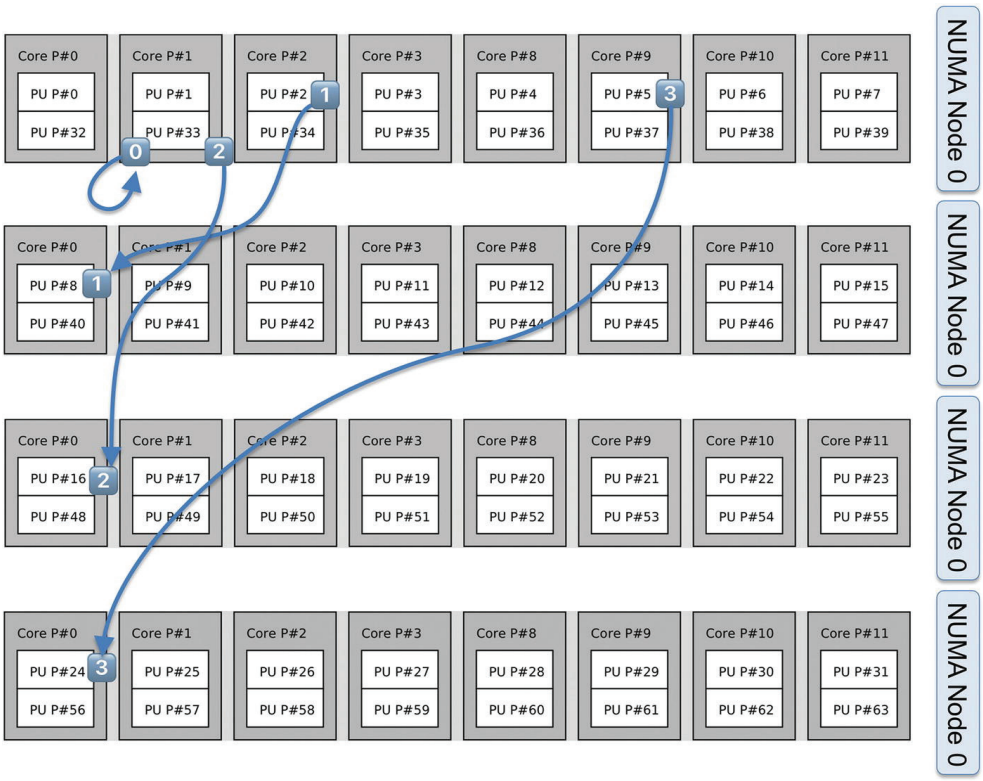


Рис. 20.9 ❖ Изображение перемещения потоков вследствие привязки к узлам NUMA на yuca

Здесь следует отметить две вещи. Во-первых, ОС первоначально создает потоки на логических ядрах одного и того же узла NUMA, потому что предполагается, что они будут сотрудничать. Потоки 0 и 2 даже созданы на одном логическом ядре. Во-вторых, потоки привязаны не к одному ядру, а к целому набору ядер, принадлежащих одному узлу NUMA. Это дает ОС некоторую свободу, если она сочтет более правильным переместить поток на другое ядро в том же узле. И для полноты картины приведен результат работы той же программы на машине aloe:

```
Before: Thread: 0 with tid 140117643171584 on core 3
After: Thread: 0 with tid 140117643171584 on core 3
Before: Thread: 1 with tid 140117634778880 on core 3
After: Thread: 1 with tid 140117634778880 on core 16
```

У пакетов `hwloc` и `likwid` много возможностей, о которых интересующийся читатель может узнать из документации и онлайновых пособий. Но и того,

о чем мы рассказали в этом разделе, хватит, чтобы двинуться дальше и, засучив рукава, заняться версией тройственной функции, учитывающей топологию NUMA.

Привлекаем hwloc и TBB к совместной работе

Очевидно, что наша главная цель – минимизировать количество нелокальных операций доступа, а значит, вычисления нужно производить на ядрах, ближайших к той памяти, где хранятся данные. Простой подход – вручную распределить данные по узлам NUMA и разрешить обработку данных только потокам, работающим в тех же узлах. В педагогических целях мы сначала опишем это решение, а в следующем разделе дадим обзор более продвинутых альтернатив.

Для размещения данных и привязки к процессорам мы могли бы опереться на `hwloc` API, но нам нужна реализация тройственной функции средствами TBB и при этом с учетом NUMA. Раз так, то потоками будет управлять планировщик TBB. Из главы 11 мы знаем, что потоки создаются в функции `tbb::task_scheduler_init`. Эта же функция создает арену по умолчанию с достаточным количеством слотов, чтобы потоки могли принять участие в выполнении задач. В нашей эталонной реализации тройственной операции (рис. 20.5) за распределение пространства итераций между задачами отвечал алгоритм `parallel_for`. Все потоки участвуют в выполнении этих задач независимо от порции итераций, которую задача обрабатывает, и от ядра, на котором работает поток. Но на платформе NUMA нам нужно совсем другое, верно?

Простейшая альтернатива – улучшить эталонную реализацию, осуществив следующих три шага:

- распределить векторы `A`, `B` и `C`, участвующие в тройственной операции, по разным узлам NUMA. Чтобы не усложнять, ограничимся пока статическим блочным разбиением. На машине yuca это означает, что на каждом из четырех узлов будет выделена память для четырех больших порций `A`, `B` и `C`;
- создать мастер-поток на каждом узле NUMA. Каждый мастер-поток создаст собственную арену задач и собственный локальный экземпляр `task_scheduler_observer`. Затем каждый мастер-поток выполнит собственный алгоритм `tbb::parallel_for`, чтобы обработать части `A`, `B` и `C`, соответствующие данному узлу NUMA;
- автоматически привязывать потоки, заходящие на каждую арену, к соответствующему узлу NUMA. Об этом позаботится локальный `task_scheduler_observer`, который мы создадим для каждой арены.

Рассмотрим реализацию каждого пункта этого плана. Функцию `main` нужно лишь немного модифицировать по сравнению с примером на рис. 20.6. На рис. 20.10 показаны новые строки, а тем, что не изменились, соответствует многоточие (...).

Аргумент программы `thds_per_node` позволяет экспериментировать с количеством потоков в каждом узле NUMA. Как и на рис. 20.6, `num_nodes` – количество узлов NUMA, которое мы получаем с помощью `hwloc` API. Поэтому мы передаем конструктору планировщика TBB `(thds_per_node-1)*(num_nodes)`, а не `thds_`

per_node*num_nodes узлов, т. к. будем явно создавать дополнительные num_nodes мастер-потоков внутри функции alloc_thr_per_node.

```
int main(int argc, char** argv)
{
  int thds_per_node = (argc>1) ? atoi(argv[1]) : 4;
  size_t vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  …
  double **data = new double*[num_nodes];
  times = std::vector<double>(num_nodes);
  //* Выделить память в каждом узле NUMA
  long doubles_per_node = vsize*3/num_nodes;
  alloc_mem_per_node(topo, data, doubles_per_node);

  //* Один мастер-поток на каждый узел NUMA
  tbb::task_scheduler_init init{(thds_per_node-1)*num_nodes};
  auto t = tbb::tick_count::now();
  alloc_thr_per_node(topo, data, vsize/num_nodes, thds_per_node);
  double ts = (tbb::tick_count::now() - t).seconds();
  …
  delete [] data;
  return 0;
}
```

Рис. 20.10 ❖ Функция main из реализации тройственной операции с учетом топологии NUMA

Функция alloc_mem_per_node по существу такая же, как на рис. 20.7, но теперь она вызывается с другим аргументом size: doubles_per_node = vsize*3/num_nodes, где vsize – размер всех трех векторов, т. е. общее количество чисел типа double умножается на 3, но делится на количество узлов для реализации блочного разбиения. Для чистоты предположим, что vsize нацело делится на num_nodes. В конце alloc_mem_per_node элемент data[i] указывает на данные, размещенные в i-м узле NUMA.

На рис. 20.11 показаны отличия адаптированной версии функции alloc_thr_per_node. Теперь она получает описатель данных, размер lsize локальных векторов, которые будут обходиться в каждом узле, и заданное пользователем количество потоков в узле, thds_per_node.

Обратите внимание, что в коде на рис. 20.11 внутри цикла по i, который обходит num_nodes узлов, есть три вложенных лямбда-выражения: (1) для объекта потока; (2) для функции-члена task_arena::execute; (3) для алгоритма parallel_for. Во внешнем лямбда-выражении мы сначала привязываем поток к соответствующему узлу NUMA, i.

На втором шаге инициализируются указатели на массивы A, B и C, память для которых была выделена в массиве data[i]. На рис. 20.10 мы передаем функции alloc_thr_per_node в качестве третьего аргумента vsize/num_nodes, потому что в каждом узле обходится только одна порция каждого из трех массивов. Поэтому аргумент функции lsize = vsize/num_nodes, он и используется в цикле, где инициализируются массивы A и B, и передается в качестве аргумента алгоритму parallel_for, вычисляющему C.

```
void alloc_thr_per_node(hwloc_topology_t topo, double** data,
                        size_t lsize, int thds_per_node){
  float alpha = 0.5;
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::vector<std::thread> vth;
  for(int i = 0; i < num_nodes; i++){
    vth.push_back(std::thread{
      [=,&topo](){
        hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                              HWLOC_OBJ_NUMANODE,i);
        int err = hwloc_set_cpubind(topo, numa_node->cpuset,
                                    HWLOC_CPUBIND_THREAD);
        assert(!err);
        double *A = data[i];
        double *B = data[i] + lsize;
        double *C = data[i] + 2*lsize;

        for(size_t j = 0; j < lsize; j++){
          A[j] = B[j] = j;
        }
        tbb::task_arena numa_arena{thds_per_node};
        PinningObserver p{numa_arena, topo, i, thds_per_node};
        auto t = tbb::tick_count::now();
        numa_arena.execute([&](){
          tbb::parallel_for(tbb::blocked_range<size_t>{0, lsize},
            [&](const tbb::blocked_range<size_t>& r){
              for (size_t i = r.begin(); i < r.end(); ++i)
                C[i] = A[i] + alpha * B[i];
            });
        });
        double ts = (tbb::tick_count::now() - t).seconds();
        times[i] = ts;
      }
    });
  }
  for (auto &th: vth) th.join();
}
```

Рис. 20.11 ❖ Функция создает по одному потоку на узел для вычисления тройственной функции в каждом узле NUMA

Далее мы инициализируем арену в узле NUMA, `numa_arena`, которая впоследствии будет передана в качестве аргумента объекту `task_scheduler_observer p` и использована при вызове `parallel_for`, ограниченного этой ареной (с помощью `numa_arena.execute`). Это и является ключом к реализации тройственной операции с учетом NUMA. Алгоритм `parallel_for` создаст задачи, которые обходят локальные порции всех трех векторов. Эти задачи будут исполняться потоками, работающими на ядрах того же узла NUMA. Но пока что мы имеем только `thds_per_node*num_nodes` потоков, из которых `num_nodes` были явно запущены как

мастера и привязаны к разным узлам NUMA, а остальные отпущены в вольное плавание и могут работать где угодно. Потоки, находящиеся в глобальном пуле, будут заходить на каждую из num_nodes арен. Мы позаботились о том, чтобы при инициализации каждой арены numa_arena завести thds_per_node слотов, один из которых уже занят мастер-потоком, а остальные оставлены для рабочих потоков. Теперь наша цель состоит в том, чтобы привязать первые thds_per_node-1 потоков, заходящих на каждую арену numa_arena, к соответствующему узлу NUMA. Для этого создадим класс PinningObserver (производный от task_scheduler_observer) и сконструируем объект этого класса p, передав конструктору четыре аргумента: PinningObserver p{numa_arena, topo, i, thds_per_node}. Напомним, что здесь i – идентификатор узла NUMA для мастер-потока i.

На рис. 20.12 показана реализация класса PinningObserver.

```
class PinningObserver : public tbb::task_scheduler_observer {
  hwloc_topology_t topo;
  hwloc_obj_t numa_node;
  int numa_id;
  int num_nodes;
  tbb::atomic<int> thds_per_node;
  tbb::atomic<int> masters_that_entered;
  tbb::atomic<int> workers_that_entered;
  tbb::atomic<int> threads_pinned;
public:
  PinningObserver(tbb::task_arena& arena, hwloc_topology_t& _topo,
                  int _numa_id, int _thds_per_node)
                  : task_scheduler_observer{arena}, topo{_topo},
                    numa_id{_numa_id}, thds_per_node{_thds_per_node}{
    num_nodes = hwloc_get_nbobjs_by_type(topo,
                                         HWLOC_OBJ_NUMANODE);
    numa_node = hwloc_get_obj_by_type(topo,
                                      HWLOC_OBJ_NUMANODE,numa_id);
    masters_that_entered = 0;
    workers_that_entered = 0;
    threads_pinned = 0;
    observe(true);
  }
  void on_scheduler_entry(bool is_worker) {
    if (is_worker) ++workers_that_entered;
    else ++masters_that_entered;
    if(--thds_per_node > 0){
      int err = hwloc_set_cpubind(topo, numa_node->cpuset,
                                  HWLOC_CPUBIND_THREAD);
      assert(!err);
      threads_pinned++;
    }
  }
};
```

Рис. 20.12 ❖ Реализация локального task_scheduler_observer для тройственной функции

С классом task_scheduler_observer мы познакомились в главе 13. В нем имеется ознакомительная возможность, позволяющая завести по одному наблюдателю для каждой арены задачи, – так называемый локальный task_scheduler_observer. При инициализации такого наблюдателя задается ссылка на арену, что и сделано в списке инициализации конструктора PinningObserver: task_scheduler_observer{arena}. В результате в момент захода потока на эту арену вызывается функция on_scheduler_entry. Конструктор класса также инициализирует количество узлов NUMA, num_nodes, и объект numa_node, который предоставляет доступ к битовой маске numa_node->cpuset. Наконец, конструктор вызывает функцию-член observe(true) и начинает следить за потоками, заходящими на арену.

Функция on_scheduler_entry запоминает, сколько потоков уже привязано к узлу numa_node, в атомарной переменной thds_per_node. Этой переменной в списке инициализации конструктора присвоено значение, равное количеству потоков на узел, заданному пользователем в первом аргументе программы. Она уменьшается на единицу, когда поток заходит на арену, а поток привязывается к данному узлу, только если thds_per_node больше нуля. Поскольку каждая арена при инициализации получила thds_per_node слотов, и один из них занят мастер-потоком, который уже привязан к узлу, то те thds_per_node - 1 потоков, которые зайдут на арену первыми, окажутся привязаны к узлу и будут работать над теми сгенерированными parallel_for задачами, которые исполняются на этой арене.

Примечание. Показанная реализация класса PinningObserver не вполне корректна. Один поток может покинуть арену и снова зайти на нее, тогда он будет закреплен дважды, но thds_per_node все равно уменьшится. Правильная реализация должна была бы проверять, не был ли зашедший на арену поток уже привязан к этой арене раньше. Но чтобы не усложнять код, мы оставляем это исправление читателю.

Теперь можно оценить пропускную способность (в МБ/с) оптимизированной для NUMA версии алгоритма на машинах yuca и aloe. Для сравнения с эталонной реализацией на рис. 20.5 зададим размеры векторов равными 10^9, а количество потоков на один узел NUMA выберем так, чтобы всего получилось 32 потока. Например, на yuca программа запускается следующим образом:

```
baseline:          ./fig_20_05 32 1000000000
NUMA conscious:    ./fig_20_10 8 1000000000
```

Результаты представлены в таблице на рис. 20.13. Они получены усреднением по 10 прогонам в ситуации, когда на yuca и aloe работал только один пользователь, т. е. платформа использовалась исключительно для экспериментов.

	Эталон	С учетом NUMA	
	Пропускная способность (МВ/с)	Пропускная способность (МВ/с)	Ускорение
yuca	10 722	18 652	1,74
aloe	38 604	59 659	1,54

Рис. 20.13 ❖ Ускорение, достигнутое в результате учета топологии NUMA

Получилось на 74 % быстрее на yuca и на 54 % на aloe! Собираетесь плюнуть на дополнительную производительность, которую можно выжать из архитектуры NUMA, если немножко потрудиться?

Чтобы продолжить исследование этого достижения, мы можем воспользоваться приложением `likwid-perfctr`, которое умеет читать аппаратные счетчики производительности. Выполнив команду `likwid-perctr -a`, мы получим список групп событий, для задания которых достаточно указать имя группы. На машине aloe `likwid` предлагает группу `NUMA`, которая собирает информацию о локальном и удаленном доступе к памяти. Чтобы измерить события этой группы для обеих реализаций – эталонной и с учетом NUMA, выполним такие команды:

```
likwid-perfctr -g NUMA ./fig_20_05 32 1000000000
likwid-perfctr -g NUMA ./fig_20_10 16 1000000000
```

В ответ будет выдано много информации о значениях некоторых счетчиков производительности на всех ядрах. Среди прочего упоминаются события

```
OFFCORE_RESPONSE_0_LOCAL_DRAM
OFFCORE_RESPONSE_1_REMOTE_DRAM
```

которые дают приблизительные сведения (поскольку основаны на выборке) об объеме данных в локальной и удаленной памяти, к которым производился доступ. Для эталонной реализации отношение объема локальных данных к объему удаленных равно всего 3.25, а для оптимизированной с учетом NUMA версии оно возрастает до 25.5. Это подтверждает тот факт, что для приложений, активно работающих с памятью, усилия, потраченные на обеспечение локальности NUMA, окупаются сторицей увеличением количества операций локального доступа, а значит, и увеличением пропускной способности.

Более сложные альтернативы

Для регулярного кода тройственной операции реализованное нами простое решение годится, но планировщик TBB с заимствованием работ вынужден балансировать нагрузку независимо на каждом узле NUMA. На машине yuca будет работать четыре алгоритма `parallel_for`, по одному на каждый узел NUMA с 8 потоками, обслуживаемыми 8 физическими ядрами. Недостаток этого простого подхода в том, что все четыре арены сконфигурированы с 8 слотами; это нормально для стационарного режима работы, но ограничивает гибкость TBB, если нагрузка не идеально сбалансирована между узлами NUMA.

Например, если один из алгоритмов `parallel_for` заканчивается первым, то все восемь потоков будут простаивать. Они вернутся в глобальный пул, но не смогут зайти на остальные три арены, занятые работой, потому что там все слоты уже заполнены. Простое решение этой проблемы – увеличить количество слотов на аренах, сохранив количество привязанных потоков равным `thds_per_node`. Тогда если какой-то `parallel_for` закончится первым, то восемь потоков, вернувшихся в глобальный пул, можно будет перераспределить между свободными слотами на трех других аренах. Отметим, что эти потоки по-

прежнему привязаны к первоначальному узлу, хотя работают теперь на другой арене другого узла и, значит, обращаются к памяти удаленно.

Можно было бы привязывать потоки, заходящие на такую расширенную арену, к соответствующему узлу NUMA, когда они занимают свободные слоты (даже если раньше они были привязаны к другому узлу). Тогда пришедшие на подмогу потоки тоже будут обращаться к локальной памяти. Однако узел может оказаться перегруженным, что обычно негативно сказывается на производительности (иначе почему бы не перегрузить все узлы NUMA с самого начала). Для каждого приложения и архитектуры нужно тщательно провести эксперименты и решить, что выгоднее: переместить поток на узел NUMA или удаленно обращаться к данным из первоначального узла. В простом и регулярном алгоритме вычисления тройственной функции ни тот, ни другой подход не даст заметного выигрыша, но в более сложных и нерегулярных случаях выигрыш возможен. Накладные расходы сопровождают не только удаленный доступ, но и перемещение потока с одной арены на другую, а также повторную привязку потока, и эти расходы необходимо амортизировать за счет улучшенной балансировки нагрузки.

Еще одна битва, в которую мы могли бы вступить, связана с разбиением данных. Мы использовали простое блочное распределение всех трех массивов, но для менее регулярных приложений, безусловно, существуют лучшие распределения. Например, вместо того чтобы заранее распределять пространство итераций между узлами NUMA, мы можем применить подход на основе управляемого планирования. Каждый мастер-поток, ведущий вычисление в узле NUMA, может получать более крупные порции пространства итераций в начале вычисления и более мелкие ближе к концу. Хитрость в том, чтобы порции были достаточно крупными для раздачи ядрам, входящим в состав узла.

Более изощренная альтернатива – обобщить систему заимствования работ на иерархию. Чтобы заимствовать работу как со своей, так и с чужой арены, можно реализовать иерархию арен. Похожая идея была реализована для Cilk Ченом и Гуо (см. раздел «Дополнительная информация»), которые предложили трехуровневый планировщик с заимствованием работ, дававший повышение производительности на 54 % по сравнению с традиционными вариантами в приложениях, активно использующих память. Отметим, что такого рода приложения больше выигрывают от локальности NUMA, чем счетные приложения. Для последних затраты на доступ к памяти обычно пренебрежимо малы по сравнению с интенсивными вычислениями. В действительности для счетных приложений увеличение сложности планировщика с целью задействовать локальность NUMA может привести к дополнительным накладным расходам, которые не окупаются.

Резюме

В этой главе мы рассмотрели несколько способов задействовать локальность NUMA, применяя комбинацию TBB со сторонними библиотеками, которые помогают реализовать управление размещением данных и привязкой к процессорам. Мы начали с изучения врага, которого вознамерились одолеть: ар-

хитектуры NUMA. Для этой цели мы познакомились с союзническими библиотеками, `hwloc` и `likwid`. Они позволяют не только запрашивать низкоуровневые детали топологии NUMA, но и управлять размещением данных и привязкой к процессорам. Мы проиллюстрировали использование некоторых функций из библиотеки `hwloc` для выделения памяти в конкретном узле NUMA, создания одного потока в каждом узле и привязки потоков к ядрам, входящим в состав узла. Располагая этими средствами, мы переписали эталонную версию алгоритма вычисления тройственной функции, обращая внимание на локальность NUMA. Простейшим решением было разбить три массива на блоки, которые выделяются в памяти разных узлов NUMA и там же обходятся. Ключом к выделению памяти и привязке потоков была библиотека `hwloc`, а классы TBB `task_arena` и `task_scheduler_observer` помогли идентифицировать потоки, входящие в конкретный узел NUMA. Это решение достаточно хорошо для такого регулярного приложения, как вычисление тройственной функции, и дает повышение производительности на 74 % и 54 % (по сравнению с эталонной версией) на двух разных платформах NUMA. Для менее регулярных и более сложных приложений в последнем разделе очерчены альтернативные подходы.

Дополнительная информация

Ниже приведены дополнительные рекомендуемые материалы по теме этой главы.

- *Christoph Lameter*. NUMA (Non-Uniform Memory Access): An Overview // ACMqueue. Vol. 11. Is. 7. 2013.
- *Ulrich Drepper*. What Every Programmer Should Know About Memory. 2017. URL: www.akkadia.org/drepper/cpumemory.pdf.
- *Quan Chen, Minyi Guo, and Haibing Guan*. LAWS: Locality-Aware Work-Stealing for Multi-socket Multi-core Architectures // International Conference on Supercomputing, ICS. 2014.

Приложение A

История и предшественники

В этом приложении мы предложим два взаимодополняющих взгляда на историю библиотеки TBB: первый – на историю ее развития, второй – на то, что ей предшествовало и стало источником идей. Мы надеемся, что вам будут интересны оба аспекта и что это поможет лучше понять, почему TBB назвали самым важным дополнением к параллельному программированию за прошедшие десять лет – с чем мы не собираемся спорить.

Десятилетие «от птенца к орлу»

В основу первой части этого приложения положен текст, написанный Джеймсом к десятой годовщине TBB (середина 2016 года).

С вашего позволения я поделюсь своими мыслями о TBB. При размышлениях о TBB мне на ум приходят четыре вещи.

1. Революция TBB внутри Intel

TBB стала нашим первым коммерчески успешным программным продуктом, для которого был раскрыт исходный код. А недавно TBB, сохраняя лидерство, перешла на модель лицензирования Apache.

Мы хотели, чтобы код TBB был открыт с самого начала, но в 2006 году, когда проект только начинался, были к этому не готовы. Проекты с открытым исходным кодом были внове как для нашей маленькой команды, так и для компании Intel в целом. Поначалу мы бросили все силы на создание крепкого продукта, с которым необходимо было выйти на рынок в середине 2006 года. А когда это случилось, мы переключили внимание на ревизию системы сборки, вычистку кода (добавление комментариев!) и десяток других вещей, без которых стыдно было бы приглашать других желающих понять наш код и предложить свои дополнения. Мы поставили целью раскрыть код в середине 2007 года. Но возникла новая проблема – TBB сразу стала хитом у пользователей. Мы не скрывали от заказчиков, что собираемся раскрыть код, но это только подстегивало их интерес к TBB. Наш успех довольно быстро стал проблемой внутри Intel, поскольку руководство задавалось вопросом: «Зачем раскрывать исходный код такого успешного продукта?» Вооружившись фактами и рисунками, подготовленными командой, я отважно изложил целый ряд причин, почему нам следует открыться. Это было ошибкой, и я не получил разрешения до конца 2006 го-

да. Пока я зализывал раны, мы пришли к выводу, что нужно доказать только одну вещь: TBB распространилась бы куда шире, если бы мы раскрыли исходный код. Ведь разработчики, принимая какую-то модель программирования, ставят на кон будущее своего кода. Для выбора модели программирования открытость, пожалуй, значит больше, чем почти для любого другого программного обеспечения. И хотя мы понимали это, нам никак не удавалось донести до руководства, что только это и имеет значение – и что только это и нужно знать, чтобы принять необходимость раскрытия кода. Твердо придерживаясь этой точки зрения, я сумел удивить старшего вице-президента по нашему направлению, которая должна была одобрить наше предложение. А удивил я ее, показав всего один листок бумаги, на котором был нарисован простой график, отражавший предполагаемое распространение TBB с раскрытием и без раскрытия кода. Мы спрогнозировали, что TBB исчезнет и уступит место чему-то другому в течение пяти лет, если наше предложение этой критической модели программирования не будет подкреплено открытым исходным кодом. Мы предсказывали большой успех, если код будет раскрыт (на самом деле, как оказалось, успех мы даже недооценили). Старший вице-президент Рени Джеймс выслушала мой двухминутный блиц-доклад, посмотрела на меня и спросила: «Почему же Вы в первый раз этого не сказали? Разумеется, мы так и должны поступить». Я мог бы ответить, что это был седьмой слайд из первоначальной слишком длинной презентации на 20 слайдов, с которой я выступал 2 месяца назад. Но ограничился простым «спасибо», а все остальное – история. Мы выбрали самую популярную на тот момент схему лицензирования продуктов с открытым исходным кодом: GPL v2 с исключением Classpath (важно для библиотек шаблонов на C++). Спустя десять лет мы перешли на лицензию Apache. Сообщество пользователей и соавторов настойчиво убеждало нас, что в наши дни это самая правильная лицензия для TBB.

2. Первая революция TBB в сфере параллелизма

Первая предложенная TBB революция в сфере параллелизма заключалась в реализации абстракции заимствования работ, а вместе с ней и полной поддержки компонуемости на C++.

Библиотека OpenMP невероятно важна, но она не поддерживает композицию. Это эпическая ошибка с далеко идущими последствиями, но изменить ничего нельзя, потому OpenMP чрезвычайно важна и обязана поддерживать совместимость. Я несу ответственность за эту ошибку библиотеки OpenMP наряду со всеми, кто принимал участие в ее создании, рецензировании и продвижении начиная с 1997 года. Мы не предвидели, насколько важным окажется вложенный параллелизм по мере развития аппаратного параллелизма. В 1997 году вопрос так просто не стоял.

Компонуемость – самое замечательное свойство TBB. Возможность не думать о перегруженности ядер, вложенном параллелизме и т. д. неизмеримо важна. TBB постепенно производит переворот в некоторых сообществах разработчиков, которым необходима компонуемость. Библиотека Intel Math Kernel Library (MKL), которая долгое время была основана на OpenMP, теперь имеет версию, надстроенную над TBB, именно по этой причине. А гораздо более

новая библиотека Intel Data Analytics Acceleration Library (DAAL) с открытым исходным кодом изначально использовала TBB и MKL на базе TBB. На самом деле TBB нашла применение даже в некоторых версиях Python.

Разумеется, сердцем TBB, истинным вершителем ее магии, является планировщик задач с заимствованием работ. В то время как пользователи HPC-продуктов (высокопроизводительных вычислений) стремятся довести производительность до высшего предела, запуская приложение на выделенных ядрах, TBB решает проблему, которая перед пользователями HPC вообще не стоит: как обеспечить хорошую работу параллельной программы, когда ядра, на которых она работает, используются кем-то еще? Представьте, что в вашем распоряжении восемь ядер, но одновременно с вашей программой на одном ядре запустился антивирус. На суперкомпьютере такого не бывает, но случается сплошь и рядом на рабочих станциях и ноутбуках! Не будь планировщик в TBB по природе своей динамичным, такая программа просто работала бы дольше на время, украденное у нее антивирусом… потому что задержку испытал бы каждый поток в приложении. А при использовании TBB на восьми ядрах вытеснение программы с одного ядра на время TIME может привести к задержке приложения всего на время TIME/8. Такая гибкость в реальном мире дорогого стоит!

Наконец, TBB – библиотека шаблонов на C++, которая с радостью приветствует введение параллелизма в сам язык C++. Приверженность TBB к C++ способствовала внесению изменений в стандарт C++. Быть может, наша величайшая мечта в том и состоит, чтобы в один прекрасный день TBB свелась к планировщику и алгоритмам, которые его используют. Многие другие вещи в TBB – помощь в распараллеливании частей STL, создании по-настоящему переносимых блокировок и атомарных переменных – призваны преодолеть недостатки в механизмах выделения памяти и других средствах, предназначенных для внесения параллелизма в C++. В конечном итоге они могут и должны стать частью стандарта языка. А может быть, не только они, но и другие элементы TBB? Время покажет.

3. Вторая революция TBB в сфере параллелизма

Вторая революция в сфере параллелизма, совершенная TBB, – превосходные альтернативы пакетному синхронному программированию.

Как бы высоко мы ни ценили планировщик TBB с заимствованием работ, чаще всего в приложениях встречаются алгоритмы, требующие большого объема синхронизации во время выполнения. Это знамение времени, показывающее, насколько успешным было параллельное программирование в прошлые годы. Но с ростом степени параллелизма это стало серьезным препятствием на пути к масштабируемости. Куда лучше было бы описать поток данных и требовать гораздо меньшего уровня синхронизации. Потоковые графы, входящие в состав TBB, занимают место во главе этой новой революции в параллельном программировании. Такой тип мышления необходим для любой модели параллельного программирования, рассчитанной на будущее.

4. Птички TBB

Совсем в другой плоскости лежит вопрос о птичках, которых мы использовали. Моя оригинальная книга по TBB вышла в издательстве O'Reilly в 2007 году, и на ее обложке по традиции было изображено животное. Издательство ясно дало мне понять, что они сами выберут животное (процедура выбора покрыта завесой тайны). Но я не смутился и предложил несколько животных, которые, как мне казалось, имели отношение к теме. O'Reilly выбрало прекрасно нарисованную канарейку – симпатичную птичку, которую я даже не рассматривал. На вкус и на цвет товарища нет, но вскоре мы стали агитировать Intel «принять птичку». Мы благодарны Белинде Эдкинсон за это переосмысление и за популярную не ущемляющую ничьих прав «певунью», которая изображена на наших футболках, стикерах и сайтах. Жизнерадостная птичка стала талисманом TBB. Мы «приняли птичку».

На задней стороне обложке оригинальной книги было написано:

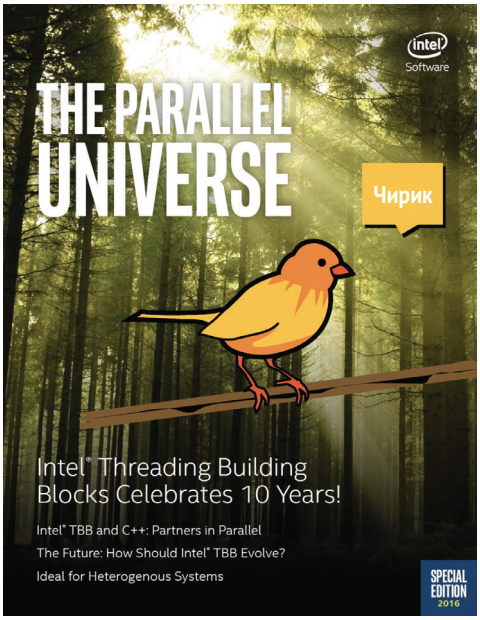
На обложке книги «Intel Threading Building Blocks» изображена дикая канарейка (Serinus canaria), небольшая певчая птица из семейства вьюрков. Ее еще называют островной, или атлантической, канарейкой, потому что она обитает на островах, расположенных к западу от Европы: на Мадейре, Азорских и Канарских островах, в честь которых она и получила название. Название происходит от латинского слова *canaria* (собачий), впервые употребленного Плинием Старшим в его «Естественной истории» из-за крупных собак, бродивших по островам. Канарейки живут в садах, на приусадебных участках и в рощах, гнезда вьют в кустах и на деревьях.

Дикая канарейка темнее и чуть больше домашней, но в остальном они похожи. Грудка желто-зеленого цвета, на спинке коричневая полоска. Как и у многих других видов, самец окрашен ярче самки. Кроме того, самец мелодичнее поет. Когда в XV веке испанцы завоевали острова, они одомашнили птиц и начали разводить их. К XVI веку канарейки высоко ценились как домашние питомцы по всей Европе. (Сэмюэл Пипис пишет о «птицах канарейках» в дневнике за 1661 год.) В результате 500 лет селекции было выведено много разновидностей канареек, в т. ч. и повсеместно встречающаяся сегодня желтая. Маленькие птички популярны в качестве домашних животных, потому что могут жить до 10 лет, почти не требуют внимания и считаются самыми мелодичными певчими птицами.

В 1980-х годах шахтеры использовали канареек в качестве сигнальной системы, по две птицы в каждой шахте. По данным Горнорудного управления США, канарейки предпочтительнее мышей, потому что чувствительнее к выделяющимся газам, и их реакция на присутствие газов более выражена. В шахте канарейка будет петь весь день, но как только концентрация угарного газа поднимается, перестает петь и сначала раскачивается на своей жердочке, а потом умирает – это знак, что шахтерам нужно поскорее убираться.

Я написал книгу о TBB весной 2007 года, пользуясь помощью всей команды разработчиков. Я бы очень хотел, чтобы в будущем мы написали новую книгу на эту тему. В этом году лично у меня нет на это времени, но если бы нашлось достаточно желающих помочь… что ж, я думаю, мы могли бы что-нибудь придумать. Я открыт для предложений. *Майкл и Рафаэль, спасибо, что вы откликнулись, так что сначала мы вместе написали пособие по TBB в 2017 году, а в 2018-м начали работать над этой книгой!*

Джеймс далее приглашал народ принять участие в подготовке специального выпуска журнала «The Parallel Universe», планируемого компанией Intel к десятилетию TBB. Он был опубликован как раз тогда, когда Джеймс ушел из штата Intel на неполный рабочий день (и стал более занятым, чем когда-либо прежде). Специальный выпуск доступен бесплатно по адресу https://software.intel.com/parallel-universe-magazine. В статье «The Genesis and Evolution of Intel Threading Building Blocks» можно подробнее прочитать об истории TBB, рассказанной ее первым архитектором, в т. ч. про две вещи в первоначальном проекте TBB 1.0, о которых он «сожалеет». За многие годы в этом журнале было опубликовано немало интересных статей о TBB.



К десятилетию TBB –
специальный выпуск журнала Intel «The Parallel Universe»

Источники идей TBB

В основу второй части этого приложения положены исторические замечания Джеймса, написанные для первой книги о TBB (2007 год), когда библиотеке исполнился всего один год. Это взгляд на то, что предшествовало TBB и стало источником идей для нее.

Сам Джеймс поначалу назвал эту главу книги «Эпилог» и ссылался на нее как на библиографию. Редактор из издательства O'Reilly Энди Орам (Andy Oram) – которого Джеймс называет лучшим редактором на планете – поумерил энтузиазм Джеймса по поводу уместности включения в техническую книгу библиографии или эпилога и переименовал эту часть в главу 12 под названием «История и родственные проекты».

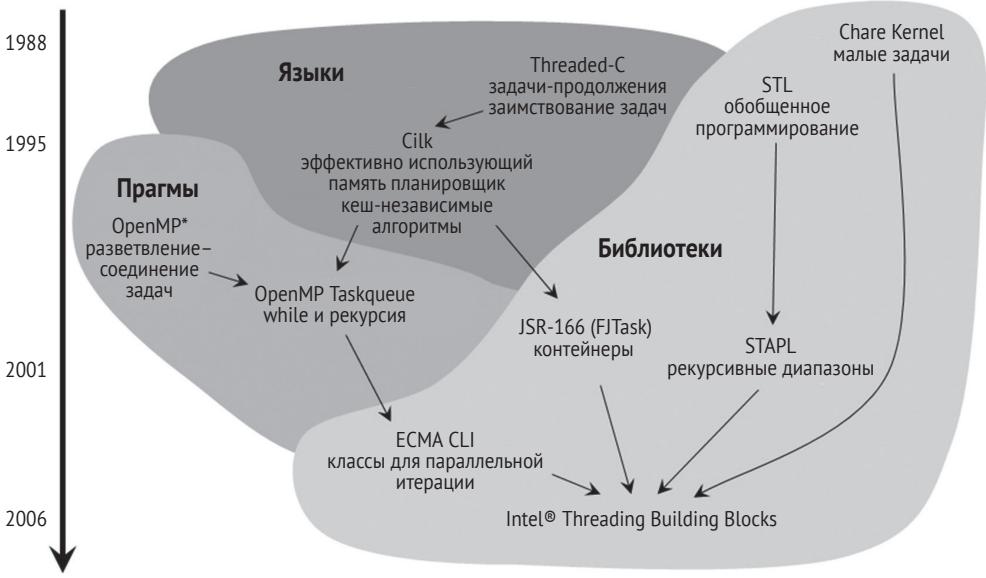


Рис. А.1 ❖ Проекты, сильнее всего повлиявшие на дизайн библиотеки Threading Building Blocks

Ничто из написанного ниже не обязательно, чтобы научиться использовать Threading Building Blocks. Мы лишь рассматриваем некоторые источники, повлиявшие на наши размышления во время работы в Intel и положенные в основу проекта и реализации библиотеки Threading Building Blocks. Список статей и книг в конце главы можно рассматривать как библиографию и рекомендации для дальнейшего чтения.

Примечание. Первоначально эта глава включала краткое введение в лямбда-функции, за включение которых в C++ ратовал Арч Робисон (Arch Robison), ведущий разработчик Threading Building Blocks. Мы убрали его, потому что спустя несколько лет лямбда-выражения таки были включены в стандарт C++11. Это упростило синтаксис TBB, а значит, и ее изучение – на что Арч и рассчитывал. Конечно, из-за этого оригинальная книга по

ТВВ кажется несколько устаревшей – ведь в примерах, опубликованных в 2007 году, C++11 никак не мог использоваться.

Информация, приведенная в этой библиографии, пожалуй, придется больше по вкусу тем, кто рассчитывает внести свой вклад в ТВВ. Проект ТВВ предлагает много пищи для размышлений, и эта глава показывает, с чего начать.

ТВВ заимствует идеи из многих источников. На рис. А.1 представлены основные проекты примерно за десять предыдущих лет, оказавшие влияние на ТВВ. Все они послужили лишь источниками идей и, если не считать McRT-Malloc, не имеют прямой связи с исходным кодом. Изучение предшественников за пределами 1988 года оставим историкам.

Библиотека ТВВ уникальная, потому что в основе ее проекта лежит несколько ключевых решений:

- поддержка C++-программ общего вида с помощью существующих компиляторов;
- ослабленное последовательное выполнение;
- использование рекурсивного параллелизма и обобщенных алгоритмов;
- использование заимствования задач.

Модель ослабленного последовательного выполнения

В ТВВ реализована модель ослабленного последовательного выполнения. Слово «ослабленное» означает, что последовательные программы чрезмерно ограничены неявными последовательными зависимостями (например, от регистра счетчика команд) и что конкурентная библиотека вносит столько параллелизма, сколько возможно, не мешая программе работать последовательно.

Мы можем считать, что эта модель ослаблена настолько, насколько возможно, чтобы сохранить способность к корректному выполнению в одном потоке. Это цель!

Способность выполнять программу последовательно дает гигантское преимущество во время отладки приложений. Она позволяет отлаживать типичные программные ошибки, прежде чем придется столкнуться с проблемами конкурентности, подлежащими отладке. Наш совет простой: начните с отладки в последовательном режиме, а затем запускайте программу в параллельном режиме, чтобы отладить ошибки, связанные с конкурентностью. Программы, которые принципиально нуждаются в конкурентности, не дают такой возможности. Кроме того, программа, требующая конкурентности, будет испытывать проблемы с производительностью, когда количество требуемых потоков больше, чем количество аппаратных потоков, поскольку накладные расходы на квантование времени могут оказаться существенными.

Библиотеки, оказавшие влияние

1988, Chare Kernel, Иллинойский университет в Урбана-Шампейн

В 1988 году это была просто библиотека на C. Ее основная идея заключалась в том, чтобы разбить программу на небольшие участки, называемые *чарами* (chare), и дать планировщику возможность эффективно (с точки

зрения памяти и времени) распределить их по процессорам. Отображение задач на потоки вместо непосредственного программирования в терминах потоков – важная идея. Впоследствии библиотека Chare Kernel была дополнена некоторыми средствами для работы на машинах с распределенной памятью и превратилась в язык Charm++.

1993, стандартная библиотека шаблонов (STL) для C++, Hewlett-Packard

В ноябре 1993 года STL была представлена на рассмотрение комитету ANSI/ISO C++, а в 1994 году компания HP сделала ее свободно доступной. Библиотека была включена в стандарт C++. Арч Робисон рассказывал: «Я как-то слышал, что Степанов прочел большую лекцию по обобщенному программированию, в которой рассказывал о том, как он написал по-настоящему общий алгоритм нахождения наибольшего общего делителя. [В печатной работе излагается тот же материал, но с большим вниманием к математическим деталям.] Во всем своем великолепии обобщенное программирование – это не только про параметрические типы, но и про программирование с помощью концепций». Работы Степанова по STL и обобщенному программированию перечислены ниже в этой главе. *Примечание: Александр Степанов любезно согласился написать предисловие к оригинальной книге, в котором похвалил разработчиков за включение обобщенного программирования в проект TBB.*

1999, запрос по спецификации Java #166 (JSR-166), Дуг Леа

Стандартизация случилась позже, но именно в 1999-м Леа впервые поднял эту тему. FJTask была попыткой включить параллелизм в духе Cilk в стандартную библиотеку Java. Запрос имел номер JSR-166, но в стандарт не вошел.

2001, Standard Template Adaptive Parallel Library (STAPL), Texas A&M

В STAPL было введено понятие рекурсивных параллельных диапазонов (pRanges) и предложено использовать эти диапазоны вместо итераторов для связывания обобщенных параллельных алгоритмов с параллельными контейнерами. В STL нет понятия рекурсивного диапазона. STAPL сложнее TBB, потому что охватывает архитектуры с распределенной памятью, типичные для высокопроизводительных вычислений (HPC). Кроме того, STAPL поддерживает спецификацию произвольного порядка выполнения для графов параллельных задач. Это дает возможность использовать несколько стратегий планирования для оптимизации времени выполнения.

2004, ECMA CLI, параллельный профиль, Intel

В этой спецификации ECMA виртуальной машины .NET имеются классы для параллельного итерирования, спроектированные Арчем Робисоном.

2006, McRT-Malloc, Intel Research

Масштабируемый транзакционный распределитель памяти McRT лег в основу масштабируемого распределителя памяти, вошедшего в состав TBB. В разделах 3 и 3.1 написанной в 2006 году статьи Хадсона, Саха, Адл-

Табатабаи и Херцберга (http://doi.acm.org/10.1145/1133956.1133967) в общих чертах описано, что представляет собой масштабируемый распределитель памяти в TBB.

Языки, оказавшие влияние

1994, Threaded-C, Массачусетский технологический институт

Параллельная машина продолжений (The Parallel Continuation Machine – PCM) входила в среду выполнения языка Threaded-C. Это был основанный на C пакет, положенный в основу потоков с передачей управления через продолжение в суперкомпьютере Connection Machine Model CM-5 корпорации Thinking Machines Corporation. Для улучшения балансировки нагрузки и локальности вычислений использовалось заимствование работ как общая стратегия планирования. Не следует путать этот язык с языком Threaded-C for EARTH, разработанным в университете Макгилла и Делаварском университете. PCM кратко упоминается на стр. 2 оригинального руководства по Cilk.

1995, Cilk, Массачусетский технологический институт

Первая реализация Cilk была прямым преемником PCM/Threaded-C. В Cilk были устранены трудности программирования задач-продолжений и предложены методы, позволяющие адаптировать создание задач к кешам, ничего не зная о размере кешей, – кеш-независимые алгоритмы. Cilk является расширением C, ориентированным на очень эффективную поддержку параллелизма типа разветвление–соединение. Его эффективность с точки зрения использования памяти обсуждается в статье http://supertech.csail.mit.edu/papers/cilkjpdc96.pdf. Библиотека быстрого дискретного преобразования Фурье FFTW (www.fftw.org) – пример кеш-независимого алгоритма.

Прагмы, оказавшие влияние

1997, OpenMP, консорциум крупных поставщиков оборудования и программного обеспечения

OpenMP поддерживает многоплатформенное параллельное программирование с разделяемой памятью на языках C и Fortran благодаря стандартному набору директив компилятора, библиотечных подпрограмм и переменных среды. До появления OpenMP многие производители компиляторов добавляли собственные прагмы, преследующие одну цель, но несовместимые между собой. В OpenMP воплощена философия разветвления–соединения. См. www.openmp.org.

1998, OpenMP Taskqueue, Kuck & Associates (KAI)

Предложенные расширения OpenMP, предназначенные для выхода за пределы циклов.

Уточненная редакция этого предложения была утверждена и включена в OpenMP в 2008 году (через год после выхода нашей книги). Она вошла в версию OpenMP 3.0.

Влияние обобщенного программирования

Бьярн Страуструп, создатель C++, когда-то сказал, что C++ поддерживает три основных стиля программирования: *процедурное программирование*, *абстрагирование данных* и *объектно-ориентированное программирование*, но позже добавил четвертый стиль: *обобщенное программирование*. За популяризацию этого стиля мы отдаем должное стандартной библиотеке шаблонов (Standard Template Library – STL), созданной Александром Степановым. Она хорошо ложится на принципы C++, в которых ценится сочетание абстрактности и эффективности.

В STL и TBB алгоритмы отделены от контейнеров. Это означает, что алгоритм принимает рекурсивный диапазон и использует его для доступа к элементам в контейнере. Точный тип контейнера алгоритму неизвестен. Такое четкое разделение между контейнерами и алгоритмами – основная идея обобщенного программирования. Поскольку алгоритмы и контейнеры разделены, конкретизация шаблона добавляет сравнительно немного кода, и, как правило, только такого, который реально будет использоваться.

Библиотека TBB базируется на тех же принципах, что STL, но в ней используются *рекурсивные диапазоны*, а не *итераторы*. Итераторы в STL (за исключением итераторов произвольного доступа) принципиально последовательные, поэтому для выражения параллелизма они непригодны.

Примечание. Расширения C++ для диапазонов стали технической спецификацией (ISO/IEC TS 21425:2017) и почти наверняка войдут в C++20. Диапазоны предпочтительнее итераторов, в частности из-за последовательной природы итераторов, почему они и были отвергнуты при проектировании TBB в 2006 году.

Учет кешей

При проектировании TBB кеши принимались во внимание, и предприняты усилия, чтобы ограничить ненужное перемещение задач и данных. Если задачу необходимо переместить на другое процессорное ядро для исполнения, то TBB выбирает ту задачу, для которой вероятность нахождения данных в кеше процессорного ядра, с которого задача перемещается, минимальна.

Интересно отметить, что параллельная быстрая сортировка – пример алгоритма, в котором кеши важнее *максимального параллелизма*. В параллельном алгоритме сортировки слиянием степень параллелизма выше, чем в параллельной быстрой сортировке. Но параллельная сортировка слиянием производится не на месте, поэтому она занимает в два раза больше места в кеше, чем быстрая сортировка. Поэтому на практике быстрая сортировка обычно работает быстрее.

Обдумывая структуру программы, принимайте во внимание локальность данных. Избегайте спорадического использования областей данных, если при-

ложение можно спроектировать так, чтобы один набор данных использовался в четко разграниченные интервалы времени. Это происходит естественно, если применить декомпозицию данных, особенно на верхних уровнях программы.

Учет стоимости квантования времени

Квантование времени рассчитано больше на логические, чем на физические потоки. Каждый логический поток обслуживается физическим потоком в течение кванта времени – короткого, определяемого операционной системой периода, по истечении которого вытесняется. Если поток работает дольше кванта времени, а так чаще всего и бывает, то он уступает физический поток и снова ждет своей очереди. В этом разделе подробно рассматриваются затраты, связанные с квантованием времени.

Самая очевидная из них – время *контекстного переключения* между логическими потоками. При каждом контекстном переключении процессор должен сохранить все регистры предыдущего логического потока и загрузить в них информацию о следующем потоке.

Не столь очевидны затраты, связанные с «охлаждением» кеша. Процессор хранит данные, к которым недавно обращался, в кеш-памяти, которая работает очень быстро, но гораздо меньше по объему, чем основная память. Когда кеш-памяти не хватает, процессор вынужден вытеснять данные из кеша обратно в основную память. Как правило, он выбирает данные, которые давно не использовались. (В реальности множественно-ассоциативный кеш несколько сложнее, но этот раздел – не введение в технологии кеш-памяти.)

Когда логический поток получает свой квант времени и впервые обращается к элементу данных, эти данные помещаются в кеш, что занимает несколько сотен тактов. Если обращения к ним производятся достаточно часто, чтобы предотвратить вытеснение, то при каждом обращении данные будут обслужены из кеша, что занимает всего несколько тактов. Про такие данные в кеше говорят, что они *горячие*.

Квантование времени сводит этот эффект на нет, потому что если поток A исчерпал свой квант, а поток B затем начал исполняться тем же физическим потоком, то B, скорее всего, вытеснит из кеша горячие данные A, если только оба потока не работают с одними и теми же данными. Когда поток A в следующий раз получит квант времени, ему придется заново загрузить в кеш вытесненные данные, что обойдется в несколько сотен тактов на каждое непопадание в кеш. Хуже того, получив очередной квант, поток A может оказаться в ином физическом потоке, у которого вообще другой кеш.

Еще одна статья расходов – *вытеснение блокировки*. Это случается, когда поток захватывает блокировку ресурса, а его квант времени заканчивается раньше, чем он успевает ее освободить. И не важно, что поток собирался удерживать блокировку совсем недолго, теперь она останется за ним по крайней мере до тех пор, пока он снова не получит свой квант. Все остальные потоки, ожидающие этой блокировки, должны будут либо бесцельно крутиться в состоянии активного ожидания, либо уступить свой квант времени. Этот эффект называется *караваном*, потому что потоки выстраиваются «корма к носу», ожидая, когда возглавляющий караван вытесненный поток возобновит движение.

Литература для дополнительного чтения

Acar U., G. Blelloch, and R. Blumofe (2000). The Data Locality of Work Stealing // Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, 1–12.

Amdahl G. M. (1967, April). Validity of the single-processor approach to achieving large scale computing capabilities // AFIP Conference Proceedings, 30. Reston, VA: AFIPS Press, 483–485.

An P., A. Jula, et al. (2003). STAPL: An Adaptive, Generic Parallel C++ Library // Workshop on Language and Compilers for Parallel Computing, 2001. Lecture Notes in Computer Science 2624, 193–208.

Austern M. H., R. A. Towle, and A. A. Stepanov (1996). Range partition adaptors: a mechanism for parallelizing STL // ACM SIGAPP Applied Computing Review. 4, 1, 5–6.

Blumofe R. D., and D. Papadopoulos (1998). Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors.

Blumofe R. D., C. F. Joerg, et al. (1996). Cilk: An Efficient Multithreaded Runtime System // Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 207–216.

Boehm H. (2006, June). An Atomic Operations Library for C++ // C++ standards committee document N2047.

Butenhof D. R. (1997). Programming with POSIX Threads. Reading. MA: Addison Wesley.

Flynn M. J. (1972, September). Some Computer Organizations and Their Effectiveness // IEEE Transactions on Computers, C-21, 9, 948–960.

Garcia R., J. Järvi, et al. (2003, October). A Comparative Study of Language Support for Generic Programming // Proceedings of the 2003 ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications.

Gustafson J. L. (1988). Reevaluating Amdahl's Law // Communications of the ACM, 31(5), 532–533.

Halbherr M., Zhou Y., and C. F. Joerg (1994, March). MIMD-Style Parallel Programming Based on Continuation-Passing Threads // Computation Structures Group Memo 355.

Halbherr M., Y. Zhou, and C. F. Joerg (1994, September). MIMD-style parallel programming with continuation-passing threads // Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications, Capri, Italy.

Hansen B. (1973). Concurrent Programming Concepts // ACM Computing Surveys, 5, 4.

Hoare C. A. R. (1974). Monitors: An Operating System Structuring Concept // Communications of the ACM, 17, 10, 549–557.

Hudson R. L., B. Saha, et al. (2006, June). McRT-Malloc: a scalable transactional memory allocator // Proceedings of the 2006 International Symposium on Memory Management. New York: ACM Press, 74–83.

Intel Threading Building Blocks 1.0 for Windows, Linux, and Mac OS – Intel Software Network (1996).

A Formal Specification of Intel Itanium Processor Family Memory Ordering (2002, October).

ISO/IEC 14882:1998(E) International Standard (1998). Programming languages – C++. ISO/IEC, 1998.

ISO/IEC 9899:1999 International Standard (1999). Programming languages – C, ISO/IEC, 1999.

Järvi J., and B. Stroustrup (2004, September). Decltype and auto (revision 4) // C++ standards committee document N1705=04-0145.

Kapur D., D. R. Musser, and A. A. Stepanov (1981). Operators and Algebraic Structures // Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, 59–63.

MacDonald S., D. Szafron, and J. Schaeffer (2004). Rethinking the Pipeline as Object-Oriented States with Transformations // Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments.

Mahmoud Q. H. (2005, March). Concurrent Programming with J2SE 5.0 // Sun Developer Network.

Massingill B. L., T. G. Mattson, and B. A. Sanders (2005). Reengineering for Parallelism: An Entry Point for PLPP (Pattern Language for Parallel Programming) for Legacy Applications // Proceedings of the Twelfth Pattern Languages of Programs Workshop.

Mattson T. G., B. A. Sanders, and B. L. Massingill (2004). Patterns for Parallel Programming. Reading. MA: Addison Wesley.

McDowell C. E., and D. P. Helmbold (1989). Debugging Concurrent Programs // Communications of the ACM, 21, 2.

Meyers S. (1998). Effective C++. Second Edition. Reading, MA: Addison Wesley, 1998.

Musser D. R., and A. A. Stepanov (1994). Algorithm-Oriented Generic Libraries // Software – Practice and Experience, 24(7), 623–642.

Musser D. R., G. J. Derge, and A. Saini, с предисловием Александра Степанова (2001). STL Tutorial and Reference Guide. Second Edition: C++ Programming with the Standard Template Library, Boston, MA: Addison Wesley, 2001.

Narlikar G., and G. Blelloch (1999). Space-Efficient Scheduling of Nested Parallelism // ACM Transactions on Programming Languages and Systems, 21, 1, 138–173.

OpenMP C and C++ Application Program Interface. Version 2.5 (May 2005).

Ottosen T. (2006, September). Range Library Core // C++ standards committee document N2068.

Plauger P. J., M. Lee, et al. (2000). C++ Standard Template Library. Prentice Hall.

Rauchwerger L., F. Arzu, and K. Ouchi (1998, May). Standard Templates Adaptive Parallel Library // Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR), Pittsburgh, PA. Также Lecture Notes in Computer Science, 1511, Springer-Verlag, 1998, 402–410.

Robison A. D. (2006). A Proposal to Add Parallel Iteration to the Standard Library.

Robison A. (2003, April). Memory Consistency & .NET // Dr. Dobb's Journal.

Samko V. (2006, February). A proposal to add lambda functions to the C++ standard // C++ standards committee document N1958=06-028.

Schmidt D. C., and I. Pyarali (1998). Strategies for Implementing POSIX Condition Variables on Win32 // Department of Computer Science, Washington University, St. Louis, MO.

Schmidt D. C., M. Stal, et al. (2000). Patterns for Concurrent and Networked Objects // Pattern-Oriented Architecture, 2.

Shah S., G. Haab, et al. (1999). Flexible Control Structures for Parallelism in OpenMP // Proceedings of the First European Workshop on OpenMP.

Siek J., D. Gregor, et al. (2005). Concepts for C++0x.

Stepanov A. A., and M. Lee (1995). The Standard Template Library // HP Laboratories Technical Report 95-11(R.1).

Stepanov A. A. (1999). Greatest Common Measure: The Last 2500 Years.

Stroustrup B. (1994). The Design and Evolution of C++. Also known as D&E. Reading, MA: Addison Wesley.

Stroustrup B. (2000). The C++ Programming Language. Special Edition. Reading, MA: Addison Wesley.

Stroustrup B., and G. Dos Reis (2005, April). A Concept Design (rev.1) // Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21.

Stroustrup B., and G. Dos Reis (2005, October). Specifying C++ concepts // Technical Report N1886=05-0146, ISO/IEC SC22/JTC1/WG21.

Su E., X. Tian, et al. (2002, September). Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures // Fourth European Workshop on OpenMP, Rome.

Sutter H. (2005, January). The Concurrency Revolution // Dr. Dobb's Journal.

Sutter H. (2005, March). The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software // Dr. Dobb's Journal.

Voss M. (2006, December). Enable Safe, Scalable Parallelism with Intel Threading Building Blocks' Concurrent Containers.

Voss M. (2006, October). Demystify Scalable Parallelism with Intel Threading Building Blocks' Generic Parallel Algorithms.

Willcock J., J. Järvi, et al. (2006). Lambda Expressions and Closures for C++. N1968-06-0038.

Приложение B

TBB в кратком изложении

До сих пор мы уделяли основное внимание обучению, но это приложение дополняет картину формальной документацией – тем, что невозможно включить в процесс преподавания. На протяжении всей книги мы опускали некоторые детали интерфейсов, чтобы текст был удобочитаемым и не слишком громоздким. В частности, такие продвинутые понятия, как контексты групп задач TGC, были введены только во второй части книги, поэтому соответствующие параметры в первой части не упоминаются.

Вот мы и решили поместить полные определения сюда. Это приложение представляет собой собрание кратких описаний для всех категорий интерфейсов TBB.

Мы приводим лаконичные, но полные определения интерфейсов, предназначенных для использования в приложениях. Там, где это полезно, включены также иллюстративные примеры типа «Hello, World». Мы надеемся, что это поможет тем читателям, которым, подобно авторам, нужно видеть простой пример, чтобы лучше понять API. В примерах кода демонстрируется правильное использование API и образец результата, мы не пытаемся поразить ваше воображение крутыми примерами параллельного программирования. Имейте в виду, что все примеры кода в книге, включая и это приложение, можно скачать с сайта `threadingbuildingblocks.org`, так что мы рекомендуем дополнить приведенные здесь простенькие примеры и исследовать API самостоятельно. Предметный указатель и оглавление книги помогут вам найти подробные обсуждения всех тем, иллюстрируемых примерами.

Часто в описаниях встречается фраза «потенциально параллельный» – не потому, что мы сомневаемся, обеспечивает ли TBB истинно параллельное выполнение и масштабируемость, а потому, что параллелизм никогда не гарантируется. Например, если программа работает на одноядерной машине, то параллельного выполнения добиться никак не удастся. Также и в сложном конвейере отдельные этапы (фильтры) могут выполняться параллельно, но это зависит от общей рабочей нагрузки. Этот момент – *потенциальная возможность* параллелизма – тонкий, но важный, он относится ко всем компонуемым параллельным решениям.

Отладка и условный код

Для помощи при отладке в TBB имеются макросы, переменная среды (`TBB_VERSION`) и функция, которая возвращает номер версии и информацию, полезную для отладки и написания условного кода.

Макросы

#include <tbb/tbb_stddef.h>

Определяет макросы, относящиеся к версии библиотеки. Особенно полезны при написании условного кода. Не следует пытаться переопределить эти макросы.

`TBB_INTERFACE_VERSION`

Текущая версия интерфейса. Значением является десятичное число вида XYYY, где X – основной, а YYY – дополнительный номер версии.

`TBB_INTERFACE_VERSION_MAJOR`

TBB_INTERFACE_VERSION/1000, т. е. основной номер версии.

`TBB_COMPATIBLE_INTERFACE_VERSION`

Основной номер самой старой еще поддерживаемой версии интерфейса.

Рис. B.1 ❖ Макросы

Переменная среды

Если присвоить переменной среды **`TBB_VERSION`** значение 1, то библиотека будет печатать информацию о версии на stderr.
Каждая напечатанная строка имеет вид «TBB: **TAG** значение», где **TAG** и значение описаны ниже. Отладочная печать может зависеть от реализации и может быть изменена без предварительного уведомления.

`VERSION`

Номер версии библиотеки Intel TBB.

`INTERFACE_VERSION`

Значение макроса TBB_INTERFACE_VERSION в момент компиляции библиотеки.

`BUILD_...`

Различная информация о конфигурации машины, на которой собиралась библиотека.

`USE_ASSERT`

Значение макроса TBB_USE_ASSERT.

`DO_ITT_NOTIFY`

1, если библиотека может активировать инструментальное оснащение для аналитических средств в Intel Parallel Studio XE; 0 или не определено в противном случае.

`ITT`

`yes`, если библиотека активировала инструментальное оснащение для аналитических средств в Intel Parallel Studio XE; `no` в противном случае. Обычно `yes`, если программа работает под управлением аналитических средств Intel Parallel Studio XE.

`ALLOCATOR`

Распределитель памяти, лежащий в основе `tbb::tbb_allocator`: `scalable_malloc`, если была успешно загружена библиотека Intel TBB malloc, иначе `malloc`

Рис. B.2 ❖ Переменная среды

Функция: `TBB_runtime_interface_version`

#include <tbb/tbb_stddef.h>

Возвращает версию интерфейса библиотеки Intel TBB, загруженной во время выполнения. Очень удобно при отладке и написании условного кода.

`extern "C" int TBB_runtime_interface_version()`

Значение, возвращенное этой функцией, может отличаться от значения макроса `TBB_INTERFACE_VERSION`, определенного на этапе компиляции. Его можно использовать, чтобы установить, было ли приложение откомпилировано с совместимой версией заголовков Intel TBB. В общем случае значение, возвращенное во время выполнения, должно быть больше или равно значению `TBB_INTERFACE_VERSION`, иначе возможна ошибка при разрешении символов на этапе выполнения.

Рис. B.3 ❖ Функция **`TBB_runtime_interface_version`**

Отладочные макросы

Четыре макроса управляют средствами отладки. В общем случае полезно включать их на этапе разработки и выключать в производственном коде, поскольку они могут снизить производительность. В таблице ниже перечислены эти макросы и их значения по умолчанию. Значение 1 включает соответствующее средство, значение 0 выключает.

Макрос	Средство	Значение по умолчанию
`TBB_USE_DEBUG`	Значение по умолчанию для всех остальных макросов в этой таблице	По умолчанию 0, *кроме* Windows, где оно равно 1, если определен макрос `_DEBUG`
`TBB_USE_ASSERT`	Включает проверку внутренних утверждений. Может значительно снижать производительность	`TBB_USE_DEBUG`
`TBB_USE_THREADING_TOOLS`	Включает полную поддержку аналитических средств в Intel Parallel Studio XE	
`TBB_USE_PERFORMANCE_WARNINGS`	Включает предупреждения о проблемах с производительностью	

Рис. B.4 ❖ Отладочные макросы

Макросы ознакомительных средств

TBB может также включать макросы для активации «ознакомительных средств». Время от времени в очередной выпуск TBB включаются экспериментальные (или «ознакомительные») средства, которые по умолчанию обычно не активированы. Ознакомительные средства включаются в полные версии, чтобы услышать мнения пользователей, но без обязательств сохранить API в будущем. В этой книге ознакомительные средства не рассматриваются – отсылаем читателя к замечаниям к версиям, где описаны такие средства в конкретной версии TBB.

Диапазоны

Диапазон можно рекурсивно разбивать на две части. Разбиение осуществляется одним из *расщепляющих конструкторов* диапазона. Существует два типа расщепляющих конструкторов:

1) основной расщепляющий конструктор. Его рекомендуется использовать, когда диапазон разбивается на две примерно равные части, хотя это и необязательно. Расщепление на приблизительно равные части обычно дает оптимальную степень параллелизма;
2) пропорционально расщепляющий конструктор. Этот конструктор необязателен и может быть опущен, если не установлена переменная-член `is_splittable_in_proportion`. Если такой конструктор используется, то для получения наилучших результатов нужно придерживаться заданной пропорции с округлением до ближайшего целого.

В идеале диапазон допускает рекурсивное расщепление до тех пор, пока не окажется, что части, представляющие порции работы, быстрее обработать последовательно, чем производить дальнейшее разбиение. Объем работы, представленный диапазоном, обычно зависит от контекста верхнего уровня, поэто-

му тип, моделирующий диапазон, как правило, должен предоставлять способ, позволяющий управлять степенью расщепления. Например, в шаблонном классе `blocked_range` имеется параметр `grainsize`, который определяет наибольший неделимый диапазон.

Если для множества значений имеет смысл понятие направления, то, по соглашению, расщепляющий конструктор должен конструировать вторую часть диапазона и модифицировать свой аргумент, так чтобы он представлял первую часть. Это позволяет алгоритмам `parallel_for`, `parallel_reduce` и `parallel_scan` при последовательной работе обрабатывать диапазон в порядке возрастания, что типично для обыкновенного последовательного цикла.

Поскольку в классе диапазона объявлены *расщепляющий* и *копирующий* конструкторы, конструктор по умолчанию автоматически не генерируется. Поэтому необходимо либо объявить его явно, либо всегда задавать аргументы при создании экземпляра диапазона.

Требования концепции `Range`
В классе `R`, реализующем концепцию диапазона, должны быть определены следующие члены.
`R::R( const R& );`
Копирующий конструктор.
`R::~R();`
Деструктор.
`bool R::is_divisible() const;`
`True`, если диапазон можно разбить на два поддиапазона.
`bool R::empty() const;`
`True`, если диапазон пуст.
`R::R( R& r, tbb::split);`
Расщепляющий конструктор – расщепляет диапазон `r` на два поддиапазона.
`R::R( R& r, tbb::proportional_split proportion );`
Факультативно. *Расщепляющий конструктор* – пропорционально расщепляющий конструктор. Расщепляет диапазон `r` на два поддиапазона в указанной пропорции.
`static const bool R::is_splittable_in_proportion;`
Факультативно. Если `true`, то для диапазона определен пропорционально *расщепляющий конструктор*, который можно использовать в параллельных алгоритмах.

Рис. В.5 ❖ Требования концепции `range`

Разбиватели

Разбиватель определяет, как шаблон цикла должен распределять свою работу между потоками.

По умолчанию шаблоны `parallel_for`, `parallel_reduce` и `parallel_scan` стараются рекурсивно разбить диапазон на столько частей, чтобы занять все процессоры; это необязательно должны быть максимально мелкие части. Необязательный параметр `partitioner` позволяет определить другое поведение, как показано в таблице на рис. В.6. В отличие от других разбивателей, объект `affinity_partitioner` передается по *неконстантной* ссылке, потому что он модифицируется, чтобы запомнить, где именно исполняются итерации цикла. Шаблон `parallel_deterministic_reduce` поддерживает только разбиватели `simple_partitioner` и `static_partitioner`, потому что остальные по природе своей недетерминированы.

Разбиватели

#include <tbb/partitioner.h>

`const auto_partitioner&`
предопределен как **tbb::auto_partitioner()**

Подразумевается по умолчанию во всех случаях, кроме `parallel_deterministic_reduce`.
Производит разбиение, достаточное для балансировки нагрузки, но необязательно настолько мелкое, насколько допускает функция `Range::is_divisible`. При использовании с такими классами, как `blocked_range`, точный выбор степени детализации не столь важен, и зачастую приемлемой производительности можно добиться, оставив значение по умолчанию 1.

`affinity_partitioner&`
определен программой **tbb::affinity_partitioner ap**,
а затем передается как **ap** (по ссылке)

Аналогичен `auto_partitioner`, но улучшает привязку к кешу за счет отображения поддиапазонов на рабочие потоки. Может значительно повысить производительность, если цикл неоднократно выполняется с одними и теми же данными и эти данные умещаются в кеше.

`const static_partitioner&`
предопределен как **tbb::static_partitioner()**

Распределяет итерации диапазона между рабочими потоками максимально равномерно, не оставляя возможности для последующей балансировки нагрузки. Как и `affinity_partitioner`, отображает поддиапазоны на рабочие потоки. Распределение работ и отображение детерминированы и зависят только от количества итераций в диапазоне, степени детализации и количества потоков.

`const simple_partitioner&`
предопределен как **tbb::simple_partitioner()**

Подразумевается по умолчанию для `parallel_deterministic_reduce`.
Рекурсивно разбивает диапазон до тех пор, пока дальнейшее разбиение не станет невозможным. За остановку рекурсивного разбиения целиком и полностью отвечает функция `Range::is_divisible`, поэтому при использовании с такими классами, как `blocked_range`, правильный выбор степени детализации критически важен для обеспечения конкурентности и одновременно ограничения накладных расходов.

Рис. B.6 ❖ Разбиватели

Алгоритмы

TBB-алгоритмы введены в главе 2 и более подробно рассматриваются в главе 16.

Алгоритм `parallel_while` в этой книге не документируется, поскольку вместо него рекомендуется новый алгоритм `parallel_do`.

Алгоритм: parallel_do

Применяет тело в виде объекта-функции к последовательности `[first,last)`. Элементы могут обрабатываться параллельно. Можно добавлять дополнительные элементы, если в объявлении `Body::operator` задан второй аргумент типа `parallel_do_feeder`. Функция завершается, когда `body(x)` вернет управление для всех элементов `x`, присутствовавших во входной последовательности или добавленных методом `parallel_do_feeder::add`. Форма `parallel_do(c,body)` эквивалентна `parallel_do(std::begin(c),std::end(c),body)`.

parallel_do

#include <**tbb**/parallel_do.h>

```
template<typename InputIterator,
         typename Body > void
tbb::parallel_do (
     InputIterator first,
     InputIterator last,
     Body          body
 [    , tbb::task_group_context &context ]
)
template<typename Container,
         typename Body > void
tbb::parallel_do (
     [const] Container  &c,
     Body  body
 [    , tbb::task_group_context &context ]
```

Первый вариант `parallel_do(first, last, body)` применяет объект-функцию `body` к последовательности `[first, last)`. Элементы могут обрабатываться параллельно. Дополнительные элементы могут добавляться телом `body`, если в его объявлении задан второй аргумент типа `parallel_do_feeder`. Функция завершается, когда `body(x)` вернет управление для всех элементов `x`, присутствовавших во входной последовательности или добавленных методом `parallel_do_feeder::add`.

Задание податчика `feeder` с помощью лямбда-выражения не поддерживается (по крайней мере, в версии TBB 2019).

Для этого шаблона разбиватель не задается.

Рис. B.7 ❖ Алгоритм `parallel_do`

parallel_do – нет примера типа Hello, World

```
/* Возможность добавления новой работы во время выполнения НЕ показана */
#include <cstdio>
#include <tbb/tbb.h>
#include <array>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter;

#define PN(Y) printf( "\nHello, Do (or Do Not) " #Y ":\n\t");
#define PV(X) {                                          \
           int mycount;                                  \
           { tbb::spin_mutex::scoped_lock hello(mylock); \
             mycount = counter++;                        \
           }                                             \
           printf(" %02d.%d", X, mycount);               \
         }

int main( int argc, char *argv[] ) {
   std::array<int,10> myarray  = { 19,13,14,11,15,20,17,16,12,18 };
   std::array<int,10> disarray = { 23,29,27,25,30,21,26,24,28,22 };

   counter = 0;
   PN(myarray);
   tbb::parallel_do( myarray.begin(),myarray.end(),
                     [](int xyzzy){ PV(xyzzy); }
                    );
   printf("\n");
```

Рис. B.8 ❖ Алгоритм `parallel_do`: пример (*начало*)

```
  counter = 0;
  PN(disarray);
  tbb::parallel_do( disarray,
                    [](int xyzzy){ PV(xyzzy); }
                   );
  printf("\n\n");
 }
```

```
Обратите внимание, что порядок посещения элементов отличается от порядка их следования в массиве.
Это лишь пример вывода – порядок может отличаться от прогона к прогону и от машины к машине!
```

```
Hello, Do (or Do Not) myarray:
        13.0 17.1 20.2 18.3 16.4 12.5 19.6 15.7 11.8 14.9

Hello, Do (or Do Not) disarray:
        29.0 23.1 30.2 27.3 25.5 26.4 21.6 22.7 28.8 24.9
```

Попробуйте ради забавы (в предположении, что результатом компиляции является файл ./a.out).

```
    for i in `seq 1 1000`;
      do (./a.out; ./a.out; ./a.out; /a.out & );
    done | grep 29 | sort | uniq
```

```
      26.0 30.1 21.2 29.3 27.4 23.5 22.6 28.7 25.8 24.9
      29.0 23.1 26.2 30.3 21.4 25.5 27.6 22.7 28.8 24.9
      29.0 23.1 30.2 27.3 25.4 21.6 26.5 22.7 28.8 24.9
      29.0 23.1 30.2 27.3 25.4 22.6 24.7 28.8 21.9 26.5
      29.0 23.1 30.2 27.3 25.4 22.6 26.5 21.7 24.8 28.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 22.7 24.8 28.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 22.7 24.9 28.8
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 22.7 28.8 24.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 28.7 24.8 22.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 28.8 22.7 24.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 21.7 24.8 28.9
      29.0 23.1 30.2 27.3 25.4 26.5 21.6 21.7 28.8 24.9
```

Рис. B.8 (*окончание*)

Требования к телу алгоритма `parallel_do` (и типу T)

Класс `Body`, реализующий концепцию `parallel_do`, должен определять:

```
Body::operator()(
  [ cv-квалификаторы ] T [ ссылка ] iitem.
  [  , parallel_do_feeder<T>& feeder ]
) const
```

Обработать элементы работы. `parallel_do` может конкурентно вызывать функцию `operator()` одного и того же объекта тела для разных элементов. Оператор может принимать элементы по ссылке или по значению, в т. ч. по ссылке `rvalue`. Если в операторе присутствует аргумент `feeder`, то разрешено добавлять новые элементы работы. ПРЕДОСТЕРЕЖЕНИЕ: одновременно определять варианты `operator()` с одним и с двумя аргументами *не* разрешается.

```
~T::T()
```

Уничтожает элемент работы.

```
T( T&& )
```

Факультативно. Перемещает элемент работы.

```
T( const T& )
```

Факультативно. Копирует элемент работы.

Рис. B.9 ❖ Алгоритм `parallel_do`: требования к телу

Алгоритм: parallel_for

Вариант с параметром range – самая общая и эффективная форма параллельного итерирования. Ему соответствует потенциально параллельное выполнение тела для каждого значения в диапазоне. Тип Range должен моделировать концепцию диапазона (требования приведены после объяснения и примера кода). Вариант с параметрами first/last/step описывает потенциально параллельное выполнение цикла: for (auto i=first; i<last; i+=step) f(). Если параметр step не задан, предполагается, что он равен 1. Факультативный параметр partitioner задает стратегию разбиения, как описано в главе 12. Параметр task_group_context задает контекст группы задач для отмены и обработки исключений.

parallel_for

#include <**tbb**/parallel_for.h>

```
template<typename Range, typename Body > void
tbb::parallel_for (const Range &range,
                   const Range &body,
           [, tbb::simple_partitioner() |
      , tbb::auto_partitioner() |
      , tbb::static_partitioner() |
      , tbb::affinity_partitioner &ap ]
           [, tbb::task_group_context &context ]
)
```

```
template<typename Range, typename Func > void
tbb::parallel_for (Iterator first,
                   Iterator last,
                 [ Iterator step, ]
                   const Func &func
           [, tbb::simple_partitioner() |
      , tbb::auto_partitioner() |
      , tbb::static_partitioner() |
      , tbb::affinity_partitioner &ap ]
           [, tbb::task_group_context &context ]
)
```

Параллельное итерирование элементов в диапазоне [first, last) или range.
Функция func применяется к одному значению итератора Iterator, если используется вариант с параметрами [first, last).
Функция body применяется к диапазону, если используется вариант с параметром range.
Если разбиватель не задан, то по умолчанию используется auto_partitioner.
Задавать параметр context необязательно.

Рис. В.10 ❖ Алгоритм parallel_for

parallel_for – печать Hello, World 10 раз

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;

int main( int argc, char *argv[] ) {
  tbb::parallel_for(
    tbb::blocked_range<int>( 0, 10 ),
    [](const tbb::blocked_range<int>& r) {
      for( int i=r.begin(); i!=r.end(); ++i ) {
        int mycount;
        { tbb::spin_mutex::scoped_lock hello(mylock);
          mycount = counter++;
        }
        printf("Hello, World (%02d):(%02d)\n",i,mycount);
      }
    } /* конец лямбда-выражения */
  ); /* конец parallel_for */
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, World (00):(00)
Hello, World (05):(01)
Hello, World (01):(02)
Hello, World (06):(03)
Hello, World (07):(04)
Hello, World (08):(05)
Hello, World (02):(06)
Hello, World (09):(07)
Hello, World (03):(08)
Hello, World (04):(09)
```

Рис. В.11 ❖ Алгоритм parallel_for: пример с диапазоном типа blocked_range

parallel_for *(с явно заданными началом и концом)* **– печать Hello, World 10 раз**

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;

int main( int argc, char *argv[] ) {
  tbb::parallel_for(
    0,
    10,
    [](int i) {
      int mycount;
      { tbb::spin_mutex::scoped_lock hello(mylock);
        mycount = counter++;
      }
      printf("Hello, World (%02d):(%02d)\n",i,mycount);
    } /* конец лямбда-выражения */
  ); /* конец parallel_for */
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, World (00):(00)
Hello, World (05):(01)
Hello, World (01):(02)
Hello, World (07):(03)
Hello, World (08):(04)
Hello, World (06):(05)
Hello, World (02):(06)
Hello, World (03):(07)
Hello, World (09):(08)
Hello, World (04):(09)
```

Рис. В.12 ❖ Алгоритм `parallel_for`: пример с параметрами `first`, `last`

Требования к телу алгоритма `parallel_for`

Класс `Body`, реализующий концепцию `parallel_for`, должен определять:

`Body::Body( const Body& )(`

Копирующий конструктор.

`Body::~Body()(`

Деструктор.

`void Body::operator()( Range& r ) const;`

Оператор вызова, применяющий тело к диапазону `r`.

Рис. В.13 ❖ Алгоритм `parallel_for`: требования к телу

Алгоритм: parallel_for_each

Применяет объект-функцию `f` к каждому элементу последовательности [`first`, `last`) или контейнера `c`, потенциально параллельно.

parallel_for_each

#include <tbb/parallel_for_each.h>

```
template<typename Container, typename Func > void
tbb::parallel_for_each (      Container &c,
                        const Func      &func
            [, tbb::task_group_context &context ]
)
```

```
template<typename Iterator, typename Func > void
tbb::parallel_for_each (Iterator first,
                        Iterator last,
                        const Func &func
            [, tbb::task_group_context &context ]
)
```

Параллельное итерирование элементов последовательности `[first, last)` или контейнера `c`.
Функция `func` применяется к одному элементу.
В этом шаблоне разбиватель не задается.
Задавать параметр `context` необязательно.

Рис. B.14 ❖ Алгоритм `parallel_for_each`

parallel_for_each – печать Hello, World 10 раз

```
#include <cstdio>
#include <array>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;
std::array<int, 10> values{ { 11,22,33,44,55,66,77,88,99,42 } };

int main( int argc, char *argv[] ) {
  tbb::parallel_for_each(
    std::begin(values),
    std::end(values),
    [](int i) {
      int mycount;
      { tbb::spin_mutex::scoped_lock hello(mylock);
        mycount = counter++;
      }
      printf("Hello, World (%02d):(%02d)\n",i,mycount);
    } /* конец лямбда-выражения */
  ); /* конец parallel_for_each */
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, World (11):(00)
Hello, World (66):(01)
Hello, World (33):(02)
Hello, World (22):(03)
Hello, World (88):(04)
Hello, World (77):(05)
Hello, World (44):(06)
Hello, World (99):(07)
Hello, World (55):(08)
Hello, World (42):(09)
```

Рис. B.15 ❖ Алгоритм `parallel_for_each`: пример

Алгоритм: parallel_invoke

Вычисляет $f_1()$, $f_2()$, ..., $f_n()$ потенциально параллельно. Аргументами могут быть объекты-функции, лямбда-выражения или указатели на функции. Поддерживается 2 и более аргументов. В первоначальной версии TBB количество параметров не должно было превышать 10, но благодаря появлению в C++11 шаблонов с переменным числом параметров это ограничение теперь снято.

parallel_invoke

#include <**tbb**/parallel_invoke.h>

```
template<typename FUNC1,
         typename FUNC2
      [, typename FUNCN]* > void
tbb::parallel_invoke ( const FUNC1 &f01,
                       const FUNC2 &f02
                   [,  const FUNCN &fn ]* )
           [, tbb::task_group_context &context ]
)
```

Выполняет список из двух или более задач параллельно и ждет завершения всех задач. Задавать параметр `context` необязательно.

Рис. B.16 ❖ Алгоритм `parallel_invoke`

parallel_invoke – печать Hello, World 10 раз

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;

void hw() {
  int mycount;
  { tbb::spin_mutex::scoped_lock hello(mylock);
    mycount = counter++;
  }
  printf("Hello, World (%02d):(%02d)\n",42,mycount);
}

int main( int argc, char *argv[] ) {
  tbb::parallel_invoke( hw,hw,hw,hw,hw, hw,hw,hw,hw,hw );
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, World (42):(00)
Hello, World (42):(01)
Hello, World (42):(02)
Hello, World (42):(03)
Hello, World (42):(04)
Hello, World (42):(05)
Hello, World (42):(06)
Hello, World (42):(07)
Hello, World (42):(08)
Hello, World (42):(09)
```

Рис. B.17 ❖ Алгоритм `parallel_invoke`: пример

Алгоритм: parallel_pipeline

Функция parallel_pipeline предоставляет строго типизированный, принимающий лямбда-выражения интерфейс для построения и выполнения конвейеров, потенциально параллельно. Благодаря строгой типизации и поддержке лямбда-выражений она более предпочтительна, чем класс pipeline. Потоковый граф предлагает гораздо более общее решение, которое следует использовать, только если эта общность необходима.

parallel_pipeline

#include <tbb/parallel_pipeline.h>

```
void tbb::parallel_pipeline (
       size_t max_number_of_live_tokens,
       const filter_t<void,void>& filter_chain
    [, tbb::task_group_context &context ]
)
```

Строго типизированный интерфейс для выполнения конвейеров с поддержкой лямбда-выражений (в отличие от класса tbb::pipeline). Для более сложных потоков следует явно использовать потоковый граф (см. главу 3). Задавать параметр context необязательно.

Рис. B.18 ❖ Алгоритм parallel_pipeline

Класс tbb::flow_control

#include <**tbb**/pipeline.h>

```
Void Stop()
```

Шаблонная функция parallel_pipeline передает объект fc типа flow_control входному функтору типа filter_t. Дойдя до конца входных данных, этот функтор должен вызвать fc.stop() и вернуть фиктивное значение.

Рис. B.19 ❖ Алгоритм parallel_pipeline, класс flow_control

Шаблонный класс tbb::filter_t

#include <**tbb**/pipeline.h>

```
filter_t()
```

Конструирует неопределенный фильтр.
ПРЕДОСТЕРЕЖЕНИЕ. Результат использования неопределенного фильтра в operator& или parallel_pipeline не определен.

```
filter_t( const filter_t<T,U>& rhs )
```

Конструирует копию rhs.

```
template<typename Func>
filter_t( filter::mode mode, const Func& f )
```

Конструирует фильтр filter_t, который использует копию функтора f для отображения входного значения t типа T в выходное значение u типа U.
Существует три режима работы фильтра:

- в режиме parallel фильтр может обрабатывать несколько элементов параллельно, порядок обработки не определен;
- в режиме serial_out_of_order фильтр обрабатывает элементы по одному, порядок обработки не определен;
- в режиме serial_in_order фильтр обрабатывает элементы по одному. Порядок обработки неявно устанавливается первым фильтром вида serial_in_order и соблюдается всеми остальными фильтрами такого вида в конвейере.

ПРИМЕЧАНИЕ. Когда в функции parallel_pipeline используется filter_t, она вычисляет u как f(t), если только тип T не совпадает с void, а в последнем случае вычисляет выражение u=f(fc), где fc имеет тип flow_control.

```
void operator=( const filter_t<T,U>& rhs )
```

Модифицирует *this, так чтобы использовался функтор rhs.

Рис. B.20 ❖ Алгоритм parallel_pipeline, класс filter_t (*начало*)

`~filter_t()`
Уничтожает объект `filter_t`.
`void clear()`
Делает фильтр `*this` неопределенным.
`template<typename T, typename U, typename Func> filter_t<T,U>` `make_filter(filter::mode mode, const Func& f)`
Возвращает `filter_t<T,U>(mode, f)`.

`template<typename T, typename V, typename U> filter_t<T,U>`
`operator& (const filter_t<T,V>& left,`
`const filter_t<V,U>& right)`

Возвращает объект `filter_t`, представляющий композицию левой и правой частей. Композиция ведет себя так, как если бы значение `left` было подано на вход `right`. Требование: выходной тип фильтра `left` должен совпадать с входным типом фильтра `right`.

Рис. В.20 (*окончание*)

parallel_pipeline – печать Hello, World 10 раз

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
int counter = 0;

void hw(int x, int v) {
  int mycount;
  { tbb::spin_mutex::scoped_lock hello(mylock);
    mycount = counter++;
  }
  printf("Hello, Stage %d (%02d):(%02d)\n",x,mycount,v);
}

int main( int argc, char *argv[] ) {
  int cdown = 3;
  tbb::parallel_pipeline(
    6,
    tbb::make_filter<void,int>(
      tbb::filter::parallel,[&](tbb::flow_control& fc)->int
        { hw(1,0);
        if (!cdown) { fc.stop(); return 0; }
        return 1000 * cdown--; }) &
    tbb::make_filter<int,float>(
      tbb::filter::parallel,[](int i)
        { hw(2,i); return ++i; }) &
    tbb::make_filter<float,float>(
      tbb::filter::parallel,[](float f)
        { hw(3,(int)f); return f+1.0f; }) &
    tbb::make_filter<float,int>(
      tbb::filter::parallel,[](float f)
        { hw(4,(int)f); return (int)f+1;}) &
    tbb::make_filter<int,void>(
      tbb::filter::parallel,[](int i)
        { hw(5,i); })
  );
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!
Признание. Чтобы получить показанный ниже результат, мы умеренно загрузили машину другой работой. Иначе первый фильтр отработал бы до конца, прежде чем второй приступил к работе, поскольку пример слишком прост, – и тогда результат получился бы скучным. Необходимо помнить, что в коде (и в тесте) возможны различные порядки параллельного выполнения.

Рис. В.21 ❖ Алгоритм `parallel_pipeline`: пример (*начало*)

```
Hello, Stage 1 (00):(00)
Hello, Stage 2 (01):(3000)
Hello, Stage 3 (02):(3001)
Hello, Stage 4 (03):(3002)
Hello, Stage 5 (04):(3003)
Hello, Stage 1 (05):(00)
Hello, Stage 2 (06):(2000)
Hello, Stage 3 (07):(2001)
Hello, Stage 4 (08):(2002)
Hello, Stage 5 (09):(2003)
Hello, Stage 1 (10):(00)
Hello, Stage 2 (11):(1000)
Hello, Stage 3 (12):(1001)
Hello, Stage 4 (13):(1002)
Hello, Stage 5 (14):(1003)
Hello, Stage 1 (15):(00)
```

Отсортированный вывод (точно тот же прогон, но с сортировкой):

```
Hello, Stage 1 (00):(00)
Hello, Stage 1 (05):(00)
Hello, Stage 1 (10):(00)
Hello, Stage 1 (15):(00)
Hello, Stage 2 (01):(3000)
Hello, Stage 2 (06):(2000)
Hello, Stage 2 (11):(1000)
Hello, Stage 3 (02):(3001)
Hello, Stage 3 (07):(2001)
Hello, Stage 3 (12):(1001)
Hello, Stage 4 (03):(3002)
Hello, Stage 4 (08):(2002)
Hello, Stage 4 (13):(1002)
Hello, Stage 5 (04):(3003)
Hello, Stage 5 (09):(2003)
Hello, Stage 5 (14):(1003)
```

Рис. B.21 (*окончание*)

Алгоритм: parallel_reduce и parallel_deterministic_reduce

Редукция (потенциально параллельная) поддерживается как в детерминированном (несколько медленнее, но результаты предсказуемые и воспроизводимые, что полезно, по крайней мере, во время отладки!), так и в недетерминированном (чуть более быстром) варианте. Для каждого варианта редукции имеется две формы. *Функциональная* форма предназначена для использования совместно с лямбда-выражениями, а *императивная* сводит к минимуму копирование данных. Функциональная форма `parallel_reduce(range, identity, func, reduction)` параллельно применяет функтор `func` к поддиапазонам `range` и редуцирует результат с помощью бинарного оператора `joinfunc`. При этом возвращается результат редукции. Параметры `func` и `joinfunc` могут быть лямбда-выражениями. Императивная форма `parallel_reduce(range,body)` выполняет параллельную редукцию, применяя тело к каждому значению в диапазоне `range`. Тип Range должен моделировать концепцию диапазона (требования были сформулированы выше в этом приложении). Тело должно моделировать требования, показанные на рис. B.22.

parallel_reduce и parallel_deterministic_reduce

#include <**tbb**/parallel_reduce.h>

```
template<typename Range, typename Value,
                         typename Func,
                         typename Reduction >
Value tbb::parallel_[deterministic_]reduce(
                    const Range     &range,
                    const Value     &identity,
                    const Func      &func,
                    const Reduction &joinfunc
           [, tbb::simple_partitioner() |
      , tbb::auto_partitioner() |
      , tbb::static_partitioner() |
      , tbb::affinity_partitioner &ap ]
           [, tbb::task_group_context &context ]
)
```

```
template<typename Range, typename Body>
void tbb::parallel_[deterministic_]reduce(
                    const Range     &range,
                    const Body      &body
           [, tbb::simple_partitioner() |
      , tbb::auto_partitioner() |
      , tbb::static_partitioner() |
      , tbb::affinity_partitioner &ap ]
           [, tbb::task_group_context &context ]
)
```

Параллельное итерирование с редукцией (детерминированной или нет).
Первая форма принимает лямбда-выражения, потому что функции операции и редукции (соединения) выделены в отдельные параметры, тогда как во второй форме требуется тело, в котором обе функции определены как члены.
Если разбиватель не задан, то по умолчанию используется `auto_partitioner`.
В случае детерминированной редукции разрешается использовать только простой или статический разбиватель (поскольку остальные недетерминированы).
Задавать параметр `context` необязательно.

Рис. B.22 ❖ Алгоритм `parallel_[deterministic_]reduce`

parallel_reduce – печать Hello, World 10 раз

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;
int values[] = { 11,22,33,44,55,66,77,88,99,42 };

int main( int argc, char *argv[] ) {
  int howmany =
    parallel_reduce(
      tbb::blocked_range<int*>( values, values+10 ),
      0,
      [](const tbb::blocked_range<int*>& r, int init)->int {
        int rr = init;
        for( int* a=r.begin(); a!=r.end(); ++a ) {
          int mycount;
          { tbb::spin_mutex::scoped_lock hello(mylock);
            mycount = counter++;
          }
          printf("Hello, World (%02d):(%02d)\n",*a,mycount);
          rr += *a;
        } /* конец лямбда-выражения */
        return rr;
      },
      []( int x, int y )->int {
        return x+y;
      }
    );
  printf("Hello, parallel_reduce(%02d)\n",howmany);
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, World (11):(00)
Hello, World (66):(01)
Hello, World (22):(02)
Hello, World (33):(03)
Hello, World (77):(04)
Hello, World (88):(05)
Hello, World (44):(06)
Hello, World (55):(07)
Hello, World (99):(08)
Hello, World (42):(09)
Hello, parallel_reduce(537)
```

Рис. В.23 ❖ Алгоритм `parallel_reduce`: пример

parallel_deterministic_reduce – печать Hello, World 10 раз

```
/*
   В ТОЧНОСТИ то же самое, что parallel_reduce…
   только parallel_reduce заменено на parallel_deterministic_reduce
 */
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;
int values[] = { 11,22,33,44,55,66,77,88,99,42 };

int main( int argc, char *argv[] ) {
  int howmany =
    parallel_deterministic_reduce(
      tbb::blocked_range<int*>( values, values+10 ),
      0,
      [](const tbb::blocked_range<int*>& r, int init)->int {
        int rr = init;
        for( int* a=r.begin(); a!=r.end(); ++a ) {
          int mycount;
          { tbb::spin_mutex::scoped_lock hello(mylock);
            mycount = counter++;
          }
          printf("Hello, World (%02d):(%02d)\n",*a,mycount);
          rr += *a;
        } /* конец лямбда-выражения */
        return rr;
      },
      []( int x, int y )->int {
        return x+y;
      }
    );
  printf("Hello, parallel_reduce(%02d)\n",howmany);
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине! «Детерминированность» проявляется в порядке выполнения редукции. Отдельные вычисления по-прежнему могут выполняться в любом порядке, потому что это не влияет на итоговое значение!

```
Hello, World (11):(00)
Hello, World (22):(01)
Hello, World (66):(02)
Hello, World (33):(03)
Hello, World (88):(04)
Hello, World (77):(05)
Hello, World (44):(06)
Hello, World (99):(07)
Hello, World (42):(08)
Hello, World (55):(09)
Hello, parallel_deterministic_reduce(537)
```

Рис. В.24 ❖ Алгоритм `parallel_deterministic_reduce`: пример

Требования к телу алгоритма `parallel_[deterministic_]reduce`
Класс `Body`, реализующий концепцию `parallel_[deterministic_]reduce`, должен определять:
`Body::Body( Body&, split )(`
Расщепляющий конструктор. Должен уметь работать конкурентно с `operator()` и методом `join`.
`Body::~Body()(`
Деструктор.
`void Body::operator()( const Range& r );`
Оператор вызова, применяющий тело к диапазону `r` и накапливающий результат.
`void Body::join()( Body& b );`
Соединение результатов. Результат в `b` должен быть объединен с результатом в `this`.

Рис. B.25 ❖ Алгоритм `parallel_[deterministic_]reduce`: требования к телу

Алгоритм: parallel_scan

Шаблонная функция `parallel_scan` вычисляет (потенциально параллельно) префикс. Эта операция, называемая еще параллельным сканированием, представляет собой продвинутую концепцию параллельного вычисления и иногда бывает полезна в ситуациях, которые на первый взгляд кажутся принципиально последовательными. Математически параллельный префикс определяется так. Пусть × – ассоциативная операция с левой единицей $id_{\times}$. Параллельным префиксом × над последовательностью $z_0, z_1, \ldots z_{n-1}$ называется последовательность $y_0, y_1, y_2, \ldots y_{n-1}$, где $y_0 = id_{\times} \times z_0$ и $y_i = y_{i-1} \times z_i$. Операция параллельного сканирования производит это вычисление параллельно, изменяя порядок вычисления × и выполняя два прохода. При этом операция × может вызываться до двух раз больше, чем при последовательном вычислении префикса. Хотя работы больше, при правильно выбранной степени детализации параллельный алгоритм может оказаться быстрее последовательного, поскольку распределяет работу между несколькими аппаратными потоками. Для получения заметного ускорения рекомендуется выполнять алгоритм в системе с тремя ядрами и больше. У шаблонной функции `parallel_scan` есть две формы. Императивная форма `parallel_scan(range, body)` реализует операцию параллельного префикса обобщенно. Тип `Range` должен моделировать концепцию диапазона (требования сформулированы выше в этом приложении). Тело должно моделировать требования, описанные в таблице ниже.

parallel_scan

#include <**tbb**/parallel_scan.h>

```
template<typename Range,
         const Body &body > void
tbb::parallel_scan(const Range    &range,
                   const Body     &body
           [, tbb::simple_partitioner() |
            , tbb::auto_partitioner() |
           [, tbb::task_group_context &context ]
)
```

```
template<typename Range,
         typename Value,
         typename Scan,
         typename ReverseJoin > Value
tbb::parallel_scan(const Range    &range,
                   const Value    &identity,
                   const Scan     &scan,
                const ReverseJoin &reverse_join
           [, tbb::simple_partitioner() |
            , tbb::auto_partitioner() |
           [, tbb::task_group_context &context ]
)
```

Параллельный префикс.
Необходимо задать либо `body`, либо `identity+scan+reverse_join`.
Если разбиватель не задан, то по умолчанию используется `auto_partitioner`.
TBB (версия TBB 2019) не поддерживает `static_partitioner` и `affinity_partitioner` в алгоритме `parallel_scan`.
TBB (версия TBB 2019) не поддерживает задания пользователем контекста в алгоритме `parallel_scan`.

Рис. B.26 ❖ Алгоритм `parallel_scan`

parallel_scan – Hello, World, сканирование

```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
// Если объявить counter как атомарную переменную (глава 5), а не просто
// как int, то отпадет необходимость в охране доступа с помощью мьютекса.
int counter = 0;
int in[10] = { 1,2,3,4,5,6,7,8,9,10 };
int out[10];

int main( int argc, char *argv[] ) {
  int re = 0;
  re = tbb::parallel_scan(
         tbb::blocked_range<int>(0,10,1),
         0,
         [](const tbb::blocked_range<int>& r, int gsum,
          bool is_final_scan)->int {
           int psum = gsum;
           int mycount;
           int tst = 0;
           for( int i=r.begin(); i<r.end(); ++i ) {
             psum += in[i];
             tst++;
             if ( is_final_scan ) out[i] = psum;
             { tbb::spin_mutex::scoped_lock hello(mylock);
               mycount = counter++;
             }
```

Рис. B.27 ❖ Алгоритм `parallel_scan`: пример (*начало*)

```
            }
            printf("Hello, World (%02d) %02d+[%02d-%02d)=%02d%c\n",
                   mycount,gsum,r.begin(),r.end(),
                   psum,is_final_scan?'F':'-');
            return psum;
          }, /* конец лямбда-выражения сканирования */
          []( int left, int right )->int {
            printf("Hello, Scan %02d + %02d\n",left,right);
            return left + right;
           } /* конец лямбда-выражения обратного соединения */
        ); /* конец parallel_scan */
  printf("Hello, Scanned%4d%4d%4d%4d%4d%4d%4d%4d%4d%4d%4d\n",
         in[0],in[1],in[2],in[3],in[4],
         in[5],in[6],in[7],in[8],in[9]);
  printf(" yielding %4d%4d%4d%4d%4d%4d%4d%4d%4d%4d%4d ->%4d\n",
         out[0],out[1],out[2],out[3],out[4],
         out[5],out[6],out[7],out[8],out[9],
         re);
}
```

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!

```
Hello, Scan 00 +  00
Hello, World (00) 00+[00-01)=01F
Hello, World (01) 01+[01-02)=03F
Hello, World (02) 00+[05-06)=06-
Hello, World (03) 06+[06-07)=13-
Hello, World (04) 00+[02-03)=03-
Hello, World (06) 03+[03-05)=12-
Hello, Scan 04 +  12
Hello, World (09) 03+[02-05)=15F
Hello, World (14) 15+[05-10)=55F
Hello, Scanned   1   2   3   4   5   6   7   8   9  10
     yielding    1   3   6  10  15  21  28  36  45  55 -> 55
```

Рис. B.27 ❖ Алгоритм `parallel_scan`: пример (*окончание*)

Требования к телу алгоритма `parallel_scan`

Класс `Body`, реализующий концепцию тела `parallel_scan`, должен определять:

`Body::Body( Body&, split )(`

Расщепляющий конструктор. Расщепляет `b` так, чтобы `this` и `b` можно было аккумулировать порознь.

`Body::~Body()(`

Деструктор.

```
void Body::operator()( const Range& r
                       pre_scan_tag );
```

Выполняет окончательную обработку итераций диапазона `r`.

`void Body::reverse_join()( Body& b );`

Объединить состояние `a`, полученное в результате предобработки, с `this`, где `a` было ранее создано из `b` расщепляющим конструктором `b`.

Рис. B.28 ❖ Алгоритм `parallel_scan`: требования к телу

Алгоритм: parallel_sort

Сортирует последовательность или контейнер, потенциально параллельно. Сортировка не является *ни* устойчивой, *ни* детерминированной – относительный порядок элементов с одинаковыми ключами *не* сохраняется, и *не* гарантируется, что при повторной сортировке порядок останется точно таким же. Требования к итератору и последовательности такие же, как для алгоритма `std::sort`. Точнее, `RandomAccessIterator` должен быть итератором произвольного доступа, а тип его значения `T` должен моделировать требования, перечисленные на рис. В.29. Вызов `parallel_sort(begin,end,comp)` сортирует последовательность `[begin, end)`, применяя аргумент `comp` для сравнения элементов. Если `comp(x,y)` возвращает `true`, то `x` будет встречаться раньше `y` в отсортированной последовательности. Вызов `parallel_sort( begin, end )` эквивалентен вызову `parallel_sort(begin,end,std::less<T>)`. Вызов `parallel_sort(c[,comp])` эквивалентен вызову `parallel_sort(std::begin(c),std::end(c)[,comp])`.

parallel_sort

#include <**tbb**/parallel_sort.h>

```
template<typename Container,
         typename Compare > void
tbb::parallel_sort(
     [const] Container   &c
   [, const  Compare     &comp ]
)
```

```
template<typename RandomAccessIterator,
         typename Compare > void
tbb::parallel_sort(
     RandomAccessIterator first,
     RandomAccessIterator last,
   [, const  Compare      &comp ]
)
```

```
template<typename T,
         typename Compare > void
tbb::parallel_sort(
     T *begin,
     T *end
  [, const  Compare &comp ]
)
```

Сортирует данные в диапазоне `[first,last)` или `[*begin,*end)`, применяя факультативно задаваемый компаратор. Если компаратор не задан, по умолчанию используется `std::less`.
Для этого шаблона разбиватель не задается.
TBB (версия TBB 2019) не поддерживает задания пользователем контекста в алгоритме `parallel_sort`.

Рис. В.29 ❖ Алгоритм `parallel_sort`

parallel_sort – Hello, World, сортировка

```
#include <cstdio>
#include <tbb/tbb.h>
#include <array>

#define PV(X) printf(" %02d", X);
#define PN(Y) printf( "\nHello, Sorted " #Y ":\t");
#define P(N) PN(N); std::for_each(N.begin(),N.end(),[](int x) { PV(x); });
#define V(Z) myvect.push_back(Z);

int main( int argc, char *argv[] ) {
  int myvalues[]               = { 3, 9, 4, 5, 1, 7, 6, 8, 10, 2 };
  std::array<int, 10> myarray  = { 19, 13, 14, 11, 15, 20, 17, 16, 12, 18 };
  std::array<int, 10> disarray = { 23, 29, 27, 25, 30, 21, 26, 24, 28, 22 };
  tbb::concurrent_vector<int> myvect;
  V(40); V(31); V(37); V(33); V(34); V(32); V(34); V(35); V(38); V(36);

  tbb::parallel_sort( myvalues,myvalues+10 );
  tbb::parallel_sort( myarray.begin(), myarray.end() );
  tbb::parallel_sort( disarray );
  tbb::parallel_sort( myvect );

  PN(myvalues); for(int i=0;i<10;i++) PV(myvalues[i]);
  P(myarray);
  P(disarray);
  P(myvect);
  printf("\n\n");
}
```

Этот результат детерминированный!

```
  Hello, Sorted myvalues:  01 02 03 04 05 06 07 08 09 10
  Hello, Sorted myarray:   11 12 13 14 15 16 17 18 19 20
  Hello, Sorted disarray:  21 22 23 24 25 26 27 28 29 30
  Hello, Sorted myvect:    31 32 33 34 34 35 36 37 38 40
```

Рис. B.30 ❖ Алгоритм `parallel_sort`: пример

Требования к итераторам для `parallel_sort`

Требования к типу итератора `It` и типу его значения `T` для алгоритма `parallel_sort`:

`void iter_swap( It a, It b )`

Обменивает значения элементов, на которые указывают итераторы `a` и `b`. `It` должен быть итератором произвольного доступа.

`bool Compare::operator()( const T& x, T& y)`

`True`, если `x` предшествует `y`.

Рис. B.31 ❖ Алгоритм `parallel_sort`: требования к итератору

Алгоритм: pipeline

Конвейер представляет применение последовательности фильтров к потоку элементов, потенциально параллельно. Каждый фильтр работает в одном из режимов параллельно, последовательно по порядку или последовательно не по порядку. Конвейер содержит один или несколько фильтров. Альтернативами классу `pipeline` являются алгоритм `parallel_pipeline` (рекомендуется, потому что принимает лямбда-выражения) и потоковый граф (рекомендуется в силу гораздо большей общности, но его следует применять, только когда эта общность действительно необходима).

pipeline

#include <**tbb**/pipeline.h>

`class pipeline;`

Это класс для конвейерного выполнения. Если требуется поддержка лямбда-выражений и строже типизированный интерфейс, обратитесь к новому алгоритму `parallel_pipeline`. Еще одна альтернатива – явный потоковый граф (см. главу 3).

`pipeline()`

Конструирует конвейер без фильтров.

`~pipeline()`

Удаляет все фильтры из конвейера и уничтожает сам конвейер.

`void add_filter( filter& a )`

Добавляет фильтр `f` в последовательность фильтров конвейера. Фильтра `f` не должно быть в конвейере.

```
void run( size_t max_number_of_live_tokens
        [, tbb::task_group_contect &group ])
)
```

Выполняет конвейер, пока первый фильтр не вернет `NULL`, а после этого ждет, когда все последующие фильтры закончат обработку всех элементов от своих предшественников. Количество параллельно обрабатываемых элементов зависит от структуры конвейера и количества доступных потоков. В каждый момент времени в конвейере может находиться не более `max_number_of_live_tokens` элементов. Конвейер можно запускать несколько раз. Добавление новых этапов между запусками безопасно. Конкурентное выполнение функции `run` для одного и того же экземпляра конвейера запрещено.
Если задан аргумент `group`, то задачи конвейера выполняются в этой группе. По умолчанию алгоритм выполняется в собственной группе задач.

`void clear()`

Удаляет все фильтры из конвейера.

Рис. B.32 ❖ Алгоритм `pipeline`

pipeline – Hello, World 10 раз

```
#include <cstdio>
#include <tbb/tbb.h>

class SayIntro : public tbb:filter {
public:
  tbb::spin_mutex firstlock; int first = 0;
  SayIntro(void) : tbb::filter(parallel) {}
  ~SayIntro(void) {}
  void* operator() (void* inp) {
    int *fortytwo = (int*)malloc(sizeof(int));
    { tbb::spin_mutex::scoped_lock hello(firstlock);
      if (first++>1) return NULL;
    }
    printf("Our pipeline begins at %d.\n",*fortytwo);
    return fortytwo;
  }
};

class SayHello : public tbb:filter {
public:
  tbb::spin_mutex mylock; int counter = 0;
  SayHello(void) : tbb::filter(parallel) {}
  ~SayHello(void) {}
  void* operator() (void* inp) {
    printf("Hello, Pipeline %02d\n",*((int*)inp));
    *(int*)inp += 1;
    return inp;
  }
};

class SayBye : public tbb:filter {
public:
  SayBye(void) : tbb::filter(serial) {}
  ~SayBye(void) {}
  void* operator() (void* inp) {
    printf("Value is finally %d; Good bye!\n",*((int*)inp));
    return NULL;
  }
};

int main( int argc, char *argv[] ) {
  SayIntro introduction;
  SayHello speaker[5];
  SayBye   conclusion;
  tbb::pipeline mypipe;

  mypipe.add_filter(introduction);
  mypipe.add_filter(speaker[0]);
  mypipe.add_filter(speaker[1]);
  mypipe.add_filter(speaker[2]);
  mypipe.add_filter(speaker[3]);
  mypipe.add_filter(speaker[4]);
  mypipe.add_filter(conclusion);
  mypipe.run(100);
  mypipe.clear();
}
```

Рис. В.33 ❖ Алгоритм `pipeline`: пример (*начало*)

Пример вывода… – может отличаться от прогона к прогону и от машины к машине!
Признание. Чтобы получить показанный ниже результат, мы умеренно загрузили машину другой работой. Иначе первый конвейер отработал бы до конца, прежде чем второй приступил к работе, поскольку пример слишком прост, – и тогда результат получился бы скучным. Необходимо помнить, что в коде (и в тесте) возможны различные порядки параллельного выполнения.

```
Our pipeline begins at 1042.
Hello, Pipeline 1042
Our pipeline begins at 2042.
Hello, Pipeline 2042
Hello, Pipeline 1043
Hello, Pipeline 2043
Hello, Pipeline 1044
Hello, Pipeline 2044
Hello, Pipeline 1045
Hello, Pipeline 2045
Hello, Pipeline 1046
Value is finally 1047; Good bye!
Hello, Pipeline 2046
Value is finally 2047; Good bye!
```

Рис. B.33 (*окончание*)

Потоковый граф

В главе 3 мы познакомились с потоковым графом, в главе 17 рассмотрели его глубже, а в главах 18 и 19 описали имеющуюся в нем поддержку гетерогенных сред.

Можно создавать как в высшей мере масштабируемые графы, так и полностью последовательные. Граф состоит из компонентов трех видов:

1) объект графа: владелец задач, создаваемых от имени потокового графа. Если пользователь хочет дождаться завершения всех задач, относящихся к выполнению графа, то должен будет ждать именно этот объект. Можно также зарегистрировать внешние взаимодействия с потоковым графом и запускать задачи, владельцем которых останется этот граф;
2) узлы: вызывают заданные пользователем объекты-функции или управляют потоком сообщений от других узлов к другим узлам. Существуют предопределенные типы узлов для буферизации, фильтрации, группового распространения и упорядочения элементов, проходящих через граф;
3) ребра: соединения между узлами. Создаются функцией `make_edge` и удаляются функцией `remove_edge`.

Потоковый граф: класс graph

graph

#include <tbb/flow_graph.h>

```
namespace tbb::flow;
class graph;
```

Примечание: поддерживаются стандартные итераторы (`begin`, `end`, `cbegin`, `cend`), но здесь мы их не описываем.

`graph``([task_group_context& group])`

Конструирует граф без узлов. Если аргумент `group` задан, то задачи графа выполняются в этой группе. По умолчанию граф выполняется в собственном контексте. Создает корневую задачу типа `empty_task`, которая будет служить родителем всех задач, сгенерированных во время выполнения графа. Счетчик ссылок корневой задачи устанавливается равным 1.

`~graph``()`

Вызывает для графа функцию `wait_for_all`, после чего уничтожает корневую задачу.

`void` **`increment_wait_count`**`()`

Используется для регистрации внешней сущности, которая может взаимодействовать с графом. Увеличивает счетчик ссылок корневой задачи.

`void` **`decrement_wait_count`**`()`

Используется для отмены регистрации внешней сущности, которая имела возможность взаимодействовать с графом. Уменьшает счетчик ссылок корневой задачи.

`void` **`run`**`( Receiver& r, Body body )`

Используется для запуска задачи, которая выполняет тело `body` и передает его выход указанному приемнику `r`. Созданная задача является потомком корневой задачи, поэтому функция `wait_for_all` не вернет управление, пока эта задача не завершится. Ставит в очередь задачу, которая вызывает `r.try_put(body)`, и не ждет ее завершения.

`void` **`run`**`( Body body )`

Запускает задачу, которая становится потомком корневой задачи. Функция `wait_for_all` не вернет управление, пока эта задача не завершится. Ставит в очередь задачу, которая вызывает `body()`, и не ждет ее завершения.

`void` **`wait_for_all`**`()`

Блокирует выполнение, пока все задачи, являющиеся потомками корневой, не завершатся и количество вызовов `decrement_wait_count` не станет равно количеству вызовов `increment_wait_count`. Поскольку вызывающий поток вызывает `wait_for_all` из корневой задачи графа, он может участвовать в заимствовании работ, пока находится в заблокированном состоянии.

`task *`**`root_task`**`()`

Возвращает указатель на корневую задачу потокового графа.

`bool` **`is_cancelled`**`()`

Возвращает `true`, если граф был отменен во время последнего обращения к `wait_for_all()`, иначе `false`.

`bool` **`exception_thrown`**`()`

Возвращает `true`, если во время последнего обращения к `wait_for_all()` произошло исключение, иначе `false`. О том, как обрабатываются исключения, см. главу 15.

`void` **`reset`**`(reset_flags f = rf_reset_protocol)`

Флаги, передаваемые `reset()`, можно комбинировать с помощью поразрядного ИЛИ. Определены следующие три флага:

- **`rf_reset_protocol`**: все ребра переключаются в состояние проталкивания, все буферы опустошаются, внутреннее состояние всех узлов повторно инициализируется. Эти действия выполняются при любом обращении к `reset()`;
- **`rf_reset_bodies`**: при создании узла, имеющего тело, заданное в конструкторе тело копируется и сохраняется. Если задан флаг `rf_reset_bodies`, то текущее тело узла удаляется и заменяется телом, сохраненным на этапе конструирования. Предостережение: если в состоянии тела имеются внешние компоненты (например, дескриптор файла), то после замены тела узел может вести себя по-другому при повторном выполнении графа. В таком случае узел следует пересоздать;
- **`rf_clear_edges`**: из графа удаляются все ребра.

Рис. B.34 ❖ Потоковый граф: класс `graph`

Потоковый граф: порты и ребра

Потоковый граф предоставляет API для управления соединениями между узлами. Если узел имеет более одного входного или выходного порта (например, узел типа `join_node`), то для соединения необходимо задать порт с помощью специальных вспомогательных функций.

Порты и ребра

#include <**tbb**/flow_graph.h>

```
namespace tbb::flow;

inline void make_edge(output, input);
inline void remove_edge(output, input);
```

Создать (или удалить) ребро:
1) из порта 0 предшественника с несколькими выходами в порт 0 преемника с несколькими входами;
2) из порта 0 предшественника с несколькими выходами к получателю;
3) из отправителя в порт 0 преемника с несколькими входами.

```
template<size_t N, typename NT>
  typename tuple_element<N,
    typename NT::input_ports_type>::type& input_port(NT &n);
template<size_t N, typename NT>
  typename tuple_element<N,
    typename NT::output_ports_type>::type& output_port(NT &n);
```

Если задана ссылка на узел типа `join_node`, `indexer_node` или `composite_node`, то возвращает ссылку на указанный входной или выходной порт.

Рис. В.35 ❖ Потоковый граф: порты и ребра

Потоковый граф: узлы

Функциональные узлы (рис. В.37) производят вычисления в ответ на выходные сообщения (если таковые имеются) и отправляют результат или сигнал преемникам. На рис. В.38 показаны типы узлов **управления потоком**. Частным случаем узлов управления потоком являются **узлы соединения** (`join_node`), и на рис. В.39 перечислены различные стратегия соединения. **Буферизующие узлы** предназначены для накопления входных сообщений и передачи их преемникам в определенном порядке, зависящем от типа узла. На рис. В.36 показаны значки узлов различных типов.

Некоторые узлы создают или используют сообщения, составленные из других сообщений. Для управления портами многопортовых узлов применяются кортежи. К узлам такого типа относятся `join_node`, `multifunction_node`, `split_node`, `indexer_node` и `composite_node`. Из многопортовых узлов кортежи отправляют или принимают `join_node` и `split_node`.

tbb::flow::tuple или std::tuple?

В настоящее время (когда компиляторы поддерживают C++11 или более поздние версии стандарта) рекомендует использовать класс `std::tuple`. Класс `tbb::flow::tuple` был включен в TBB еще до выхода C++11 и до сих пор поддерживается. Но если в доступной версии STL имеется класс `std::tuple`, то `tbb::flow::tuple` автоматически становится псевдонимом (typedef) `std::tuple`.

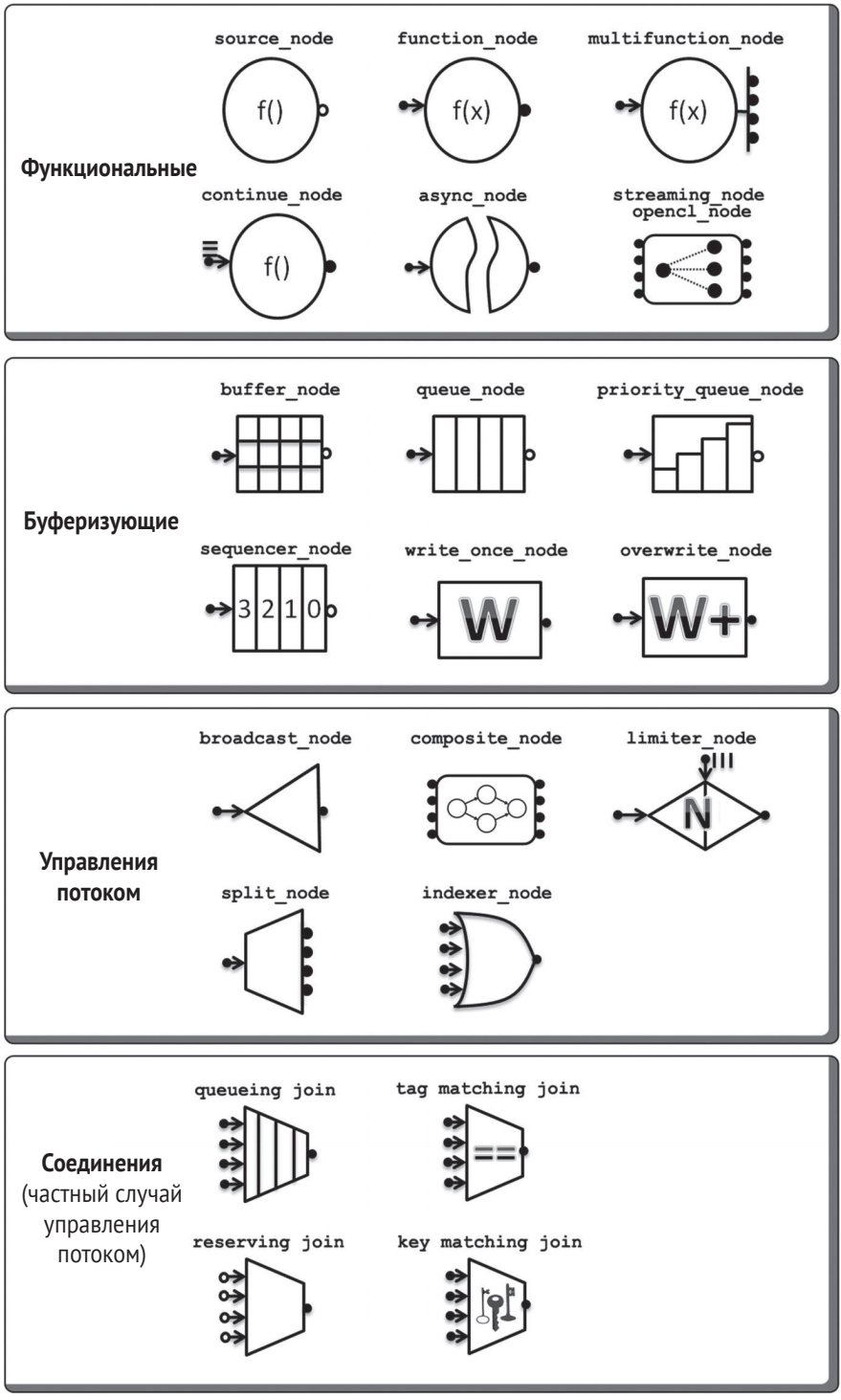


Рис. В.36 ❖ Типы узлов потокового графа (см. также главы 3, 17, 18, 19)

Функциональные типы узлов

source_node

```
template<typename OutType> class source_node;
```

Пример сигнатуры конструктора

```
source_node(graph &g, Body b,
            bool is_active=true)
```

Сигнатура тела

```
[…](OutType& out) -> bool { … }
```

Генерирует сообщения. Если `is_active==true`, то задача Body запускается сразу после конструирования. В противном случае первая задача Body запускается в момент вызова функции `activate`. Новая задача запускается всякий раз, как Body возвращает `true`. Если в результате выполнения Body возвращено `true`, но ни один из узлов-преемников не готов принять значение, то подготовленное значение сохраняется в буфере на один элемент и запуск задачи откладывается до того момента, как это значение будет потреблено. Буфер допускает резервирование (см. главу 17).

function_node

```
template<typename InType [,typename OutType=continue_msg]
         [,typename Policy=queueing]
         [,typename Allocator=cache_aligned_allocator<Input>]
       > class function_node;
```

Пример сигнатуры конструктора

```
function_node(graph &g, size_t,
              concurrency, Body b)
```

Сигнатура тела

```
[…](const InType& in) -> OutType { … }
```

Применяет функцию, генерирующую один выход для каждого полученного входного сообщения. Задача Body запускается, если количество уже работающих задач меньше предельной степени конкурентности. Если предел конкурентности не позволяет запустить новую задачу, то входящее сообщение буферизуется в узле и ждет там, пока количество незавершенных задач не опустится ниже предела.

multifunction_node

```
template<typename InType, typename OutTuple [,typename Policy=queueing]
           [,typename Allocator=cache_aligned_allocator<Input>]
         > class multifunction_node;
```

Пример сигнатуры конструктора

```
multifunction_node(graph &g, size_t
                   concurrency, Body b)
```

Сигнатура тела

```
[…](const InType& in,
    output_ports_type& p) { … }
```

Применяет функцию к каждому полученному сообщению. Задача Body запускается, если количество уже работающих задач меньше предельной степени конкурентности. Если предел конкурентности не позволяет запустить новую задачу, то входящее сообщение буферизуется в узле и ждет там, пока количество незавершенных задач не опустится ниже предела. Body также получает ссылку на кортеж выходных портов. Body отправляет выходные сообщения, явно вызывая функцию `try_put` для выходных портов. Этот узел полезен для реализации условных путей выполнения и циклов в графе.

continue_node

```
template<typename OutType
         [,typename Policy=internal::Policy<void>] > class continue_node;
```

Пример сигнатуры конструктора

```
continue_node(graph &g, size_t,
              initial_count, Body b)
```

Сигнатура тела

```
[…](const continue_msg& in) -> OutType { … }
```

Используется для создания графов зависимостей. Узел подсчитывает количество входящих в него ребер и количество полученных сообщений. Когда количество полученных сообщений становится равно количеству предшественников, он запускает задачу, выполняющую его тело, после чего сбрасывает счетчик полученных сообщений. Конструктору узла можно передать параметр `initial_count` и тем самым увеличить количество сообщений, которые узел должен получить до запуска задачи. Этот параметр используется, если ожидается, что узел будет получать сообщения, явно отправленные с помощью `try_put`, помимо приходящих по ребрам. Задача Body получает единственное сообщение типа `tbb::flow_continue_msg` и выводит одно сообщение типа `OutputType`.

Рис. B.37 ❖ Типы функциональных узлов в интерфейсе потоковых графов. Эти узлы обсуждаются в главе 3, за исключением узла `async_node`, обсуждаемого в главе 18 (*начало*)

async_node

```
template<typename InType, typename OutType
       [,typename Policy=queueing_lightweight]
       [,typename  Allocator=cache_aligned_allocator<Input>]
 > class async_node;
```

Пример сигнатуры конструктора

```
async_node(graph &g, size_t,
          concurrency, Body b)
```

Сигнатура тела

```
[…](const continue_msg& in) -> OutType { … }
```

Позволяет потоковому графу взаимодействовать с внешней сущностью, управляемой пользователем или другой средой выполнения. Этот узел получает сообщения типа `Input` и вызывает Body, чтобы отправить сообщение внешней операции. Чтобы получить результат и перенаправить его в выходной порт, предоставляется шлюз `gateway_type`. Этот тип узла подробно обсуждается в главе 18.

streaming_node и ***opencl_node***

```
template<typename... Args> class streaming_node|opencl_node;
```

или

```
template<typename... Ports, typename JP, typename Factor
        > class streaming_node|opencl_node;
```

Узел типа `streaming_node` дает возможность использовать модели потокового программирования, которые поддерживают передачу вычислительных ядер и данных на устройства с помощью очередей. Тип `opencl_node` – пример `streaming_node` с конкретной моделью, он позволяет использовать устройства, поддерживающие язык OpenCL, и координировать их с помощью потокового графа. Разработчик может самостоятельно определить фабрики для поддержки других моделей. Этот тип узлов рассматривается в главе 19.

Рис. B.37 (*окончание*)

Узлы управления потоком

broadcast_node

```
template<typename InOutType> class broadcast_node;
```

Пример сигнатуры конструктора

```
broadcast_node(graph &g)
```

Рассылает копии каждого входящего сообщения всем своим преемникам. Обычно применяется для организации единой точки входа в граф или подграф, так чтобы один явный вызов `try_put` мог разослать сообщение в несколько узлов. Редко встречается в качестве внутреннего узла графа.

composite_node

```
template<typename InTuple, typename OutTuple> class composite_node;
```

Пример сигнатуры конструктора

```
composite_node(graph &g)
```

Инкапсулирует подграф, содержащий несколько узлов. Имеет один или несколько входных и выходных портов. Предоставляет функции, позволяющие связать входные и выходные порты подграфа с портами входящих в него узлов.

indexer_node

```
template<typename T0, ..., typename T9> class indexer_node;
```

Пример сигнатуры конструктора

```
indexer_node(graph &g)
```

Рассылает всем преемникам каждое сообщение, пришедшее в любой из входных портов, снабжая его тегом, равным номеру порта, в который оно пришло.

limiter_node

```
template<typename InOutType> class limiter_node;
```

Пример сигнатуры конструктора

```
limiter_node(graph &g, size_t limit, size_t initial_count=0)
```

Условно рассылает всем преемникам каждое пришедшее сообщение без каких-либо изменений. Этот узел подсчитывает проходящие через него сообщения. Когда счетчик достигает значения `limit`, рассылка входящих сообщений прекращается. У узла имеется второй порт, служащий для уменьшения счетчика сообщений, что позволяет пропустить дополнительные сообщения. Примеры приведены в главе 17.

Рис. B.38 ❖ Узлы управления потоком в интерфейсе потоковых графов. Эти узлы обсуждаются в главе 3 (*начало*)

split_node

```
template<typename InTuple> class split_node;
```

Пример сигнатуры конструктора

```
split_node(graph &g)
```

Расщепляет кортеж на составные элементы. Имеет единственный входной порт, который получает кортеж значений и направляет элементы кортежа в отдельные выходные порты.

Рис. B.38 (*окончание*)

Узлы соединения

С очередями

```
template<typename OutTuple, queueing> class join_node;
```

Пример сигнатуры конструктора

```
join_node(graph &g)
```

Объединяет сообщения, пришедшие в каждый из входных портов, и создает из них выходной кортеж типа `OutTuple`. Узел `join_node` с очередями ведет очереди в каждом входном порту и создает кортежи, выбирая по одному сообщению из каждой очереди в порядке «первым пришел– первым ушел». Если сгенерированный кортеж не может быть принят узлом-преемником, он буферизуется до тех пор, пока не будет потреблен.

Резервирующий

```
template<typename OutTuple, reserving> class join_node;
```

Пример сигнатуры конструктора

```
join_node(graph &g)
```

Объединяет сообщения, пришедшие в каждый из входных портов, и создает из них выходной кортеж типа `OutTuple`. Резервирующий `join_node` используется в сочетании с буферизующими узлами. Каждый из его входных портов может быть соединен с буферизующим узлом или узлом типа `source_node`. Сообщение из предшествующего узла потребляется только тогда, когда узел может зарезервировать место в каждом порту. Если сгенерированный кортеж не может быть принят узлом-преемником, он буферизуется до тех пор, пока не будет потреблен.

Со сравнением ключей

```
template<typename OutTuple,
         key_matching<typename K, class Khash=tbb_hash_compare<K>>
        > class join_node;
```

Пример сигнатуры конструктора

```
template <typename B0, typename B1, ... >
join_node(graph &g, B0 b0, B1 b1, ...)
```

Объединяет сообщения, пришедшие в каждый из входных портов, и создает из них выходной кортеж типа `OutTuple`. Когда во входной порт приходит сообщение, предоставленный пользователем объект-функция, например `b0`, применяется к нему, чтобы извлечь ключ. Затем сообщение помещается в хеш-таблицу, ассоциированную с входным портом. Когда в каждом входном порту имеется сообщение с данным ключом, узел `join_node` удаляет из всех входных портов такие сообщения, конструирует из них кортеж и пытается разослать его всем своим преемникам. Если сгенерированный кортеж не может быть принят узлом-преемником, он буферизуется до тех пор, пока не будет потреблен.

Со сравнением тегов

```
template<typename OutTuple, tag_matching> class join_node;
```

Пример сигнатуры конструктора

```
template <typename B0, typename B1, ... >
join_node(graph &g, B0 b0, B1 b1, ...)
```

Объединяет сообщения, пришедшие в каждый из входных портов, и создает из них выходной кортеж типа `OutTuple`. Узел `join_node` со сравнением тегов является частным случаем узла со сравнением ключей, когда ключи имеют тип `size_t`. В остальном его поведение такое же, как у узлов со сравнением ключей. Если сгенерированный кортеж не может быть принят узлом-преемником, он буферизуется до тех пор, пока не будет потреблен.

Рис. B.39 ❖ Политики узлов соединения в интерфейсе потоковых графов. Эти узлы обсуждаются в главе 3

Буферизующие узлы

buffer_node

```
template<typename InOutType
         [,typename  Allocator=cache_aligned_allocator<InOutType>]
        > class buffer_node;
```

Пример сигнатуры конструктора

```
buffer_node(graph &g)
```

Неупорядоченный буфер элементов типа `InOutType`. Поддерживает резервирование.

queue_node

```
template<typename InOutType
         [,typename  Allocator=cache_aligned_allocator<InOutType>]
        > class queue_node;
```

Пример сигнатуры конструктора

```
queue_node(graph &g)
```

Очередь элементов типа `InOutType`, обслуживаемая в порядке «первым пришел – первым ушел» (FIFO). Поддерживает резервирование.

priority_queue_node

```
template<typename InOutType[, typename Compare]
         [,typename  Allocator=cache_aligned_allocator<InOutType>]
> class priority_queue_node;
```

Пример сигнатуры конструктора

```
priority_queue_node(graph &g)
```

Очередь элементов типа `InOutType`, обслуживаемая в порядке, определяемом типом `Compare`. Поддерживает резервирование.

sequencer_node

```
template<typename InOutType
         [,typename  Allocator=cache_aligned_allocator<InOutType>]
> class sequencer_node;
```

Пример сигнатуры конструктора

```
sequencer_node(graph &g, const Sequencer& s)
```

Буфер элементов типа `InOutType`, упорядоченный по порядковому номеру. Порядковый номер выделяется из каждого сообщения применением к нему функтора `Sequencer s`. Элементы отправляются дальше строго в порядке возрастания порядкового номера – сообщение хранится в буфере до тех пор, пока не будут получены и разосланы все сообщения с меньшими номерами. Поддерживает резервирование.

write_once_node

```
template<typename InOutType> class write_once_node;
```

Пример сигнатуры конструктора

```
write_once_node(graph &g)
```

Буфер на один элемент с однократной записью. Может быть сброшен, после чего разрешено записать в буфер новое значение. Если в тот момент, когда узел содержит значение, провести ребро между ним и каким-нибудь другим узлом, то значение будет отправлено по новому ребру этому узлу. После отправки значение не уничтожается, а остается в буфере и потому может быть передано узлам, которые присоединятся позднее. Этот узел полезен в графах, которые могут расти во время выполнения. Поддерживает резервирование.

overwrite_node

```
template<typename InOutType> class overwrite_node;
```

Пример сигнатуры конструктора

```
overwrite_node(graph &g)
```

Этот узел похож на `write_once_node`, но допускает многократную запись. Если в тот момент, когда узел содержит значение, провести ребро между ним и каким-нибудь другим узлом, то значение будет отправлено по новому ребру этому узлу. После отправки значение не уничтожается, а остается в буфере и потому может быть передано узлам, которые присоединятся позднее. Узлы, соединенные ребром с выходом этого узла, получают все значения, записанные в буфер после соединения. Поддерживает резервирование.

Рис. В.40 ❖ Типы буферизующих узлов в интерфейсе потоковых графов. Эти узлы обсуждаются в главе 3

Политики выполнения узлов графа

Дадим некоторые рекомендации, связанные с планированием задач для функциональных узлов графа. Облегченные политики для функциональных узлов помогают уменьшить накладные расходы, связанные с планированием выполнения. Их следует использовать только после тщательной оценки конкретного узла. Если для нескольких преемников одного узла задана облегченная политика, то степень параллелизма в графе может значительно уменьшиться, что негативно отразится на масштабируемости. Циклы в потоковом графе, состоящие *только* из узлов с облегченной политикой, могут привести к взаимоблокировке.

Политика `lightweight` применяется, когда нужно указать, что объем работы в теле узла невелик и по возможности его следует выполнять без накладных расходов на планирование задачи. Все функциональные узлы потоковых графов, кроме `source_node`, поддерживают политику `lightweight` как одно из возможных значений необязательного параметра шаблона `Policy`. Чтобы воспользоваться политикой `lightweight`, задайте в параметре шаблона `Policy` значение `queueing_lightweight`, `rejecting_lightweight` или `lightweight`. Если в функциональном узле параметр шаблона `Policy` имеет значение по умолчанию, то задание политики `lightweight` приводит к дополнению поведения, определяемого политикой по умолчанию, поведением, определяемым политикой `lightweight`. Например, если по умолчанию подразумевается политика `queueing`, то задание в качестве значения параметра `Policy` политики `lightweight` эквивалентно заданию политики `queueing_lightweight`. Подробное обсуждение облегченных политик см. в главе 17.

Значения параметра `Policy` перечислены на рис. В.41. Отметим, что политика `tbb::flow::reserving` не включена, потому что это специальная политика, предназначенная только для узлов типа `join_node` и не применимая к узлам типа `async_node`, `continue_node`, `function_node` и `multifunction_node`.

Политики выполнения узлов

Применяются к узлам типа `async_node`, `continue_node`, `function_node` и `multifunction_node`.

`tbb::flow::queueing`

Политика по умолчанию для всех узлов, кроме `async_node`. Если в узел приходит сообщение, но степень конкурентности не позволяет сразу же запустить новую задачу для выполнения его тела, то сообщение буферизуется до тех пор, пока не представится возможность запустить задачу.

`tbb::flow::rejecting`

Если в узел приходит сообщение, но степень конкурентности не позволяет сразу же запустить новую задачу для выполнения его тела, то вызов `try_put` возвращает `false` и сообщение не буферизуется. Эту политику можно использовать, чтобы не дать предшествующему узлу `source_node` генерировать новые сообщения.

`tbb::flow::lightweight`

Политика **`lightweight`** означает, что объем работы в теле узла невелик, и по возможности его следует выполнять без накладных расходов на планирование задачи. Обычно тело выполняется в момент обращения к `try_put`. Если в функциональном узле параметр шаблона `Policy` имеет значение по умолчанию, то задание политики `lightweight` приводит к дополнению поведения, определяемого политикой по умолчанию, поведением, определяемым политикой `lightweight`. Например, если по умолчанию подразумевается политика `queueing`, то задание в качестве значения параметра `Policy` политики `lightweight` эквивалентно заданию политики `queueing_lightweight`.

Рис. В.41 ❖ Потоковый граф: политики выполнения узлов (*начало*)

`tbb::flow::queueing_lightweight`

Политика по умолчанию для узлов `async_node`. Если в узел приходит сообщение, но степень конкурентности не позволяет применить тело немедленно, то сообщение буферизуется до тех пор, пока не представится возможность применить тело. Поскольку это облегченная политика, для выполнения тела обычно не запускается новая задача.

`tbb::flow::rejecting_lightweight`

Если в узел приходит сообщение, но степень конкурентности не позволяет применить тело немедленно, то вызов `try_put` возвращает `false` и сообщение не буферизуется. Если тело разрешено выполнять, то оно выполняется прямо внутри обращения к `try_put`.

Рис. B.41 (*окончание*)

Функции-члены узлов

#include <tbb/`flow_graph.h`>

`namespace tbb::flow`

Примечания:

1) функции возвращают `true`, если явно не оговорено противное (`false` обычно означает, что операция не завершена);
2) функция возвращает управление, не дожидаясь завершения выполнения тела.

`bool activate()`

Поддерживается только для:
`source_node`: переводит узел в активное состояние, в котором разрешено генерировать сообщения.

`bool try_put( const input_type &v )`

`async_node`: запускает задачу, которая выполняет тело (`v`).
`broadcast_node`: рассылает `v` всем преемникам.
`buffer_node`: помещает `v` в буфер. Если `v` – единственный элемент в буфере, то также запускается задача, которая переправляет этот элемент преемнику.
`continue_node`: увеличивает счетчик полученных вызовов `try_put`. Когда счетчик сравнивается с количеством известных предшественников, запускается задача для выполнения тела, после чего счетчик сбрасывается в 0.
`function_node`: рассылка запускается немедленно, если предел конкурентности еще не превышен, в противном случае сообщение ставится в очередь, где ждет, пока степень конкурентности не опустится до уровня, разрешающего рассылку. Возвращает `false`, если сообщение было отвергнуто (такое может случиться, только если достигнут предел конкурентности и задана политика `rejecting` вместо подразумеваемой по умолчанию `queueing`).
`limiter_node`: если счетчик рассылок (`C`) меньше порогового, то `v` рассылается всем преемникам и `C` увеличивается на единицу. Если ни один преемник не принял сообщение, то `C` уменьшается на единицу. Возвращает `true`, если сообщение было успешно доставлено хотя бы одному преемнику, и `false` в противном случае.
`multifunction_node`: запускает тело немедленно, если предел конкурентности еще не превышен, в противном случае отвергает входное сообщение и возвращает `false`.
`overwrite_node` и **`write_once_node`:** сохраняет `v` во внутреннем буфере на один элемент и вызывает `try_put(v)` для каждого преемника.
`sequencer_node`: добавляет `v` в `sequencer_node`. Если порядковый номер `v` совпадает со следующим номером в последовательности, то запускается задача, отправляющая сообщение преемнику.
`split_node`: разбирает входящий кортеж на элементы и рассылает элементы узлам, соединенным с выходными портами. Элемент кортежа `v` с индексом `i` будет отправлен через `i`-й выходной порт.

`bool try_get( output_type &v )`

Недоступна для узла типа `split_node`, потому что он не знает, о каком порте идет речь; технически можно было бы использовать порт, заданный функцией `output_port()`, но поскольку узел небуферизующий, то мы бы получили просто `false` и были бы страшно разочарованы.
Возвращает `false` для узлов, не поддерживающих буферизацию вывода (`async_node`, `continue_node`, `function_node`, `indexer_node`, `limiter_node`).
Копирует буферизованное сообщение в `v`, если это возможно (сообщение может быть получено и не зарезервировано). Если доступных сообщений нет и в узле имеется тело, то тело будет вызвано в попытке сгенерировать новое сообщение, которое можно будет скопировать в `v`. Возвращает `true`, если сообщение скопировано в `v`, в противном случае `false`.

Рис. B.42 ❖ Потоковый граф: функции-члены узлов (*начало*)

bool try_reserve(output_type &v)

Возвращает false для узлов, не поддерживающих резервирование (async_node, broadcast_node, continue_node, function_node, indexer_node, join_node, limiter_node, split_node).
overwrite_node и **write_once_node:** копирует внутренний буфер в v и возвращает true, если внутренний буфер действителен, иначе возвращает false.
Все остальные типы узлов: устанавливает режим резервирования, если возможно. Это означает, что любое потребление буферов (в т. ч. отправка преемникам) приостанавливается до отмены резервирования. Если сообщение может быть буферизовано и узел еще не зарезервирован, то узел резервируется для вызывающей стороны, и значение копируется в v. Возвращает true, если узел (буфер) зарезервирован для вызывающей стороны, в противном случае false.

bool try_release()

Возвращает false для узлов, не поддерживающих резервирование (async_node, broadcast_node, continue_node, function_node, indexer_node, join_node, limiter_node, split_node).
overwrite_node и **write_once_node:** возвращает true.
Все остальные типы узлов: отменяет установленное резервирование. Сообщение, находящееся во внутреннем буфере, сохраняется. Возвращает true, если резервирование существовало, в противном случае false.

bool try_consume()

Возвращает false для узлов, не поддерживающих резервирование (async_node, broadcast_node, continue_node, function_node, indexer_node, join_node, limiter_node, split_node).
overwrite_node и **write_once_node:** возвращает true.
Все остальные типы узлов: отменяет установленное резервирование и очищает верхней элемент во внутреннем буфере. Возвращает true, если резервирование существовало, в противном случае false.

bool is_valid()

Доступна только для:
overwrite_node и **write_once_node** – возвращает true, если в буфере находится значение, в противном случае false.

bool clear()

Доступна только для:
overwrite_node и **write_once_node** – делает значение в буфере недействительным.

input_ports_type &input_ports()

Доступна только для:
indexer_node – возвращает кортеж получателей.
composite_node – возвращает кортеж получателей. Каждый элемент является ссылкой на сам узел или входной порт, который был ассоциирован с соответствующей позицией в результате вызова функции set_external_ports(). Вызов **input_ports()** без предварительного вызова set_external_ports() считается неопределенным поведением.
join_node – возвращает кортеж получателей. Поведение портов зависит от выбора политики буферизации (queueing, reserving, key_matching, tag_matching). См. главу 3.

input port(JNT &jn)

Доступна только для:
join_node – возвращает N-й входной порт для join_node jn.

output_ports_type &output_ports()

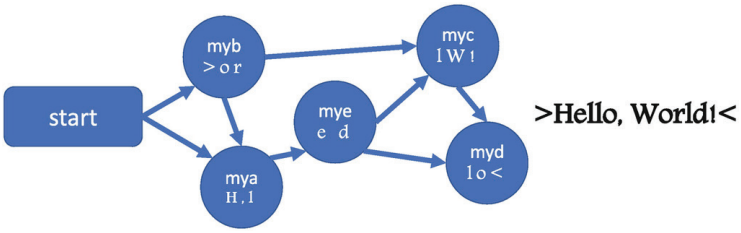
Доступна только для:
composite_node – возвращает кортеж получателей. Каждый элемент является ссылкой на сам узел или выходной порт, который был ассоциирован с соответствующей позицией в результате вызова функции set_external_ports(). Вызов **output_ports()** без предварительного вызова set_external_ports() считается неопределенным поведением.
multifunction_node и **split_node** – возвращает кортеж выходных портов.

set_external_ports(input_ports_type&& in_ports_tuple, output_ports_type&& out_ports_tuple)

Доступна только для:
composite_node – создает входные и выходные порты узла **composite_node** как псевдонимы портов, на которые ссылаются элементы кортежей **in_ports_tuple** и **out_ports_tuple** соответственно. Порт в позиции N кортежа **in_ports_tuple** отображается на N-й входной порт **composite_node**, и аналогично для выходных портов.

Рис. B.42 (*окончание*)

Потоковый граф – Hello, World




```
#include <cstdio>
#include <tbb/tbb.h>

tbb::spin_mutex mylock;
int counter = 0;

struct hw {
  char myhw1, myhw2, myhw3;
  hw(const char hw1,const char hw2,const char hw3) :
      myhw1(hw1),myhw2(hw2),myhw3(hw3) {}
  void operator()(tbb::flow::continue_msg) const {
    int mycount;
    { tbb::spin_mutex::scoped_lock hello(mylock);
      mycount = counter++;
    }
    printf("%c",
           (mycount < 5) ? myhw1 : (mycount < 10) ? myhw2 : myhw3);
  }
};

int main() {
  tbb::flow::graph g;

  tbb::flow::broadcast_node<tbb::flow::continue_msg> start(g);
#define X tbb::flow::continue_node<tbb::flow::continue_msg>
  X mya(g,hw('H',',','l'));
  X myb(g,hw('>','o','r'));
  X myc(g,hw('l','W','!'));
  X myd(g,hw('l','o','<'));
  X mye(g,hw('e',' ','d'));

  tbb::flow::make_edge(start, mya);
  tbb::flow::make_edge(start, myb);
  tbb::flow::make_edge(myb, mya);
  tbb::flow::make_edge(myb, myc);
  tbb::flow::make_edge(myc, myd);
  tbb::flow::make_edge(mya, mye);
  tbb::flow::make_edge(mye, myc);

  for (int i = 0; i < 3; ++i) {
    start.try_put(tbb::flow::continue_msg());
    g.wait_for_all();
  }
  printf("\n");
  return 0;
}
```

Этот результат детерминированный! На самом деле два ребра не нужны – их можно было бы опустить, и результат все равно остался бы детерминированным. Остальные ребра нужны для сохранения сообщения – можно было бы ради эксперимента попробовать удалить несколько ребер, и тогда мы получили бы недетерминированный результат, так что в одном прогоне все будет правильно, а в другом появится орфографическая ошибка.

```
>Hello, World!<
```

Рис. В.43 ❖ Потоковый граф: пример

Выделение памяти

В главе 7 полностью рассмотрены все распределители памяти, имеющиеся в TBB; здесь же приводится краткий обзор поддерживаемых API с примечаниями. Распределители памяти, входящие в состав TBB, не зависят от других частей библиотеки, поэтому их можно использовать совместно с любой моделью многопоточного программирования.

Шаблонные классы распределителей памяти (всего пять)

Первые четыре класса моделируют концепцию распределителя (Allocator) – все они подходят для замены **`std::allocator`**. Если переменной среды TBB_VERSION присвоено значение 1, то на stderr выводится полезная информация, из которой можно узнать, используется ли библиотека Intel TBB malloc (см. раздел «Отладка и условный код» в начале этого приложения).

```
#include <tbb/tbb_allocator.h>
template<typename T> class tbb_allocator
```

Шаблонный класс для масштабируемого выделения памяти, если имеется подходящий распределитель, в противном случае может быть немасштабируемым. `tbb_allocator` выделяет и освобождает память с помощью библиотеки TBB malloc, если она доступна, иначе обращается к стандартным функциям `malloc` и `free`.

```
#include <tbb/tbb_scalable_allocator.h>
template<typename T> class scalable_allocator
```

Шаблонный класс для масштабируемого выделения памяти (принудительно). `scalable_allocator` выделяет и освобождает память способом, масштабируемым на имеющееся количество процессоров. Использование `scalable_allocator` вместо `std::allocator` может повысить производительность программы. Память, выделенную с помощью `scalable_allocator`, необходимо освобождать с его же помощью, а не средствами `std::allocator`. Для использования `scalable_allocator` должна присутствовать соответствующая библиотека выделения памяти. Если ее нет, то обращение к `scalable_allocator` завершится ошибкой, тогда как `tbb_allocator` в этом случае прибегает к помощи стандартных функций `malloc` и `free`.

```
#include <tbb/cache_aligned_allocator.h>
template<typename T> class cache_aligned_allocator
```

Шаблонный класс для выделения памяти способом, предотвращающим ложное разделение. Для этой цели `cache_aligned_allocator` выделяет память на границе строк кеша. Ложное разделение возникает, когда логически не связанные элементы оказываются в одной и той же строке кеша, что может негативно отразиться на производительности, если несколько потоков попытаются одновременно обратиться к различным элементам в этой строке. Хотя элементы логически не связаны, процессорам придется перенести всю строку из одного кеша в другой. В результате трафик в памяти оказывается гораздо интенсивнее, чем в случае, когда логически не связанные элементы размещаются в разных строках кеша.

Этот распределитель можно использовать вместо `std::allocator`. При осмотрительном использовании `cache_aligned_allocator` может повысить производительность, уменьшив частоту ложного разделения. Но иногда он оказывает негативное действие. За выделение памяти на границе строки кеша приходится расплачиваться неявным дополнением областей памяти, обычно до кратного 128 байтам. Поэтому при создании большого количества малых объектов с помощью `cache_aligned_allocator` может возрасти потребление памяти.

```
#include <tbb/tbb_allocator.h>
template<typename T> class zero_allocator
```

Шаблонный класс для выделения обнуленной памяти. `zero_allocator<T,A>` можно конкретизировать любым классом `A`, моделирующим концепцию распределителя. По умолчанию подразумевается, что `A` совпадает с `tbb_allocator`. Класс `zero_allocator` переправляет запросы на выделение классу `A` и обнуляет полученную память, прежде чем вернуть ее вызывающей стороне.

```
#include <tbb/aligned_space.h>
template<typename T> class aligned_space
```

Шаблонный класс для выделения неинициализированной памяти для массива заданного типа. НЕ моделирует концепцию распределителя, поэтому его нельзя использовать вместо `std::allocator` или совместно с `zero_allocator`. Область, выделенная `aligned_space`, занимает достаточно памяти и правильно выровнена для размещения массива `T[N]`. За инициализацию и уничтожение объектов в этой памяти отвечает клиент. Обычно `aligned_space` используется для локальных переменных или полей в ситуациях, когда нужен блок неинициализированной памяти фиксированной длины.

Рис. В.44 ❖ Выделение памяти: шаблонные классы распределителей памяти

Функции выделения памяти (интерфейсы с языком C)

Описанные ниже функции составляют C-интерфейс к масштабируемому распределителю. Каждая функция `scalable_x` ведет себя как соответствующая библиотечная функция `x`. Память, выделенная функцией `scalable_x`, должна быть освобождена или перераспределена функцией из того же семейства, а не функцией из стандартной библиотеки C. Аналогичная память, выделенная функцией из стандартной библиотеки C, должна быть освобождена или перераспределена функцией из той же библиотеки, а не функцией `scalable_x`. Присвоив переменной среды TBB_VERSION значение 1, вы получите полезную для отладки информацию.

```
#include <tbb/tbb_allocator.h>
void scalable_aligned_free (void * ptr)
```

Аналог «`_aligned_free`».

```
void* scalable_aligned_malloc (size_t size,
                               size_t alignment)
```

Аналог «`_aligned_realloc`».

```
void* scalable_aligned_realloc (void * ptr,
                                size_t size,
                                size_t alignment)
```

Аналог «`_aligned_realloc`».

```
void* scalable_calloc (size_t nobj, size_t size)
```

Аналог «`calloc`», дополняющий `scalable_malloc`.

```
void scalable_free (void * ptr)
```

Аналог «`free`» для освобождения ранее выделенной памяти.

```
void* scalable_malloc (size_t size)
```

Аналог «`malloc`» для выделения блока памяти размером `size` байт.

```
size_t scalable_msize (void * ptr)
```

Аналог `msize/malloc_size/malloc_usable_size`. Возвращает размер доступной для использования части блока памяти, выделенного ранее функцией `scalable_x`, или 0, если `ptr` не указывает на такой блок.

```
int scalable_posix_memalign (void ** memptr,
                             size_t alignment,
                             size_t size)
```

Аналог «`posix_memalign`».

```
void* scalable_realloc (void * ptr, size_t size)
```

Аналог «`realloc`», дополняющий `scalable_malloc`.

Рис. B.45 ❖ Выделение памяти: функции выделения памяти

Средства управления распределителем памяти (C-интерфейсы)

Описанные ниже функции можно использовать для настройки поведения масштабируемого распределителя памяти.

#include <**tbb**/tbb_allocator.h>

```
int scalable_allocation_mode (int param, intptr_t value)
```

scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 1)
Сообщает распределителю, что можно использовать большие страницы, если это разрешено операционной системой. Присваивание переменной среды TBBMALLOC_USE_HUGE_PAGES дает такой же эффект. Режим, установленный функцией scalable_allocation_mode(), переопределяет действие переменной среды.
scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES, 0)
Отменяет предыдущий режим.
Обе функции возвращают TBBMALLOC_NO_EFFECT, если большие страницы не поддерживаются на данной платформе.
По состоянию на 2019 год TBB поддерживала большие страницы в ОС Linux. Работает как с явно сконфигурированными, так и с прозрачными большими страницами. Если хотите, чтобы TBB могла использовать большие страницы, не забудьте включить и настроить их в ОС.
ПРЕДОСТЕРЕЖЕНИЕ: при работе с большими страницами не рекомендуется округлять с избытком размер запрашиваемой памяти или выравнивать память на границу, кратную размеру большой страницы, поскольку это может привести к неэффективному использованию памяти и снижению производительности.
scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)
Раздает пороговый размер в байтах количества памяти, которую распределитель может запросить у ОС. Превышение порога заставляет распределитель освободить свои внутренние буферы, но не препятствует запросу большего количества памяти, если это необходимо.

```
int scalable_allocation_command (int cmd, void * param)
```

Второй параметр зарезервирован и должен быть равен 0.
scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)
Очищает внутренние буферы памяти и, возможно, уменьшает объем занятой памяти. В результате время выполнения последующих операций выделения памяти может возрасти. Команда не предназначена для частого использования, рекомендуется тщательно оценивать ее влияние на производительность. ПРИМЕЧАНИЕ: не гарантируется, что будет освобождена вся неиспользуемая память.
scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)
Очищает внутренние буферы памяти, но только для вызывающего потока. В результате время выполнения последующих операций выделения памяти может возрасти; рекомендуется тщательно оценивать влияние команды на производительность.
Эти функции возвращают TBBMALLOC_NO_EFFECT, если ни один буфер не был освобожден.

Рис. B.46 ❖ Выделение памяти: средства управления распределителем памяти (C-интерфейсы)

Концепция распределителя

#include <tbb/tbb_allocator.h>

tbb_allocator, **scalable_allocator**, **cache_aligned_allocator** и **zero_allocator** моделируют концепцию распределителя. Эта концепция в TBB похожа на «Требования к распределителю» в таблице 32 стандарта ISO C++ (2003), но с дополнительными гарантиями, наличия которых стандарт требует для контейнеров ISO C++. В таблице ниже описана концепция распределителя. Здесь `A` и `B` представляют экземпляры класса распределителя.

`typedef T* A::pointer`
Указатель на `T`.
`typedef const T* A::const_pointer`
Указатель на `const T`.
`typedef T& A::reference`
Ссылка на `T`.
`typedef const T& A::const_reference`
Ссылка на `const T`.
`typedef T A::value_type`
Тип подлежащего выделению значения.
`typedef size_t A::size_type`
Тип для представления количества значений.
`typedef ptrdiff_t A::difference_type`
Тип для представления разности указателей.
`template<typename U> struct rebind { typedef A<U> A::other; };`
Привязка к другому типу.
`A() throw()`
Конструктор по умолчанию.
`A( const A& ) throw()`
Копирующий конструктор.
`template<typename U> A( const A& )`
Перепривязывающий конструктор.
`~A() throw()`
Деструктор.
`T* A::address( T& x ) const`
Получение адреса.
`const T* A::const_address( const T& x ) const`
Получение константного адреса.
`T* A::allocate( size_type n, const void* hint=0 )`
Выделение памяти для `n` значений.
`void* A::deallocate( T* p, size_t n )`
Освобождение памяти, занятой `n` значениями.
`size_type A::max_size( ) const throw()`
Максимально допустимый аргумент метода `allocate`.

`void A::construct( T* p, const T& value )`
`new(p) T(value)`
`void A::destroy( T* p )`
`p->T::~T()`
`bool operator==( const A&, const B& )`

Возвращает `true`, если `A` равно `B`.
`bool operator!=( const A&, const B& )`
Возвращает `false`, если `A` равно `B`.

Рис. В.47 ❖ Выделение памяти: концепция распределителя

Шаблонные классы `memory_pool` и `fixed_pool`

`#define` TBB_PREVIEW_MEMORY_POOL 1

`#include` <**`tbb`**/memory_pool.h>

Как `memory_pool`, так и `fixed_pool` выделяют и освобождают память способом, масштабируемым на количество процессоров; оба моделируют концепцию пула памяти (Memory Pool) (см. рис. В.49). Шаблонные классы для масштабируемого выделения памяти обслуживаются базовым распределителем. Класс `fixed_pool` получает всю доступную для выделения память от базового распределителя в момент конструирования, и затем пул доступной памяти не растет. Класс `memory_pool` получает память от базового распределителя большими порциями, и при необходимости может быть запрошена дополнительная память. Предостережение: если базовый распределитель обращается к другому масштабируемому пулу памяти, то внутренний пул (или пулы) должен быть уничтожен до того, как будет уничтожен или рециклирован внешний пул.

```
explicit fixed_pool(const Alloc &src = Alloc())
```

Конструирует пул памяти с помощью экземпляра базового распределителя типа `Alloc`, скопированного из `src`. Возбуждает исключение `bad_alloc`, если исполняющая среда не смогла сконструировать экземпляр класса.

```
// Пример выделения памяти из фиксированного пула
#define TBB_PREVIEW_MEMORY_POOL 1
#include <tbb/memory_pool.h>
...
char buf[512*1024];
tbb::fixed_pool my_pool(buf, 512*1024);
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);
```

```
explicit memory_pool(const Alloc &src = Alloc())
```

Конструирует пул памяти с помощью экземпляра базового распределителя типа `Alloc`, скопированного из `src`. Возбуждает исключение `bad_alloc`, если исполняющая среда не смогла сконструировать экземпляр класса.

```
// Пример выделения памяти из расширяемого пула
#define TBB_PREVIEW_MEMORY_POOL 1
#include <tbb/memory_pool.h>
...
tbb::memory_pool<std::allocator<char> > my_pool;
void* my_ptr = my_pool.malloc(10);
my_pool.free(my_ptr);
```

Рис. В.48 ❖ Выделение памяти: шаблонные классы `memory_pool` и `fixed_pool`

Концепция пула памяти

`#define` TBB_PREVIEW_MEMORY_POOL 1

`#include` <**`tbb`**/memory_pool.h>

Пулы памяти выделяют и освобождают память из заданной области или с помощью базового распределителя, предоставляющего потокобезопасные масштабируемые операции. Здесь `P` обозначает экземпляр класса пула памяти. Шаблонный класс **`memory_pool`** и класс **`fixed_pool`** моделируют концепцию пула памяти.

```
~P() throw()
```

Деструктор. Освобождает всю память, занятую выделенными объектами.

```
void P::recycle();
```

Освобождает всю память, занятую выделенными объектами.

```
void* P::malloc(size_t n);
```

Возвращает указатель на `n` байт, выделенных из пула памяти.

```
void P::free(void* ptr);
```

Освобождает память, на которую указывает `ptr`.

```
void* P::realloc( vpod* ptr, size_t n )
```

Перераспределяет память, на которую указывает `ptr`, делая ее размером `n` байт.

Рис. В.49 ❖ Выделение памяти: концепция пула памяти

Контейнеры

В главе 6 полностью рассмотрены классы конкурентных контейнеров в TBB, в этом разделе приводится краткий обзор поддерживаемых API с примечаниями. Классы конкурентных контейнеров, предоставляемые TBB, не зависят от остальной части библиотеки, поэтому могут использоваться с любой моделью многопоточного программирования. Примеры использования имеются в главе 6, поэтому здесь не приводятся.

Типичный STL-контейнер в C++ не допускает конкурентного обновления, такие попытки часто приводят к повреждению контейнера. STL-контейнеры можно обернуть мьютексом, сделав их безопасными для конкурентного доступа, поскольку в каждый момент времени с контейнером сможет работать только один поток, но такой подход исключает всякую конкурентность, а значит, ограничивает ускорение, достигаемое распараллеливанием.

Поэтому TBB предлагает конкурентные контейнеры, допускающие одновременный доступ нескольких потоков для обновления элементов в контейнере. Для обеспечения конкурентности в TBB используется либо мелкозернистая блокировка, либо безблокировочные методы. За это приходится расплачиваться чуть более высокими накладными расходами, чем в обыкновенных STL-контейнерах. Поэтому использовать конкурентные контейнеры имеет смысл, только когда ускорение вследствие дополнительной конкурентности перевешивает замедление выполнения в последовательной программе.

Как и для большинства объектов в C++, конструктор или деструктор объекта контейнера не должен вызываться одновременно с другой операцией над этим объектом. В противном случае возникающая гонка может привести к тому, что операция будет применена к объекту в неопределенном состоянии.

Имя класса и замечания, касающиеся C++11	Конкурентный обход и вставка	С ключами ассоциированы значения	Поддержка конкурентного стирания	Встроенная блокировка	Отсутствие видимой блокировки (безблокировочный интерфейс)	Разрешена вставка одинаковых элементов	Акцессоры [] и at
`concurrent_hash_map` Предшествует C++11	✓	✓	✓	✓	×	×	×
`concurrent_unordered_map` *Очень похож на unordered_map* в *C++11*	✓	✓	×	×	✓	×	
`concurrent_unordered_multimap` *Очень похож на unordered_multimap* в *C++11*	✓	✓	×	×	✓	×	
`concurrent_unordered_set` *Очень похож на unordered_set* в *C++11*	✓	×	×	×	✓	×	×
`concurrent_unordered_multiset` *Очень похож на unordered_multiset* в *C++11*	✓	×	×	×	✓	✓	×

Рис. B.50 ❖ Контейнеры: сравнения классов отображения

Класс конкурентного хеш-отображения (хеш-таблицы)

#include <**tbb**/concurrent_hash_map.h>

Класс tbb::concurrent_hash_map.h отображает ключи на значения таким образом, что несколько потоков могут одновременно обращаться к значениям для поиска, вставки и стирания. Отвечает требованиям к контейнерам в стандарте ISO C++. Примеры использования класса concurrent_hash_map см. в главе 6.

Синтаксис конструктора

```
concurrent_hash_map<typename Key,
                    typename T,
                    [ , typename HashCompare = tbb_hash_compare<Key> ]
                    [ , typename A=tbb::tbb_allocator<std::pair<Key, T> ] > >
```

Конкурентный доступ: ***акцессор***

Для параллельного использования хеш-отображения (из нескольких задач или потоков) TBB предоставляет классы const_accessor *и* accessor*, называемые акцессорами. Акцессор действует как интеллектуальный указатель на пару, хранящуюся в* concurrent_hash_map*. Он удерживает неявную блокировку пары до тех пор, пока объект не будет уничтожен или не будет вызван метод акцессора* release().

Класс	value_type	Неявная блокировка на pair
const_accessor	const std::pair< const Key,T>	Читательская блокировка – разрешает доступ совместно с другими читателями
accessor	std::pair< const Key,T>	Писательская блокировка – разрешает только монопольный доступ одному потоку. Блокирует доступ со стороны других потоков.

Член (const_accessor и accessor)	**Описание**
bool **empty**() const	Возвращает true, если экземпляр ни на что не указывает, и false, если он указывает на пару ключ/значение
void **release**()	Если !empty(), то освобождает неявную блокировку пары, после чего экземпляр больше ни на что не указывает. В противном случае не делает ничего
const value_type& **operator***() const	Генерирует ошибку утверждения, если empty() и константа препроцессора TAB_USE_ASSERT отлична от нуля. Возвращает константную ссылку на пару ключ/значение
const value_type* **operator->**() const	Возвращает &operator*()
const_accessor()	Конструирует экземпляр const_accessor, который ни на что не указывает
~const_accessor()	Если экземпляр указывает на пару ключ/значение, то освобождает неявную блокировку
Член (только accessor**)**	**Описание**
value_type& **operator***() const	Генерирует ошибку утверждения, если empty() и константа препроцессора TAB_USE_ASSERT отлична от нуля. Возвращает константную ссылку на пару ключ/значение
value_type* **operator->**() const	Возвращает &operator*()

Рис. В.51 ❖ Контейнеры: класс конкурентного хеш-отображения (хеш-таблицы)

Конкурентное хеш-отображение...	
Операции над таблицей в целом	
Член	**Описание**
`~concurrent_hash_map()`	Вызывает `clear()`. Этот метод небезопасно выполнять одновременно с другими методами того же самого конкурентного хеш-отображения
`concurrent_hash_map& operator= ( concurrent_hash_map& source )`	Если исходная и конечная (`this`) таблицы различны, то очищает конечную таблицу и копирует в нее все пары ключ/значение из исходной. В противном случае ничего не делает. Возвращает ссылку на `*this`
`concurrent_hash_map& operator= ( concurrent_hash_map&& source )`	Перемещает данные из исходной таблицы в `*this`. Исходная таблица остается в неопределенном состоянии, но может быть безопасно уничтожена
`concurrent_hash_map& operator= (std::inititalizer_list<value_type> il)`	В `*this` записываются данные из `il`. Возвращает ссылку на `*this`
`void swap(concurrent_hash_map&table)`	Обменивает содержимое и распределители `this` и `table`
`void rehash( size_type n = 0 )`	Внутри таблица разбита на ячейки. Метод `rehash` реорганизует эти ячейки, так чтобы повысить производительность будущих операций поиска. Увеличивает количество ячеек до `n`, если `n > 0` и `n` больше текущего числа ячеек. ПРЕДОСТЕРЕЖЕНИЕ: в текущей реализации количество ячеек никогда не уменьшается. В будущих реализациях оно, возможно, будет уменьшаться, если `n` меньше текущего числа ячеек. Примечание: отношение количества элементов к количеству ячеек влияет на пространственно-временные характеристики хеш-таблицы. Если оно велико, то экономится память в ущерб времени поиска. Если мало, то наоборот. По умолчанию это отношение в среднем равно от 0.5 до 1 элемента на ячейку
`void clear()`	Удаляет все пары ключ/значение из таблицы. Не хеширует и не сравнивает ключи. Если константа препроцессора `TBB_USE_PERFORMANCE_WARNINGS` отлична от нуля, то выдается предупреждение, если функция хеширования недостаточно случайна, и это может существенно снизить производительность
`allocator_type get_cllocator() const`	Возвращает копию распределителя, который использовался при конструировании таблицы

Рис. B.51 (*продолжение*)

Конкурентное хеш-отображение...	
Конкурентные операции	
Член	**Описание**
`size_type count( const Key& key ) const`	ПРЕДОСТЕРЕЖЕНИЕ: этот метод может сделать недействительными ранее полученные итераторы. В последовательном коде можно использовать метод `equal_range`, не вызывающий таких проблем. Возвращает 1, если отображение содержит ключ `key`, иначе 0
`bool find( accessor& result, const Key& key ) const`	Ищет в таблице пару с указанным ключом. Если ключ найден, устанавливает `result`, так чтобы иметь доступ к соответствующей паре только для чтения. ПРЕДОСТЕРЕЖЕНИЕ: этот метод может сделать недействительными ранее полученные итераторы. В последовательном коде можно использовать метод `equal_range`, не вызывающий таких проблем. Возвращает `true`, если ключ найден, иначе `false`

`[A]`
`bool insert(`
`  accessor& result,`
`  const Key& key )`
`[B]`
`bool insert(`
`  [ accessor& result,`
`  const value_type& value )`
`[C]`
`bool insert( const value_type& value )`
	Ищет в таблице пару с указанным ключом.
	Если ключ не найден, то вставляет в таблицу
	[A] новую пару `pair(key,T())`
	или
	[B или C] новую пару, созданную копирующим конструктором из `value`.
	Устанавливает `result`, так чтобы иметь доступ к соответствующей паре только для чтения.
	Возвращает `true`, если новая пара вставлена, или `false`, если ключ уже имеется в таблице.
	Совет: если вам не нужен доступ к данным после вставки, то пользуйтесь формой `insert`, не принимающей акцессора; она работает быстрее и захватывает меньше блокировок

`template<typename InputIterator> void`
`insert( InputIterator first,`
`  InputIterator last )`
	Для каждой пары `p` в полуоткрытом интервале `[first, last)` выполняет `insert(p)`. Не определено, в каком порядке производятся вставки и производятся ли они конкурентно.
	ПРЕДОСТЕРЕЖЕНИЕ: в текущей реализации вставки производятся по порядку. В будущем они, возможно, будут производиться конкурентно. Если в `[first, last)` существуют повторяющиеся ключи, организуйте программу так, чтобы результат не зависел от порядка их вставки

`void insert(` `std::initializer_list<value_type> il )`	Вставляет в таблицу каждый элемент из списка инициализации. Не определено, в каком порядке производятся вставки и производятся ли они конкурентно. ПРЕДОСТЕРЕЖЕНИЕ: в текущей реализации вставки производятся по порядку. В будущем они, возможно, будут производиться конкурентно. Если в `[first, last)` существуют повторяющиеся ключи, организуйте программу так, чтобы результат не зависел от порядка их вставки
`bool erase( const Key& key )`	Ищет в таблице пару с указанным ключом. Удаляет найденную пару, если таковая существует. Если существует акцессор, указывающий на эту пару, то она все равно удаляется из таблицы, но ее уничтожение откладывается до тех пор, пока не останется ни одного указывающего на нее акцессора. Возвращает `true`, если пара была удалена, или `false`, если найти такую пару не удалось

Рис. B.51 (*продолжение*)

`bool` **`erase`**`(` *accessor*`& item_accessor )`	Требование: `item_accessor.empty()==false`. Результат: удаляется пара, на которую ссылается акцессор. Конкурентная вставка с таким же ключом создает в таблице новую пару, которая может временно сосуществовать с уничтожаемой. Если для доступа используется `const_accessor` и существует еще один `const_accessor`, указывающий на ту же пару, то она все равно удаляется из таблицы, но ее уничтожение откладывается до тех пор, пока не останется ни одного указывающего на нее акцессора. Возвращает `true`, если пара была удалена в текущем потоке, или `false`, если она была удалена в другом потоке
Конкурентное хеш-отображение...	
Параллельное итерирование	
Член	**Описание**
`const_range_type` **`range`**`( size_t grainsize=1) const`	Конструирует диапазон типа `const_range_type`, представляющий все ключи в таблице. Параметр `grainsize` измеряется в ячейках хеш-таблицы. В каждой ячейке в среднем находится приблизительно одна пара ключ/значение. Возвращает объект типа `const_range_type` для таблицы
`range_type` **`range`**`( size_t grainsize=1)`	Возвращает объект типа `range_type` для таблицы
Конкурентное хеш-отображение...	
Емкость	
Член	**Описание**
`size_type` **`size`**`() const`	Возвращает количество пар ключ/значение в таблице
`bool` **`empty`**`() const`	Возвращает `size()==0`. Примечание: этот метод работает постоянное время, но медленнее, чем для большинства STL-контейнеров
`size_type` **`max_size`**`() const`	Возвращает включительную верхнюю границу количества пар ключ/значение в таблице.
`size_type` **`bucket_count`**`() const`	Возвращает текущее количество ячеек. См. обсуждение ячеек в описании метода `rehash`
Конкурентное хеш-отображение...	
Итераторы	
Член	**Описание**
`[const_]iterator` **`begin()`**	Возвращает `[const_]iterator`, указывающий на начало последовательности пар ключ/значение
`[const_]iterator` **`end()`**	Возвращает `[const_]iterator`, указывающий на позицию за концом последовательности пар ключ/значение
`std::pair<iterator, iterator>` **`equal_range`**`( const Key& key )`	Возвращает пару итераторов `(i,j)` такую, что полуоткрытый диапазон `[i,j)` содержит все пары в таблице, для которых ключ равен `key` (и только такие пары). Поскольку в хеш-таблице не может быть повторяющихся ключей, этот полуоткрытый диапазон либо пуст, либо содержит только одну пару. Совет: эти методы являются последовательными альтернативами конкурентным методам `count` и `find`
`std::pair<const_iterator, const_iterator>)` **`equal_range`**`( const Key& key ) const`	

Рис. B.51 (*продолжение*)

Конкурентное хеш-отображение...	
Глобальные функции	
Член	**Описание**
`template<typename Key, typename T, typename HashCompare, typename A1, typename A2> bool operator=( const concurrent_hash_map<Key,T,HashCompare,A1>& a, const concurrent_hash_map<Key,T,HashCompare,A2>& b)`	Возвращает `true`, если `a` и `b` содержат одинаковые множества ключей и для каждой пары `(k,v1)∈a` и `(k,v2)∈b` выражение `bool(v1==v2)` равно `true`
`template<typename Key, typename T, typename HashCompare, typename A1, typename A2> bool operator!=( const concurrent_hash_map<Key,T,HashCompare,A1>& a, const concurrent_hash_map<Key,T,HashCompare,A2>& b)`	Возвращает `!(a==b)`
`template<typename Key, typename T, typename HashCompare, typename A> void swap( concurrent_hash_map<Key,T,HashCompare,A>& a, concurrent_hash_map<Key,T,HashCompare,A>& b)`	`a.swap(b)`

Рис. B.51 (*продолжение*)

Классы конкурентных неупорядоченных отображения, мультиотображения, множества и мультимножества

#include <**tbb**/concurrent_unordered_map.h>

#include <**tbb**/concurrent_unordered_set.h>

Эти классы поддерживают одновременную вставку и обход. Множества – это коллекции элементов (ключей), отображения – коллекции элементов (ключей), отображаемых на значения. В отображении и множестве ключи должны быть уникальны, мультиотображения и мультимножества допускают повторяющиеся ключи.

Синтаксис конструктора

```
concurrent_unordered_map<typename Key,
                         typename Element,
                         typename Hasher = tbb_hash<Key>,
                         typename Equality = std::equal_to<Key>,
                         typename Allocator =
                            tbb::tbb_allocator<std::pair<const Key, Element> >
concurrent_unordered_multimap< то же, что concurrent_unordered_map >

concurrent_unordered_set<typename Key,
                         typename Hasher = tbb_hash<Key>,
                         typename Equality = std::equal_to<Key>,
                         typename Allocator =
                            tbb::tbb_allocator<Key> >
concurrent_unordered_multiset< то же, что concurrent_unordered_set >
```

Рис. B.52 ❖ Контейнеры: классы конкурентных неупорядоченных отображения, мультиотображения, множества и мультимножества (*начало*)

Различия между неупорядоченными (мульти)множествами в STL и в TBB

- Некоторые методы, требующие средств C++11 (например, ссылок на `rvalue`), опущены.
- Методы стирания снабжены префиксом `unsafe`, чтобы показать, что они потоконебезопасны.
- Методы, работающие с ячейками, снабжены префиксом `unsafe`, чтобы показать, что они потоконебезопасны относительно вставки.
- Для класса `concurrent_unordered_set` методы вставки могут создавать временный элемент, который уничтожается, если другой поток конкурентно вставит такой же элемент.
- Как и в `std::list`, вставка новых элементов не делает итераторы недействительными и не изменяет порядка элементов, уже находящихся в отображении. Вставка и обход могут производиться одновременно.
- Типы итераторов `iterator` и `const_iterator` принадлежат к категории однонаправленных итераторов.
- Вставка не делает недействительными и не обновляет итераторы, возвращенные методом `equal_range`, поэтому в результате вставки в конце диапазона могут оказаться неравные элементы.

Конкурентные неупорядоченные множество и мультимножество...

Уничтожение и копирование

Член	Описание
`~concurrent_unordered_[multi]set``()`	Уничтожает множество
`concurrent_unordered_[multi]set& operator= (const concurrent_unordered_[multi]set& m);`	Присваивает `*this` содержимое `m`. Возвращает ссылку на `*this`
`concurrent_unordered_[multi]set& operator= (concurrent_unordered_[multi]set&& m);`	Перемещает данные из `m` в `*this`. Объект `m` остается в неопределенном состоянии, но может быть безопасно уничтожен. Возвращает ссылку на `*this`
`concurrent_unordered_[multi]set& operator= (std::initializer_list<value_type> il);`	Присваивает `*this` содержимое `il`. Возвращает ссылку на `*this`
`allocator_type` **`get_allocator()`** `const;`	Возвращает копию распределителя, ассоциированного с `*this`

Конкурентные неупорядоченные множество и мультимножество...

Размер и емкость

Член	Описание
`bool empty() const`	Возвращает `size()==0`
`size_type` **`size`**`() const`	Возвращает количество элементов в `*this`. ПРЕДОСТЕРЕЖЕНИЕ: хотя сложность текущей реализации $O(1)$, будущие реализации, возможно, будут иметь сложность $O(P)$, где P – количество аппаратных потоков
`size_type max_size() const`	Возвращает верхнюю границу элементов в `*this`. ПРЕДОСТЕРЕЖЕНИЕ: верхняя граница может быть гораздо больше количества элементов, реально помещающихся в контейнере

Конкурентные неупорядоченные множество и мультимножество...

Итераторы

Член	Описание
`[const_]iterator` **`begin()`**	Возвращает `[const_]iterator`, указывающий на первый элемент множества
`[const_]iterator` **`end()`**	Возвращает `[const_]iterator`, указывающий на позицию за последним элементом множества
`const_iterator cbegin()`	Возвращает `const_iterator`, указывающий на первый элемент множества
`const_iterator cend()`	Возвращает `const_iterator`, указывающий на позицию за последним элементом множества

Рис. B.52 (*продолжение*)

Конкурентные неупорядоченные множество и мультимножество...	
Модификаторы	
Член	**Описание**
`std::pair<iterator, bool> insert(const value_type& x);`	Конструирует копию `x` и пытается вставить ее в множество. Если попытка завершается неудачно, поскольку элемент с таким ключом уже существует, то копия уничтожается. Возвращает `std::pair(iterator, success)`, где `iterator` указывает на элемент множества с данным ключом, а `success` равно `true`, если элемент был вставлен, и `false` в противном случае

`iterator insert(`
`  const_iterator hint,`
`  const value_type& x);`
	То же, что `insert(k)`.
	Примечание: в текущей реализации аргумент `hint` игнорируется. Он присутствует, чтобы функция была похожа на классы `unordered_set` и `unordered_multiset` в C++11. Его цель – подсказать, откуда начинать поиск. Как правило, он должен указывать на элемент по соседству с местом, куда будет произведена вставка.
	Возвращает итератор, указывающий на вставленный элемент или на элемент с тем же ключом, который уже присутствовал в множестве

`std::pair<iterator, bool> insert(value_type&& x);`	Перемещает `x` в новый экземпляр `value_type` и пытается вставить его в множество. Если попытка завершается неудачно, поскольку элемент с таким ключом уже существует, то экземпляр уничтожается. Возвращает то же, что `insert(const value_type& x)`

`iterator insert(`
`  const_iterator hint,`
`  value_type&& x);`
	То же, что `insert(x)`.
	Примечание: в текущей реализации аргумент `hint` игнорируется. Он присутствует, чтобы функция была похожа на классы `unordered_set` и `unordered_multiset` в C++11. Его цель – подсказать, откуда начинать поиск. Как правило, он должен указывать на элемент по соседству с местом, куда будет произведена вставка.
	Возвращает то же, что `insert(const_iterator hint, const value_type& x)`

`template<class InputIterator>`
`  void insert(`
`  InputIterator first`
`  InputIterator last);`
	Выполняет `insert(*i)` для каждого `i` в полуоткрытом интервале `[first, last)`

`void insert(` `  std::initializer_list<value_type> il);`	Вставляет в множество все элементы из списка инициализации

`template<typename... Args>`
`  std::pair<iterator hint,`
`  emplace(Args&&... args);`
	Конструирует новый экземпляр `value_type` из `args` и пытается вставить его в множество. Если попытка завершается неудачно, поскольку элемент с таким ключом уже существует, то экземпляр уничтожается.
	Возвращает то же, что `insert(const value_type& x)`

`template<typename... Args>`
`  iterator`
`  emplace_hint(`
`  const_iterator hint,`
`  Args&&... args);`
	То же, что `emplace(args)`.
	Примечание: в текущей реализации аргумент `hint` игнорируется. Он присутствует, чтобы функция была похожа на классы `unordered_set` и `unordered_multiset` в C++11. Его цель – подсказать, откуда начинать поиск. Как правило, он должен указывать на элемент по соседству с местом, куда будет произведена вставка.
	Возвращает итератор, указывающий на вставленный элемент или на элемент с тем же ключом, который уже присутствовал в множестве

Рис. В.52 (*продолжение*)

`iterator` **`unsafe_erase`**`(` `  const_iterator position);`	Удаляет из множества элемент, на который указывает `position`. Возвращает итератор, указывающий на элемент, следующий непосредственно за удаленным, или `end()`, если был удален последний элемент множества
`size_type` **`unsafe_erase`**`(` `  const key_type& k);`	Удаляет элемент с ключом `k`, если таковой существует. Возвращает 1, если элемент был удален, иначе 0

`iterator` **`unsafe_erase`**`(`
`  const_iterator first,`
`  const_iterator last);`
	Удаляет все элементы `*i`, где `i` принадлежит полуоткрытому интервалу `[first, last)`.
	Возвращает `last`

`void` **`clear`**`()`	Удаляет все элементы из множества
`void` **`swap`**`(` `  concurrent_unordered_[multi]set& m)`	Обменивает содержимое `*this` и `m`
Конкурентные неупорядоченные множество и мультимножество...	
Наблюдатели	
Член	**Описание**
`hasher hash_function() const`	Возвращает функтор хеширования, ассоциированный с множеством
`key_equal key_eq() const`	Возвращает функтор эквивалентности ключей, ассоциированный с множеством
Конкурентные неупорядоченные множество и мультимножество...	
Поиск	
Член	**Описание**
`iterator` **`find`**`(const key_type& k)`	Возвращает итератор, указывающий на элемент с ключом, эквивалентным `k`, или `end()`, если такого элемента не существует
`const_iterator` **`find`**`(const key_type& k)` `const`	Возвращает `const_iterator`, указывающий на элемент с ключом, эквивалентным `k`, или `end()`, если такого элемента не существует
`size_type` **`count`**`(const key_type& k)` `const`	Возвращает количество элементов с ключом, эквивалентным `k`
`std::pair<iterator,iterator>` **`equal_range`**`(const key_type& k)`	Возвращает диапазон множества, содержащий все ключи, эквивалентные `k`
`std::pair<const_iterator,const_iterator>` **`equal_range`**`(const key_type& k) const`	Возвращает диапазон множества, содержащий все ключи, эквивалентные `k`
Конкурентные неупорядоченные множество и мультимножество...	
Параллельное итерирование	
Член	**Описание**
`const_range_type` **`range`**`() const`	Возвращает объект типа `const_range_type`, представляющий все ключи в множестве
`range_type` **`range`**`()`	Возвращает объект типа `range_type`, представляющий все ключи в множестве

Рис. B.52 (*продолжение*)

Конкурентные неупорядоченные множество и мультимножество...	
Интерфейс для работы с ячейками	
Член	**Описание**
`size_type` **`unsafe_bucket_count`**`() const`	Возвращает количество ячеек
`size_type` **`unsafe_max_bucket_count`**`() const`	Возвращает верхнюю границу возможного количества ячеек
`size_type` **`unsafe_bucket_size`**`(size_type n)`	Возвращает количество элементов в ячейке `n`
`size_type` **`unsafe_bucket`**`(const key_type& k) const`	Возвращает индекс ячейки, в которую был бы помещен ключ `k`
`local_iterator` **`unsafe_begin`**`(size_type n)`	Возвращает `local_iterator`, указывающий на первый элемент в ячейке `n`
`const_local_iterator` **`unsafe_begin`**`(size_type n) const`	Возвращает `const_local_iterator`, указывающий на первый элемент в ячейке `n`
`local_iterator unsafe_end(size_type n)`	Возвращает `local_iterator`, указывающий на позицию за последним элементом в ячейке `n`
`const_local_iterator unsafe_end(size_type n)`	Возвращает `const_local_iterator`, указывающий на позицию за последним элементом в ячейке `n`
`const_local_iterator` **`unsafe_cbegin`**`(size_type n) const`	Возвращает `const_local_iterator`, указывающий на первый элемент в ячейке `n`
`const_local_iterator unsafe_cend(size_type n)`	Возвращает `const_local_iterator`, указывающий на позицию за последним элементом в ячейке `n`
Конкурентные неупорядоченные множество и мультимножество...	
Политика хеширования	
Член	**Описание**
float **load_factor**() const	Возвращает количество элементов в ячейке
float **max_load_factor**() const	Возвращает максимальный размер ячейки. Если в результате вставки элемента образуется большая ячейка, то реализация может перераспределить элементы между ячейками или увеличить количество ячеек
void **max_load_factor**(float z)	Устанавливает максимальный размер ячейки равным `z`
void **rehash**(size_type n)	Ничего не делает, если текущее количество ячеек не меньше `n`. В противном случае увеличивает количество ячеек до `n`. *Примечание*: `n` должно быть степенью 2

Рис. В.52 (*окончание*)

Требования к концепции *диапазона контейнера*

Классы `concurrent_hash` и `concurrent_vector` должны иметь типы-члены `range_type` и `const_range_type`, моделирующие концепцию диапазона контейнера (Container Range).
Эти типы следует использовать в сочетании с алгоритмами `parallel_for`, `parallel_reduce` и `parallel_scan` для обхода элементов контейнера.
Класс `R`, реализующий концепцию диапазона контейнера, должен содержать следующие функции-члены и типы:

Из концепции диапазона	`R::R( const R& );`
	Копирующий конструктор
	`R::~R();`
	Деструктор
	`bool R::is_divisible() const;`
	`true`, если диапазон можно разбить на два поддиапазона
	`bool R::empty() const;`
	`true`, если диапазон пуст
	`R::R( R& r&, split );`
	Расщепляет диапазон на два поддиапазона
	`R::R( R& r&, proportional_split proportion );`
	Факультативно. Пропорционально расщепляющий конструктор. Расщепляет диапазон на два поддиапазона в пропорции `proportion`
	`static const bool R::is_splittable_in_proportion;`
	Факультативно. Если `true`, то пропорционально расщепляющий конструктор определен для данного диапазона и может использоваться в параллельных алгоритмах

Дополнительные требования для диапазона контейнера	`R::value_type`
	Тип элемента
	`R::reference`
	Тип ссылки на элемент
	`R::const_reference`
	Тип константной ссылки на элемент
	`R::difference_type`
	Тип разности двух итераторов
	`R::iterator`
	Тип итератора диапазона
	`R::iterator R::begin()`
	Первый элемент диапазона
	`R::iterator R::end()`
	Позиция за последним элементом диапазона
	`R::size_type R::grainsize() const`
	Степень детализации

Рис. В.53 ❖ Контейнеры: требования к концепции диапазона контейнера

Класс конкурентной очереди

#include <**tbb**/concurrent_queue.h>

Класс tbb::concurrent_queue – структура очереди, обслуживаемой в порядке «первым пришел – первым ушел», которая позволяет нескольким потокам конкурентно помещать и извлекать элементы. Емкость очереди ограничена только объемом памяти на целевой машине. Интерфейс аналогичен std::queue с отличиями, необходимыми для безопасности конкурентных обновлений.

Примеры кода приведены в главе 6 (рис. 6.7–6.9).

Конкурентные неупорядоченные множество и мультимножество...

Свойство	std::queue	tbb::concurrent_queue
Доступ к началу и к концу очереди	Методы front и back	Отсутствуют. При конкурентном выполнении операции могут быть небезопасны
Требуется наличие копирующего конструктора в типе T	Нет	Да
unsafe_size()	Возвращает количество элементов в очереди	Возвращает количество элементов в очереди. Может возвращать неправильное значение, если несколько операций push или try_pop выполняются одновременно

Копирование и извлечение элемента, если очередь q не пуста	bool b=!q.empty();
	if (b) {
	x=q.front();
	q.pop();
	}
		bool b = q.try_pop(x);

Член	Описание
concurrent_queue([const allocator_type& a])	Конструирует пустую очередь
concurrent_queue(const concurrent_queue& src [, const allocator_type& a])	Конструирует копию src
template<typename InputIterator> concurrent_queue(InputIterator first, InputIterator last [, const allocator_type& a])	Конструирует очередь, содержащую копии элементов из полуоткрытого интервала [first, last)
concurrent_queue(concurrent_queue&& src [const allocator_type& a])	Конструирует новую очередь, перемещая содержимое src с применением факультативно заданного распределителя. Очередь src остается в неопределенном состоянии, но может быть безопасно уничтожена
~concurrent_queue()	Уничтожает все элементы в очереди и сам контейнер, так что больше его использовать нельзя
void push(const T& source)	Помещает копию source в конец очереди
void push(T&& elem)	Помещает заданный элемент в очередь, вызывая его перемещающий конструктор (если таковой имеется)
template<typename... Arguments> void emplace(Arguments&&... args)	Помещает новый элемент в очередь. Элемент конструируется из заданных аргументов
bool try_pop(T& destination)	Если очередь не пуста, извлекает из нее элемент, присваивает его переменной destination и уничтожает. В противном случае не делает ничего. Возвращает true, если значение было извлечено, иначе false

Рис. В.54 ❖ Контейнеры: класс конкурентной очереди (*начало*)

`void clear()`	Очищает очередь. По завершении `size() == 0`. Поддерживается *только* для `concurrent_queue`
`size_type unsafe_size() const`	Возвращает количество элементов в очереди. Если одновременно выполняются модификации, то значение может не отражать истинное положение вещей. Поддерживается *только* для `concurrent_queue`
`bool empty() const`	Возвращает `true`, если в очереди нет элементов, иначе `false`
`allocator_type get_allocator() const`	Возвращает копию распределителя, заданного при конструировании очереди

Рис. B.54 (*окончание*)

Класс *ограниченной* конкурентной очереди

`#include <tbb/concurrent_queue.h>`

Класс ограниченной очереди `tbb::concurrent_bounded_queue` похож на `tbb::concurrent_queue`, но добавлена возможность задать емкость и, соответственно, модифицированы некоторые операции (`push` может ждать, добавлена операция `pop` с ожиданием, изменен `size()`, и добавлена операция отмены ожидающих `push` и `pop`).

Примеры кода приведены в главе 6 (рис. 6.7–6.9).

Ниже показаны только методы, отличающиеся от `tbb::concurrent_queue`

Описание конструкторов, деструктора (помимо смены имени) и неизменившихся членов (`try_pop`, `clear`, `get_allocator`) см. выше.

Член	Описание
`void push(const T& source)`	*Поведение изменилось из-за появления емкости.* Ждет, пока `size()<capacity`, затем помещает копию `source` в конец очереди
`void push(T&& source)`	*Поведение изменилось из-за появления емкости.* Ждет, пока `size()<capacity`, затем перемещает заданный элемент `source` в конец очереди
`template<typename... Arguments> void emplace(Arguments&&... args)`	*Поведение изменилось из-за появления емкости.* Ждет, пока `size()<capacity`, затем помещает новый элемент в очередь. Элемент конструируется из заданных аргументов
`void pop( T& destination )`	*Помимо метода `try_pop`, добавлен метод `pop()` с ожиданием.* Ждет, когда значение станет доступно, затем извлекает его из очереди и присваивает переменной `destination`. Исходное значение уничтожается
`void abort()`	*Дополнительный метод.* Пробуждает все потоки, ожидающие завершения операций `push` и `pop`, и возбуждает в этих потоках исключение `abort`. Эта возможность недоступна, если не установлена константа препроцессора `TBB_USE_EXCEPTIONS`
`bool try_push( [ const T& source ] T&& source ] )`	*Дополнительный метод.* Если `size()<capacity`, то помещает копию `source` в конец очереди. Возвращает `true`, если копия была помещена, иначе `false`
`template<typename... Arguments> bool try_emplace(Arguments&&... args)`	*Поведение изменилось из-за появления емкости.* Если `size()<capacity`, то конструирует элемент из заданных аргументов и перемещает его в конец очереди. Возвращает `true`, если элемент был перемещен, иначе `false`
`size_type size() const`	*Поведение изменилось из-за появления емкости.* Возвращает количество операций `push` минус количество операций `pop`. Результат отрицателен, если существуют операции `pop`, ожидающие соответствующих `push`. Результат может превышать емкость `capacity()`, если очередь полна и существуют операции `push`, ожидающие соответствующих `pop`.

Рис. B.55 ❖ Контейнеры: класс ограниченной конкурентной очереди (*начало*)

`bool empty() const`	*Поведение изменилось из-за появления емкости.* Возвращает `size()<=0`
`size_type capacity() const`	*Дополнительный метод.* Возвращает максимальное количество значений в очереди
`void set_capacity( size_type capacity)`	*Дополнительный метод.* Устанавливает максимальное количество значений в очереди

Рис. B.55 (*окончание*)

Класс конкурентной очереди *с приоритетами*

`#include <tbb/concurrent_priority_queue.h>`

Класс `tbb::concurrent_priority_queue` представляет контейнер, в который могут добавлять или извлекать элементы несколько потоков одновременно. Элементы извлекаются в порядке приоритета, определенного параметром шаблона. Емкость очереди ограничена только объемом памяти на целевой машине. Интерфейс аналогичен `std::priority_queue` с отличиями, необходимыми для безопасности конкурентных обновлений.

Примеры кода приведены в главе 6 (рис. 6.7–6.9).

Различия между очередями в STL и TBB

Свойство	**`std::priority_queue`**	**`tbb::concurrent_priority_queue`**
Выбор базового контейнера	Параметр шаблона	Базовый контейнер не задается, вместо него задается распределитель
Доступ к элементу с наивысшим приоритетом	`const value_type& top() const`	Отсутствует. Для конкурентного контейнера небезопасно

Копирование и извлечение	`bool b = !q.empty();`
	`if (b) {`
	`  x = q.top();`
	`  q.pop();`
	`}`
		`bool b = q.try_pop(x);`

Получение количества элементов в очереди	`size_type size() const`	Так же, но результат может быть неточным из-за ожидающих операций `push` или `pop`
Проверка наличия элементов в очереди	`bool empty() const`	Так же, но результат может быть неточным из-за ожидающих операций `push` или `pop`

Конкурентная очередь с приоритетами

Член	**Описание**
`concurrent_priority_queue( [ size_type init_capacity ] [ запятая, если необходимо ] [ const allocator_type& a ] )`	Конструирует пустую очередь. Факультативно может быть задана начальная емкость. *Запятая необходима, если заданы оба необязательных параметра*
`concurrent_priority_queue( const concurrent_priority_queue& src [ запятая, если необходимо ] [ , const allocator_type& a ] )`	Конструирует копию `src`
`template<typename InputIterator> concurrent_priority_queue( InputIterator first, InputIterator last [ , const allocator_type& a ] )`	Конструирует очередь, содержащую копии всех элементов из полуоткрытого интервала `[first, last)`
`concurrent_priority_queue( std::initializer_list<T> il [ , const allocator_type& a ] )`	Эквивалентно `concurrent_priority_queue( il.begin(), il.end(), a)`

Рис. B.56 ❖ Контейнеры: класс конкурентной очереди с приоритетами (*начало*)

`concurrent_priority_queue( concurrent_priority_queue&& src [ , const allocator_type& a ] )`	Конструирует новую очередь, перемещая содержимое `src` с применением факультативно заданного распределителя. Очередь `src` остается в неопределенном состоянии, но может быть безопасно уничтожена
`~concurrent_priority_queue()`	Уничтожает все элементы в очереди и сам контейнер, так что больше его использовать нельзя
`concurrent_priority_queue& operator= (const concurrent_priority_queue& src)`	Присваивает `*this` содержимое `src`. Эта операция потоконебезопасна и может привести к ошибке или некорректной копии `src`, если другой поток одновременно модифицирует `src`. Возвращает ссылку на `*this`
`concurrent_priority_queue& operator= (concurrent_priority_queue&& src)`	Перемещает в `*this` данные из `src`. Очередь `src` остается в неопределенном состоянии, но может быть безопасно уничтожена. Эта операция небезопасна, если существуют ожидающие конкурентные операции с очередью `src`. Возвращает ссылку на `*this`
`concurrent_priority_queue& operator= (std::initializer_list<T> il)`	Присваивает `*this` содержимое списка инициализации `il`. Возвращает ссылку на `*this`
`template<typename InputIterator> void assign(InputIterator begin, InputIterator end, const allocator_type&] )`	Присваивает `*this` содержимое полуоткрытого интервала `[begin, end)`
`void assign( std::initializer_list<T> il)`	Эквивалентно `assign(il.begin(), il.end())`
`bool empty() const`	Возвращает `true`, если в очереди нет элементов, иначе `false`. Результат может быть неверен, если существуют ожидающие операции `push` или `try_pop`. Операция читает разделяемые данные, и при конкурентном использовании инструменты обнаружения гонок могут идентифицировать ее как гонку
`size_type size() const`	Возвращает количество элементов в очереди. Результат может быть неверен, если существуют ожидающие операции `push` или `try_pop`. Операция читает разделяемые данные, и при конкурентном использовании инструменты обнаружения гонок могут идентифицировать ее как гонку
`void push(const_reference elem)`	Помещает копию `elem` в очередь. Операция может безопасно выполняться одновременно с `push`, `try_pop` и `emplace`
`void push(T&& elem)`	Помещает заданный элемент в очередь, применяя перемещающий конструктор. Операция может безопасно выполняться одновременно с `push`, `try_pop` и `emplace`
`template<typename... Args> void emplace(Args&&... args)`	Помещает новый элемент в очередь. Элемент конструируется из заданных аргументов. Операция может безопасно выполняться одновременно с `push`, `try_pop` и `emplace`
`bool try_pop(reference elem)`	Если очередь не пуста, извлекает из нее элемент с наивысшим приоритетом, присваивает его переменной `elem` и уничтожает. В противном случае не делает ничего. Операция может безопасно выполняться одновременно с `push`, `try_pop` и `emplace`. Возвращает `true`, если элемент был извлечен, иначе `false`
`void clear()`	Очищает очередь. По завершении `size() == 0`. Операция потоконебезопасна
`void swap(concurrent_priority_queue& other)`	Обменивает содержимое двух очередей. Операция потоконебезопасна
`allocator_type get_allocator() const`	Возвращает копию распределителя, заданного при конструировании очереди

Рис. B.56 (*продолжение*)

Класс конкурентного вектора

#include <**tbb**/concurrent_vector.h>

Класс tbb::concurrent_vector<T, [A]> представляет динамически растущий массив элементов типа T с распределителем A (по умолчанию подразумевается распределитель tbb::cache_aligned_allocator<T>). Увеличение размера concurrent_vector безопасно в присутствии других потоков, выполняющих операции над его элементами, и даже когда эти операции сами приводят к увеличению размера. Для обеспечения безопасного роста concurrent_vector располагает тремя методами: push_back, grow_by и grow_to_at_least.

Класс concurrent_vector удовлетворяет всем требованиям концепций контейнера и обратимого контейнера, перечисленным в стандарте ISO C++. Он не удовлетворяет требованиям концепции последовательности из-за отсутствия методов insert() и erase().

В отличие от std::vector, класс concurrent_vector никогда не перемещает существующих элементов при увеличении размера. Этот контейнер выделяет память для ряда непрерывных массивов. При первом резервировании места, росте или операции присваивания определяется размер первого массива. Если начальный размер выбран слишком малым, то элементы массива будут копироваться в разные строки кеша, и возникающая фрагментация увеличивает время доступа к элементу. Метод shrink_to_fit() объединяет несколько массивов в один большой непрерывный массив, что может ускорить доступ.

Примеры кода приведены в главе 6.

Класс конкурентного вектора...

Методы конструирования

Эти операции нельзя конкурентно вызывать для одного и того же вектора

Член	Описание

concurrent_vector(
  [size_type n
  [, const_reference t=T()]]
  [запятая при необходимости]
  [const allocator_type& a]
)
	Конструирует вектор, содержащий n копий t, применяя факультативно заданный распределитель.
	Если n не задано, то создается пустой вектор.
	Если t не задано, то каждый элемент создается конструктором по умолчанию, а не копирующим конструктором

template<typename InputIterator>
concurrent_vector(
  InputIterator first,
  InputIterator last
  [, const allocator_type& a]
)
	Конструирует вектор, являющийся копией последовательности [first, last), выполняя только N обращений к копирующему конструктору типа T, где N – расстояние между first и last

concurrent_vector(
  std::initializer_list<T> il
  [, const allocator_type& a])
	Эквивалентно concurrent_vector(il.begin(), il.end(), a)

concurrent_vector(concurrent_vector& src)	Конструирует копию src

concurrent_vector(
  concurrent_vector&& src
  [, const allocator_type& a]
)
	Конструирует новый вектор, перемещая в него содержимое src, с применением факультативно заданного распределителя a. Вектор src остается в неопределенном состоянии, но может быть безопасно уничтожен

concurrent_vector& operator= (const concurrent_vector&)	Присваивает *this содержимое src. Возвращает ссылку на *this
concurrent_vector& operator= (concurrent_vector&&)	Перемещает в *this данные из src. Вектор src остается в неопределенном состоянии, но может быть безопасно уничтожен. Возвращает ссылку на *this
template<typename M> concurrent_vector& operator= (const concurrent_vector<T,M>& src)	Присваивает *this содержимое src. Возвращает ссылку на *this
concurrent_vector& operator= (std::initializer_list<T> il)	Присваивает *this содержимое списка инициализации il. Возвращает ссылку на *this

Рис. В.57 ❖ Контейнеры: класс конкурентного вектора

Класс конкурентного вектора...	
Методы, применяемые к вектору в целом: копирование, присваивание, уничтожение *Конкурентное применение этих операций к одному и тому же вектору небезопасно*	
Член	**Описание**
`void assign( size_type n, const_reference t )`	Присваивает `*this` `n` копий `t`
`template<class InputIterator> void assign(InputIterator first, InputIterator last)`	Присваивает содержимое последовательности `[first, last)`, выполняя только `N` обращений к копирующему конструктору типа `T`, где `N` – расстояние между `first` и `last`
`void assign( std::initializer_list<T> il )`	Эквивалентно `assign(il.begin(), il.end(), a)`
`void reserve( size_type n )`	Резервирует место как минимум для `n` элементов. Возбуждает исключение `std::length_error`, если `n>max_size()`. Исключение возбуждается также в случае, когда распределитель возбуждает исключение. Если возбуждено исключение, вектор остается в корректном состоянии
`void shrink_to_fit()`	Уплотняет внутреннее представление, чтобы избежать фрагментации
`void swap(concurrent_vector& x )`	Обменивает содержимое двух векторов за время $O(1)$
`void clear()`	Стирает все элементы. После этого `size()==0`. Не освобождает память, занятую внутренними массивами. Совет: для освобождения внутренних массивов следует вызвать `shrink_to_fit()` после `clear()`
`~concurrent_vector()`	Стирает все элементы и уничтожает сам вектор
Класс конкурентного вектора...	
Методы увеличения размера *Описанные в этом разделе методы можно конкурентно применять к одному и тому же вектору*	
Член	**Описание**

`iterator grow_by( size_type delta )`
 	
`iterator grow_by( size_type delta, const_reference_t )`
	Добавляет последовательность, содержащую `delta` новых элементов, в конец вектора. Если задан второй параметр `t`, то новые элементы инициализируются копированием `t`, в противном случае они создаются конструктором по умолчанию. Возвращает итератор, указывающий на начало добавленной последовательности

`template<typename ForwardIterator> iterator grow_by( ForwardIterator first, ForwardIterator last )`	Добавляет последовательность в конец вектора, копируя каждый элемент в диапазоне `[first, last)`. Возвращает итератор, указывающий на начало добавленной последовательности
`iterator grow_by( std::initializer_list<T> il )`	Добавляет последовательность в конец вектора, копируя каждый элемент из списка инициализации. Возвращает итератор, указывающий на начало добавленной последовательности

Рис. В.57 (*продолжение*)

`iterator grow_to_at_least( size_type n )`
 	
`iterator grow_to_at_least( size_type n, const_reference t )`
	Добавляет минимальную последовательность элементов такую, что `vector.size()>=n`. Если задан второй параметр `t`, то новые элементы инициализируются копированием `t`, в противном случае они создаются конструктором по умолчанию. Блокирует остальные потоки, пока не будет выделена память для всех элементов в диапазоне `[0..n)` (но необязательно дожидается завершения их конструирования, поскольку они могут находиться в процессе конструирования в другом потоке).
	Совет: если поток должен знать, завершилось ли конструирование элемента, то можно применить следующую технику. Создать экземпляр `concurrent_vector`, используя распределитель `zero_allocator`. Определить конструктор `T()`, так чтобы по завершении он присваивал некоторому полю `T` ненулевое значение. Поток может узнать, сконструирован ли элемент вектора `concurrent_vector`, сравнив это значение с нулем.
	Возвращает итератор, указывающий на начало добавленной последовательности, или указатель на `(*this)[n]`, если не было добавлено ни одного элемента

`iterator push_back( const_reference value )`	Добавляет копию `value` в конец вектора. Возвращает итератор, указывающий на эту копию
`iterator push_back( T&& value )`	Перемещает `value` в новый элемент, добавленный в конец вектора. Возвращает итератор, указывающий на новый элемент
`template<typename... Args> emplace_back( Args&&... args )`	Добавляет новый элемент в конец вектора. Элемент конструируется из заданных аргументов. Возвращает итератор, указывающий на новый элемент
Класс конкурентного вектора...	
Методы доступа *Описанные в этом разделе методы можно конкурентно применять к одному и тому же вектору. Однако возвращенная ссылка может указывать на элемент, который еще находится в процессе конкурентного конструирования.*	
Член (необязательный квалификатор `const` ниже встречается либо два раза, либо ни одного)	**Описание**
`[const_]reference operator[] ( size_type index ) [const]`	Возвращает [константную] ссылку на элемент с указанным индексом
`[const_]reference at( size_type index ) [const]`	Возвращает [константную] ссылку на элемент с указанным индексом. Возбуждает исключение `std::out_of_range`, если `index >= size()` или `index` относится к поврежденной части вектора
`[const_]reference front() [const]`	Возвращает `(*this)[0]`
`[const_]reference back() [const]`	Возвращает `(*this)[size()-1]`
Класс конкурентного вектора...	
Методы параллельного итерирования *Типы `const_range_type` и `range_type` моделируют концепцию диапазона контейнера (Container Range). Они отличаются только тем, что границы `const_range_type` имеют тип `const_iterator`, а границы `range_type` – тип `iterator`.*	
Член	**Описание**
`[const_]range_type range[] ( size_t grainsize=1 ) [const]`	Возвращает диапазон, охватывающий весь `concurrent_vector`, допускающий доступ только для чтения (при наличии `const`) или для чтения и записи

Рис. B.57 (*продолжение*)

Класс конкурентного вектора...	
Методы, относящиеся к емкости	
Член	**Описание**
`size_type size() const`	Возвращает количество элементов в векторе. Результат может включать элементы, память для которых уже выделена, но которые еще находятся в процессе конструирования вследствие конкурентного вызова одного из методов увеличения размера
`bool empty() const`	Возвращает `size() == 0`
`size_type capacity() const`	Возвращает максимальный размер, до которого может вырасти вектор без выделения дополнительной памяти. *Примечание*: в отличие от `std::vector`, `concurrent_vector` не перемещает существующие элементы при выделении дополнительной памяти
`size_type max_size() const`	Возвращает наибольший размер, до которого может вырасти вектор
Класс конкурентного вектора...	
Методы итераторов Шаблонный класс `concurrent_vector<T>` поддерживает итераторы произвольного доступа, описанные в стандарте ISO C++. В отличие от `std::vector`, итераторы не являются простыми указателями. Класс `concurrent_vector<T>` удовлетворяет требованиям к обратимому контейнеру из стандарта ISO C++.	
Член (необязательный квалификатор `const` ниже встречается либо два раза, либо ни одного)	**Описание**
`[const_]iterator begin() const`	Возвращает [константный] итератор, указывающий на начало вектора
`[const_]iterator end() const`	Возвращает [константный] итератор, указывающий на позицию за концом вектора
`[const_]reverse_iterator rbegin() const`	Возвращает [константный] итератор, указывающий на начало обращенного вектора
`[const_]reverse_iterator rend() const`	Возвращает [константный] итератор, указывающий на позицию за концом обращенного вектора

Рис. B.57 (*окончание*)

Синхронизация

В главе 5 полностью рассмотрена синхронизация в TBB; в этом разделе приводится краткий обзор поддерживаемых API с примечаниями. Далее будет дан обзор поточно-локальной памяти, которая также была рассмотрена в главе 5. TBB предоставляет платформенно-независимое взаимное исключение (мьютексы) и атомарные операции. Они появились в TBB раньше, чем были добавлены в стандарт C++. После включения аналогичных возможностей в стандарт

TBB стала поддерживать интерфейсы C++11 (определенные в пространстве имен `std`, а не `tbb`) для условных переменных и блокировки, ограниченной областью видимости. Однако имеется несколько отличий:

- поддержка со стороны TBB обеспечивается вне зависимости от того, имеется ли в конкретной системе полная поддержка C++11;
- в реализации используется интерфейс `tbb::tick_count` вместо интерфейса C++11 `<chrono>`;
- реализация возбуждает исключение `std::runtime_error`, если класс C++11 `std::system_error` недоступен;
- реализация опускает или аппроксимирует такие требующие наличия языковой поддержки со стороны C++11 средства, как `constexpr` и явные операторы;
- реализация работает с классом `tbb::mutex` в тех случаях, когда спецификация C++11 требует `std::mutex`;
- функция `notify_all_at_thread_exit()` не поддерживается.

Мьютексы							
имя класса	**#include<tbb/?>**	Масштабируемый	Справедливый	Реентерабельный	Долгое ожидание	Использует TSX	Размер
mutex	mutex.h	Зависит от ОС		Нет	Блокирует	Нет	>=3 слов
recursive_mutex	recursive_mutex.h			Да		Нет	>=3 слов
spin_mutex	spin_mutex.h	Нет	Нет	Нет	Уступает	Нет	1 байт
speculative_spin_mutex		Зависит от оборудования				Да	2 строки кеша
queuing_mutex	queueing_mutex.h	Да	Да	Нет		Нет	1 слово
spin_rw_mutex	spin_rw_mutex.h	Нет	Нет	Нет		Нет	1 слово
speculative_spin_rw_mutex		Зависит от оборудования				да	3 строки кеша
queueing_rw_mutex	queueing_rw_mutex.h	Да	Да	Нет		Нет	1 слово
null_mutex	null_mutex.h	–	Да	Да	Никогда	Нет	Пустой
null_rw_mutex	null_rw_mutex.h						

Рис. B.58 ❖ Синхронизация: сравнение различных мьютексов

Совместимость с C++11

Подробное описание членов стандартных мьютексов см. в тексте стандарта C++11. Классы `mutex`, `recursive_mutex`, `spin_mutex` и `spin_rw_mutex` поддерживают описанные ниже интерфейсы C++11.

Псевдосигнатура	Семантика
`void M::lock()`	Захватить блокировку
`bool M::try_lock()`	Попытаться захватить блокировку. Возвращает `true`, если блокировка захвачена, иначе `false`
`void M::unlock()`	Освободить блокировку
`class lock_guard<M>` `class unique_lock<M>`	См. полную документацию в стандарте C++. Класс `std::lock_guard` удерживает ассоциированный с ним мьютекс захваченным в течение всего времени своего существования. Он захватывает блокировку в конструкторе и освобождает ее в деструкторе. Класс `std::unique_lock` обладает более гибкими возможностями захвата мьютекса. Он предоставляет такой же интерфейс, как `std::lock_guard`, но имеет дополнительные методы для явного захвата и освобождения мьютексов и откладывания блокировки при конструировании
Классы `mutex` и `recursive_mutex` поддерживают также имеющуюся в C++11 идиому доступа к базовым описателям на уровне ОС, описанную ниже (здесь `M` – `mutex` или `recursive_mutex`)	
`M::native_handle_type`	Платформенный тип описателя. В Windows это тип `LPCRITICAL_SECTION`, в остальных операционных системах – `pthread_mutex`
`native_handle_type M::native_handle()`	Получить платформенный описатель мьютекса `M`

Рис. B.59 ❖ Синхронизация: поддержка мьютексов C++11

Мьютексы

#include <*см. ниже***>**

Мьютексы в TBB обеспечивают взаимное исключение потоков в областях кода, обертывая соответствующие вызовы ОС. TBB-интерфейсы построены на основе паттерна блокировки, ограниченной областью видимости, который широко используется в C++, т. к., с одной стороны, не требует явного освобождения блокировки, а с другой – автоматически освобождает блокировку, если произошло исключение, выводящее за пределы области, защищенной этой блокировкой. В интерфейсах, в название которых входит слово `speculative`, используется *гипотетическая* блокировка (TSX в процессорах Intel), они поддерживаются только в системах, где эта возможность реализована аппаратно. Гипотетическая блокировка позволяет нескольким потокам захватить один и тот же мьютекс при условии, что не будет конфликтов, способных дать результаты, отличающиеся от тех, что были бы получены в случае негипотетической блокировки. Эти мьютексы масштабируемы при работе в условиях низкой частоты конфликтов, т. е. преимущественно в режиме гипотетической блокировки.

Рис. B.60 ❖ Синхронизация: примеры мьютексов (*начало*)

```
#include <cstdio>
#include <tbb/tick_count.h>
#include <tbb/mutex.h>
#include <tbb/recursive_mutex.h>
#include <tbb/spin_mutex.h>
#include <tbb/queuing_mutex.h>
#include <tbb/spin_rw_mutex.h>
#include <tbb/queuing_rw_mutex.h>
#include <tbb/null_mutex.h>
#include <tbb/null_rw_mutex.h>
int main( int argc, char *argv[] ) {
  tbb::mutex                      my_mutex_01;
  tbb::recursive_mutex            my_mutex_02;
  tbb::spin_mutex                 my_mutex_03;
  tbb::speculative_spin_mutex     my_mutex_04;
  tbb::queuing_mutex              my_mutex_05;
  tbb::spin_rw_mutex              my_mutex_06;
  tbb::speculative_spin_rw_mutex  my_mutex_07;
  tbb::queuing_rw_mutex           my_mutex_08;
  tbb::null_mutex                 my_mutex_09;
  tbb::null_rw_mutex              my_mutex_10;
  {
   tbb::mutex::scoped_lock                    mylock01a(my_mutex_01);
   // следующая строка привела бы к зависанию, потому что
   // этот мьютекс уже захвачен
   // tbb::mutex::scoped_lock                 mylock01b(my_mutex_01);
   tbb::recursive_mutex::scoped_lock          mylock02a(my_mutex_02);
   // рекурсия разрешена, поэтому следующая строка не приводит
   // к зависанию
   tbb::recursive_mutex::scoped_lock          mylock02b(my_mutex_02);
   // рекурсия разрешена, поэтому следующая строка не приводит
   // к зависанию
   tbb::recursive_mutex::scoped_lock          mylock02c(my_mutex_02);
   tbb::spin_mutex::scoped_lock               mylock03a(my_mutex_03);
   tbb::speculative_spin_mutex::scoped_lock   mylock04a(my_mutex_04);
   // читатель...
   tbb::spin_rw_mutex::scoped_lock            mylock06a(my_mutex_06);
   // следующая строка привела бы к зависанию!
   // мы уже захватили мьютекс в этом потоке
   // RW разрешает захватывать мьютекс только один раз в потоке -
   // это не рекурсивная блокировка!
   // tbb::spin_rw_mutex::scoped_lock         mylock06c(my_mutex_06);
   tbb::speculative_spin_rw_mutex::scoped_lock mylock07a(my_mutex_07);
   // писатель...
   tbb::queuing_rw_mutex::scoped_lock        mylock08a(my_mutex_08,true);
   tbb::null_mutex::scoped_lock              mylock09a(my_mutex_09);
   // null ничего не делает... это не приводит к зависанию...
   tbb::null_mutex::scoped_lock              mylock09b(my_mutex_09);
   tbb::null_rw_mutex::scoped_lock           mylock10a(my_mutex_10);
   // null ничего не делает... это не приводит к зависанию...
   tbb::null_rw_mutex::scoped_lock           mylock10b(my_mutex_10);
   // null ничего не делает... это не приводит к зависанию...
   tbb::null_rw_mutex::scoped_lock           mylock10c(my_mutex_10);
   printf("Locks acquired!\nHello, World!\n");
  }
}
```

Если вы не станете раскомментировать строки, которые привели бы к зависанию, будет напечатано вот что:

```
Locks acquired!
Hello, World!
```

Рис. В.60 (*продолжение*)

Концепции мьютексов	
Концепция мьютекса (Mutex) для мьютекса типа M	
Псевдосигнатура	**Семантика**
`M()`	Конструирует незаблокированный мьютекс
`~M()`	Уничтожает незаблокированный мьютекс. Попытка уничтожить заблокированный мьютекс приводит к неопределенному поведению
`typename M::scoped_lock`	Соответствующий тип блокировки, ограниченной областью видимости
`M::scoped_lock()`	Конструирует блокировку без захвата мьютекса
`M::scoped_lock(M&)`	Конструирует блокировку и захватывает мьютекс
`M::~scoped_lock()`	Освобождает блокировку (если она захвачена)
`M::scoped_lock::acquire(M&)`	Захватывает мьютекс
`bool M::scoped_lock::try_acquire(M&)`	Пытается захватить мьютекс. Возвращает `true`, если мьютекс захвачен, иначе `false`
`M::scoped_lock::release()`	Освобождает мьютекс
`static const bool M::is_rw_mutex`	`true`, если мьютекс чтения/записи, иначе `false`
`static const bool M::is_recursive_mutex`	`true`, если рекурсивный мьютекс, иначе `false`
`static const bool M::is_fair_mutex`	`true`, если справедливый мьютекс, иначе `false`
Концепция мьютекса чтения/записи (ReaderWriterMutex) расширяет концепцию мьютекса (*все приведенные выше требования применимы*). Классы `spin_rw_mutex`, `speculative_spin_rw_mutex` и `queuing_spin_rw_mutex` моделируют концепцию мьютекса чтения/записи. Ниже перечислены дополнительные требования к мьютексу чтения/записи.	
`RW::scoped_lock(RW&, bool write=true)`	Конструирует блокировку и захватывает мьютекс. Блокировка по типу записи, если параметр `write` равен `true`, иначе по типу чтения
`RW::scoped_lock::acquire(RW&, bool write=true)`	Захватывает мьютекс. Блокировка по типу записи, если параметр `write` равен `true`, иначе по типу чтения
`bool RW::scoped_lock::try_acquire(RW&, bool write=true)`	Пытается захватить мьютекс. Блокировка по типу записи, если параметр `write` равен `true`, иначе по типу чтения. Возвращает `true`, если мьютекс захвачен, иначе `false`
`bool RW::scoped_lock::upgrade_to_writer()`	Изменяет вид блокировки с чтения на запись. Если блокировка по типу чтения не была захвачена, то поведение не определено. Возвращает `false`, если блокировка была освобождена и снова захвачена, иначе `true`
`bool RW::scoped_lock::downgrade_to_reader()`	Изменяет вид блокировки с записи на чтение. Если блокировка по типу записи не была захвачена, то поведение не определено. Возвращает `false`, если блокировка была освобождена и снова захвачена, иначе `true`

Рис. В.61 ❖ Синхронизация: концепции мьютексов

Класс `atomic<T>`

`#include <tbb/atomic.h>`

Тип `atomic<T>` поддерживает операции атомарного чтения, записи, выборки-и-сложения, выборки-и-сохранения и сравнения-с-обменом. Тип `T` может быть целочисленным, перечислительным или указательным. Если тип `T` указательный, то арифметические операции интерпретируются в смысле арифметики указателей. Например, если `x` имеет тип `atomic<float*>` и тип `float` занимает четыре байта, то `++x` продвигает `x` на четыре байта вперед. Арифметические операции с `atomic<T>` не разрешены, если `T` – перечислительный тип, `void*` или `bool`.

Для всех атомарных операций стандарт C++ предусматривает семантику последовательной согласованности памяти, но в TBB допускаются отклонения от этого умолчания.

```
#include <tbb/atomic.h>
int main( int argc, char *argv[] ) {
  double value = 9.9;
  tbb::atomic<double> y(value); // Не атомарно
  tbb::atomic<double> z;
  z=value;                      // Атомарное присваивание
  return 0;
 }
```

Член	Описание
`enum memory_semantics`	Определяет значения, используемые для выбора варианта шаблона, который допускает более избирательный контроль видимости операций (см. ниже)

Вид (enum)	Описание	По умолчанию для
acquire	Операции после атомарной никогда не накладываются на нее	чтение
release	Операции до атомарной никогда не накладываются на нее	запись
full_fence	Последовательная согласованность. Операции по обе стороны никогда не накладываются, последовательно согласованные атомарные операции всегда имеют глобальный порядок	`fetch_and_store`, `fetch_and_add`, `compare_and_swap`
relaxed	Без упорядочения	–

`atomic() = default`	Конструктор по умолчанию, генерируемый компилятором
`constexpr atomic( value_type arg)`	Инициализировать `*this` значением `arg`. Если аргумент является константой времени компиляции, то инициализация производится на этапе компиляции, в противном случае на этапе выполнения
`template<memory_semantics M> value_type fetch_and_add( value_type addend )`	Пусть `x` – значение `*this`. Атомарно производит обновление `x = x + addend`. Возвращает первоначальное значение `x`
`template<memory_semantics M> value_type fetch_and_increment()`	Пусть `x` – значение `*this`. Атомарно производит обновление `x = x + 1`. Возвращает первоначальное значение `x`
`template<memory_semantics M> value_type fetch_and_decrement()`	Пусть `x` – значение `*this`. Атомарно производит обновление `x = x - 1`. Возвращает первоначальное значение `x`
`template<memory_semantics M> value_type compare_and_swap( value_type new_value, value_type comparand )`	Пусть `x` – значение `*this`. Атомарно сравнивает `x` с `comparand` и, если они равны, присваивает `x=new_value`. Возвращает первоначальное значение `x`
`template<memory_semantics M> value_type fetch_and_store( value_type new_value )`	Пусть `x` – значение `*this`. Атомарно обменивает `x` с `new_value`. Возвращает первоначальное значение `x`
`template<memory_semantics M> value_type load()`	Возвращает значение `x`
`template<memory_semantics M> void store(value_type new_value)`	Атомарно сохраняет `new_value`

Рис. В.62 ❖ Синхронизация: класс `atomic<T>`

Поточно-локальная память (TLS)

В главе 5 поточно-локальная память в TBB рассматривается в более широком контексте синхронизации; в этом разделе приводится краткий обзор поддерживаемых API. Для наших целей достаточно считать, что поточно-локальная память состоит из приватизированных копий данных в каждом потоке. Важная особенность TBB заключается в том, что мы не знаем, сколько потоков используется в каждый момент времени, поэтому поточно-локальная память организована так, что автоматически соответствует количеству потоков, создаваемых библиотекой TBB на этапе выполнения, а оно может сильно зависеть от платформы, на которой работает программа.

TBB предлагает два шаблонных класса поточно-локальной памяти. Оба гарантируют хранение локального экземпляра элемента в каждом потоке. И оба отложенно создают элементы по запросу. Различаются они предполагаемым способом использования:

- класс `combinable` предоставляет поточно-локальную память для выполнения в каждом потоке частичных вычислений, которые впоследствии будут редуцированы в один результат;
- класс `enumerable_thread_specific` предоставляет поточно-локальную память, которая работает как STL-контейнер, содержащий по одному элементу в каждом потоке. Этот контейнер допускает обход с помощью стандартной идиомы STL-итератора.

Шаблонный класс `flatten2d` поддерживает идиому, согласно которой `enumerable_thread_specific` представляет разбиватель контейнера на потоки. Он включен, потому что очень полезен при отладке программы.

combinable

#include <**tbb**/atomic.h>
Шаблонный класс `atomic<T>`.

Член	Описание
combinable()	Конструирует объект `combinable`, так что поточно-локальные экземпляры `T` создаются конструктором по умолчанию
`template<typename FInit>` `explicit` **combinable**(`FInit finit`)	Конструирует объект `combinable`, так что поточно-локальные экземпляры `T` создаются путем копирования результата `finit()`. Вычисление выражения `finit()` в нескольких потоках должно быть безопасно. Оно вычисляется при каждом создании нового поточно-локального элемента
combinable(`const combinable& other`)	Конструирует копию `other`, так что она содержит копии всех элементов `other` в тех же потоках, что оригиналы
combinable(`combinable&& other`)	Конструирует объект `combinable`, перемещая содержимое `other`. Объект `other` остается в неопределенном состоянии, но может быть безопасно уничтожен
~combinable()	Уничтожает все элементы `*this`
`combinable&` **operator=** (`const combinable& other`)	Присваивает `*this` копию `other`

Рис. B.63 ❖ TLS: класс `combinable` (*начало*)

`combinable& operator= ( combinable&& other )`	Перемещает в `*this` содержимое `other`. Объект `other` остается в неопределенном состоянии, но может быть безопасно уничтожен
`void clear()`	Удаляет все элементы из `*this`
`T& local( [bool& exists] )`	Элемент создается, если в текущем потоке его еще не существует. Кроме того, если задан параметр `exists`, то ему присваивается значение `true`, если элемент уже присутствовал в текущем потоке, и `false`, если его пришлось создать. Возвращает ссылку на поточно-локальный элемент
`template<typename BinaryFunc> T combine(BinaryFunc a)`	Вычисляет редукцию всех элементов, применяя бинарный функтор `f`. Вычисления `f` производятся последовательно в вызывающем потоке. Если элементов нет, то результат создается по правилам создания нового элемента. Возвращает результат редукции
`template<typename UnaryFunc> void combine_each(UnaryFunc a)`	Вычисляет `f(x)` для каждого поточно-локального элемента `x` в `*this`. Все вычисления производятся последовательно в вызывающем потоке

Рис. В.63 (*окончание*)

combinable – пример

```
#include <tbb/parallel_for.h>
#include <tbb/combinable.h>
#include <iostream>
#include <stdio.h>

#define HOWMANY 10

void dump(tbb::combinable<int> *pTLS) {
  tbb::parallel_for( 0, HOWMANY, [&](int i){
    bool truth = false;
    int val = 0;
    val = pTLS->local(truth);
    printf("%d%c ",val,truth ? 'T':'F');
  } );
  printf("\n");
}

int main() {
  int gval;
  tbb::combinable<int> myTLS([](){return 0;});
  tbb::combinable<int> mycopiedTLS([](){return 6;});

  dump( &myTLS );

  myTLS.clear();
  printf("cleared\n");

  dump( &myTLS ); dump( &myTLS );
  tbb::parallel_for( 0, HOWMANY, [&](int i){
     myTLS.local() += i;
  } );
  printf("added local values into local sums\n");
  dump( &myTLS );

  gval = myTLS.combine([](int a,int b){return a+b;});
  printf("global value = %d\n",gval);
  mycopiedTLS = myTLS;
  gval = mycopiedTLS.combine([](int a,int b){return a+b;});
  printf("global copied value = %d\n",gval);
```

Рис. В.64 ❖ TLS: класс `combinable` (*начало*)

```
  myTLS.clear();
  printf("cleared\n");
  gval = myTLS.combine([](int a,int b){return a+b;});
  printf("global value = %d\n",gval);
  gval = mycopiedTLS.combine([](int a,int b){return a+b;});
  printf("global copied value = %d\n",gval);
  mycopiedTLS.combine_each([](int a){printf("%d ",a);});
  printf("<< values from combine_each\n");
  gval = mycopiedTLS.combine([](int a,int b){return a+b;});
  printf("global copied value = %d\n",gval);
  return 0;
}
```

Пример вывода – порядок может отличаться от прогона к прогону и от машины к машине!

```
0F 0T 0T 0T 0T 0T 0F 0F 0T 0T
cleared
0F 0T 0T 0T 0T 0T 0F 0F 0T 0T
0T 0T 0T 0T 0F 0T 0T 0T 0T 0T
added local values into local sums
45T 45T 45T 45T 45T 0T 0T 0T 0T 45T
global value = 45
global copied value = 45
cleared
global value = 0
global copied value = 45
0 0 45 0 << values from combine_each
global copied value = 45
```

Рис. В.64 (*окончание*)

enumerable_thread_specific

#include <**tbb**/enumerable_thread_specific.h>

Шаблонный класс enumerable_thread_specific<T> предоставляет каждому потоку хранилище объектов типа T, которое ведет себя как STL-контейнер с одним элементом на каждый поток. Контейнер допускает обход элементов с помощью стандартных идиом STL-итераторов.

Член enumerable_thread_specific	Описание
Конструкторы	
enumerable_thread_specific()	Конструирует объект enumerable_thread_specific, так что поточно-локальные элементы создаются конструктором по умолчанию
template<typename Finit> explicit **enumerable_thread_specific**(Finit finit)	Конструирует объект enumerable_thread_specific, так что поточно-локальные элементы создаются путем копирования результата finit(). Вычисление выражения finit() в нескольких потоках должно быть безопасно. Оно вычисляется при каждом создании нового поточно-локального элемента. Этот конструктор доступен (т. е. участвует в разрешении перегрузки), только если finit() – допустимое выражение
explicit **enumerable_thread_specific**(const T& exemplar) explicit **enumerable_thread_specific**(T&& exemplar)	Конструирует объект enumerable_thread_specific, так что поточно-локальные элементы создаются путем копирования exemplar. Перемещающий конструктор T можно использовать для внутреннего хранения exemplar, но поточно-локальные элементы всегда создаются копирующим конструктором
template<typename... Args> **enumerable_thread_specific**(Args... args)	Конструирует объект enumerable_thread_specific, так что поточно-локальные элементы создаются путем вызова T(args…). Этот конструктор не участвует в разрешении перегрузки, если первый аргумент в args... имеет тип T или enumerable_thread_specific<T> или если foo() – допустимое выражение для значения foo этого типа
enumerable_thread_specific(const **enumerable_thread_specific**& other) template<typename Alloc, ets_key_usage_type Cachetype> **enumerable_thread_specific**(const **enumerable_thread_specific** <T, Alloc, Cachetype>& other)	Конструирует объект enumerable_thread_specific как копию other. Значения поточно-локальных элементов создаются из other копирующим конструктором и находятся в тех же потоках, что оригиналы
enumerable_thread_specific (enumerable_thread_specific& other)	Конструирует объект enumerable_thread_specific, перемещая содержимое other. Объект other остается в неопределенном состоянии, но может быть безопасно уничтожен
template<typename Alloc, ets_key_usage_type Cachetype> **enumerable_thread_specific**(const enumerable_thread_specific <T, Alloc, Cachetype>&& other)	Конструирует объект enumerable_thread_specific, применяя перемещающий конструктор к каждому значению other, так что копии остаются в тех же потоках, что оригиналы. Объект other остается в неопределенном состоянии, но может быть безопасно уничтожен

Рис. В.65 ❖ TLS: класс enumerable_thread_specific (*начало*)

Операции, применимые к контейнеру в целом Эти операции не должны применяться конкурентно к одному и тому же экземпляру `enumerable_thread_specific`.	
`~enumerable_thread_specific()`	Уничтожает все элементы в `*this`. Удаляет все платформенные ключи TLS, которые были созданы для данного экземпляра
`enumerable_thread_specific& operator= ( const enumerable_thread_specific& other )`	Копирует в `*this` содержимое `other`
`template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific& operator=( const enumerable_thread_specific <T, Alloc, Cachetype>& other )`	Копирует в `*this` содержимое `other`. После этого вызова распределитель и специализация использования ключей остаются прежними
`enumerable_thread_specific& operator= ( enumerable_thread_specific&& other );`	Перемещает в `*this` содержимое `other`. Объект `other` остается в неопределенном состоянии, но может быть безопасно уничтожен
`template<typename Alloc, ets_key_usage_type Cachetype> enumerable_thread_specific& operator=( enumerable_thread_specific <T, Alloc, Cachetype>&& other )`	Перемещает в `*this` содержимое `other`, применяя перемещающий конструктор к каждому значению `other`, так что копии остаются в тех же потоках, что оригиналы. Объект `other` остается в неопределенном состоянии, но может быть безопасно уничтожен После этого вызова распределитель и специализация использования ключей остаются прежними
`void clear()`	Уничтожает все элементы в `*this`. Удаляет, а затем заново создает все платформенные ключи TLS, которые были созданы для данного экземпляра
Конкурентные операции	
`reference local( [ bool& exists ] )`	Элемент создается, если в текущем потоке его еще не существует. Кроме того, если задан параметр `exists`, то ему присваивается значение `true`, если элемент уже присутствовал в текущем потоке, и `false`, если его пришлось создать. Возвращает ссылку на поточно-локальный элемент
`size_type size() const`	Возвращает количество элементов в `*this`. Значение равно количеству различных потоков, которые вызывали `local()` после конструирования или последней очистки `*this`
`bool empty() const`	Возвращает `size() == 0`
Операции комбинирования Эти методы обходят контейнер последовательно в вызывающем потоке.	
`template<typename BinaryFunc> T combine(BinaryFunc a)`	Вычисляет редукцию всех элементов, применяя бинарный функтор `f`. Если элементов нет, то результат создается по правилам создания нового элемента. Возвращает результат редукции
`template<typename UnaryFunc> void combine_each(UnaryFunc a)`	Вычисляет `f(x)` для каждого экземпляра `x` в `*this`
Параллельное итерирование	
`[const_]range_type range( size_t grainsize=1 ) [const]`	Возвращает объект типа `[const_]range_type`, представляющий все элементы в `*this`. Параметр `grainsize` измеряется в элементах
Итераторы Итераторы доступа, позволяющие обойти множество всех элементов в контейнере.	
`[const_]iterator begin() [ const ]`	Возвращает `[const_]iterator`, указывающий на начало множества элементов
`[const_]iterator end() [ const ]`	Возвращает `[const_]iterator`, указывающий на позицию за концом множества элементов

Рис. B.65 (*окончание*)

enumerable_thread_specific – пример

```
#include <cstdio>
#include <utility>
#include <tbb/task_scheduler_init.h>
#include <tbb/enumerable_thread_specific.h>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

typedef tbb::enumerable_thread_specific< std::pair<int,int> > CounterType;
CounterType MyCounters (std::make_pair(0,0));

struct Body {
  void operator()(const tbb::blocked_range<int>& r) const {
  CounterType::reference my_counter = MyCounters.local();
  ++my_counter.first;
  for (int i = r.begin(); i != r.end(); ++i)
    ++my_counter.second;
  }
};

int main() {
  tbb::parallel_for(
    tbb::blocked_range<int>(0, 100000000), Body());

    for (CounterType::const_iterator i = MyCounters.begin();
         i != MyCounters.end(); ++i) {
      printf("Thread stats:\n");
      printf("      calls to operator(): %d", i->first);
      printf("      total # of iterations executed: %d\n",
             i->second);
    }
    std::pair<int,int> sum =
      MyCounters.combine([](std::pair<int,int> x,
                            std::pair<int,int> y) {
        return std::make_pair(x.first+y.first,
                              x.second+y.second);
      });
    printf("Total calls to operator() = %d, "
           "total iterations = %d\n", sum.first, sum.second);
}
```

Пример вывода – порядок может отличаться от прогона к прогону и от машины к машине!

```
Thread stats:
      calls to operator(): 63      total # of iterations executed: 7812500
Thread stats:
      calls to operator(): 63      total # of iterations executed: 9375000
Thread stats:
      calls to operator(): 256      total # of iterations executed: 81250000
Thread stats:
      calls to operator(): 21      total # of iterations executed: 1562500
Total calls to operator() = 403, total iterations = 100000000
```

Рис. В.66 ❖ TLS: пример enumerable_thread_specific

flatten2d

`#include <tbb/enumerable_thread_specific.h>`

Шаблонный класс `flatten2d` поддерживает широко употребительную идиому, согласно которой `enumerable_thread_specific` представляет разбиватель контейнера на потоки. Объект `flatten2d` дает плоское представление контейнера контейнеров, что помогает отлаживать приложения.

Член flatten2d	Описание
`explicit flatten2d( const Container& c )`	Конструирует объект `flatten2d`, представляющий последовательность элементов во внутренних контейнерах, содержащихся во внешнем контейнере `c`
`flatten2d( const Container& c, typename Container::const_iterator first, typename Container::const_iterator last)`	Конструирует объект `flatten2d`, представляющий последовательность элементов во внутренних контейнерах из полуоткрытого интервала `[first, last)` контейнера `c`
`size_type size() const`	Возвращает суммарный размер внутренних контейнеров, видимых в представлении `flatten2d`
`iterator begin()`	Возвращает итератор, указывающий на начало множества локальных копий
`iterator end()`	Возвращает итератор, указывающий на позицию за концом множества локальных копий
`template<typename Container> flatten2d<Container> flatten2d<const Container& c>, const typename Container::const_iterator b, const typename Container::const_iterator e)`	Конструирует и возвращает объект `flatten2d`, предоставляющий итераторы для обхода элементов в контейнерах из полуоткрытого интервала `[first, last)` контейнера `c`
`template<typename Container> flatten2d( const Container& c )`	Конструирует и возвращает объект `flatten2d`, предоставляющий итераторы для обхода элементов во всех контейнерах внутри контейнера `c`

Рис. B.67 ❖ Пример TLS::flatten2d

flatten2d – пример

```
#include <iostream>
#include <utility>
#include <vector>
#include <tbb/task_scheduler_init.h>
#include <tbb/enumerable_thread_specific.h>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>
// В типе VecType имеется по одному вектору std::vector<int> на каждый поток
typedef
  tbb::enumerable_thread_specific< std::vector<int> >
  VecType;

VecType MyVectors;
int k = 10000000; // в 10 раз больше, чтобы пример вывода поместился в книге

struct Func {
  void operator()(const tbb::blocked_range<int>& r) const {
    VecType::reference v = MyVectors.local();
    for (int i=r.begin(); i!=r.end(); ++i)
      if( i%k==0 )
        v.push_back(i);
  }
};

int main() {
  tbb::parallel_for(
    tbb::blocked_range<int>(0, 100000000), Func());

  tbb::flattened2d<VecType> flat_view =
      tbb::flatten2d( MyVectors );

  for( tbb::flattened2d<VecType>::const_iterator
       i = flat_view.begin(); i != flat_view.end(); ++i)
       std::cout << *i << std::endl;

  return 0;
}
```

Пример вывода – порядок может отличаться от прогона к прогону и от машины к машине!

```
0
10000000
20000000
50000000
60000000
30000000
80000000
40000000
70000000
90000000
```

Рис. В.68 ❖ TLS: пример `flatten2d`

Хронометраж

TBB включает платформенно-независимый, поддерживающий потоки способ получения временной метки с высоким разрешением (TBB стремится использовать таймер с самым высоким разрешением, существующий на данной платформе). Это средство было включено в TBB, чтобы помочь в отладке и настройке кода – вещь естественная для любого программиста, желающего распараллелить код и оценить результат.

tick_count – класс для надежного измерения времени

#include <**tbb**/tick_count.h>

Класс tbb::tick_count представляет абсолютную временную метку. Для вычисления относительного времени один объект tbb::tick_count можно вычесть из другого. В результате получается объект типа tbb::tick_count::interval_t, такие объекты можно складывать, вычитать и преобразовывать в секунды.

В Linux при использовании класса tbb::tick_count следует компоновать программу, добавив флаг -lrt.

```
#include <cstdio>
#include <thread>
#include <chrono>
#include <tbb/tick_count.h>
volatile int foo = 4;
int main( int argc, char *argv[] ) {
  tbb::tick_count t0 = tbb::tick_count::now();
  while (foo--) std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  tbb::tick_count t1 = tbb::tick_count::now();
  printf("resolution for timing on this platform is =\n%12.8f seconds\n",
         tbb::tick_count::resolution() );
  printf("time for action =\n%12.8f seconds\n", (t1-t0).seconds() );
  }
```

Пример вывода – порядок может отличаться от прогона к прогону и от машины к машине!

```
resolution for timing on this platform is =
  0.00000100 seconds
time for action =
  4.00328800 seconds
```

Член tbb::tick_count	Описание
static tick_count **now**()	Текущее время
static double **resolution**()	Разрешающая способность таймера в секундах
tick_count::interval_t **operator-**(const tick_count& t1, const tick_count& t0)	Время, прошедшее с момента t0 до t1
Член tbb::tick_count::interval	**Описание**
interval_t()	Конструирует объект interval_t, представляющий нулевую продолжительность
interval_t(double sec)	Конструирует объект interval_t, представляющий заданное число секунд
double **seconds**() const	Возвращает длительность интервала в секундах
операции **+= -= + -**	Вычисляют сумму и разность значений типа interval_t

Рис. В.69 ❖ Хронометраж: класс tick_count

Группы задач: использование планировщика с заимствованием задач

В этом разделе приведена сводка высокоуровневых API планировщика задач в TBB, а в следующем описаны более многочисленные низкоуровневые API. Высокоуровневые API позволяют легко создавать группы потенциально параллельных задач из функторов или лямбда-выражений (предисловие, главы 1–3). Планировщик задач лежит в основе всех TBB-алгоритмов (главы 2–16) и потоковых графов (главы 3 и 17–19).

Функторы, являющиеся аргументами различных методов, в этом разделе должны как минимум обладать копирующим конструктором, деструктором и оператором применения `()`.

Классы `task_group` и `structured_task_group`

`#include <tbb/task_group.h>`

Член	**Описание**
`template<typename Func> void` **`run`**`( Func&& f )`	Запускает задачу для вычисления `f()` и сразу возвращает управление
`template<typename Func> void` **`run`**`( task_handle<Func>& handle )`	Запускает задачу для вычисления `handle()` и сразу возвращает управление
`template<typename Func> task_group_status` **`run_and_wait`**`( const Func& f )`	Эквивалентно `{ run(f); return wait();}`, но гарантируется, что `f()` работает в текущем потоке. *Примечание*: идея в том, что шаблонный метод `run_and_wait` должен быть эффективнее, чем отдельные вызовы `run()` и `wait()`
`task_group_status` **`wait`**`()`	Ожидает завершения или отмены всех задач в группе. Возвращает состояние `task_group`
`enum` **`task_group_status`** `{ not_complete, // Не отменена и не все задачи в группе завершились complete, // Не отменена, все задачи в группе завершились cancelled // Группа задач получила запрос на отмену };`	
`bool` **`is_cancelling`**`()`	Возвращает `true`, если эта группа задач находится в процессе отмены
`void` **`cancel`**`()`	Отменяет все задачи в группе
`tbb::`**`is_current_task_group_cancelling`**`()`	`true`, если самая глубоко вложенная группа задач, исполняемая в текущем потоке, находится в процессе отмены

Класс **`structured_task_group`** похож на `task_group`, но обладает лишь подмножеством его функциональности. В будущем он, возможно, будет подвергнут оптимизации. Ниже перечислены ограничения:

- методы `run` и `run_and_wait` принимают только аргументы типа `task_handle`, а не функторы общего вида. Класс `task_handle` создан специально для использования совместно с `structured_task_group`. Но для единообразия `task_group` тоже принимает аргументы типа `task_handle`;
- методы `run` и `run_and_wait` не копируют свои аргументы `task_handle`. Вызывающая сторона не должна уничтожать эти аргументы, пока метод `wait` или `run_and_wait` не вернул управление;
- методы `run`, `run_and_wait`, `cancel` и `wait` должны вызываться только из потока, в котором был создан объект `structured_task_group`;
- объект `task_handle` создается из функции или функтора с помощью шаблонной функции `task_handle`:

  `template<typename TFunc> task_handle<Func> make_task( TFunc&& f );`

Метод `wait` (или `run_and_wait`) следует вызывать только один раз для каждого экземпляра `structured_task_group`.

Рис. B.70 ❖ Группы задач: пример `[structured_]task_group`

Классы task_group и structured_task_group

```
#include <stdio.h>
#include <tbb/task_group.h>
#define FIB(X) printf("Fib(%d) = %d\n",X,Fib(X))
int Fib(int n) {
  if( n<2 ) {
    return n;
  } else {
    int x, y;
    tbb::task_group g;
    g.run([&]{x=Fib(n-1);}); // запустить задачу
    g.run([&]{y=Fib(n-2);}); // запустить еще одну задачу
    g.wait();                // ждать завершения обеих задач
    return x+y;
  }
}
int main( int argc, char *argv[] ) {
  FIB(1);
  FIB(2);
  FIB(3);
  FIB(11);
  FIB(20);
  printf(tbb::is_current_task_group_canceling() ?
         "Cancelling\n" : "Not cancelling.\n");
}
```

```
Fib(1) = 1
Fib(2) = 1
Fib(3) = 2
Fib(11) = 89
Fib(20) = 6765
```

Рис. B.71 ❖ Группы задач: пример task_group

Планировщик задач: точный контроль над планировщиком с заимствованием задач

В этом разделе приведена сводка (многочисленных) низкоуровневых API планировщика задач в TBB, а в предыдущем были рассмотрены высокоуровневые API. Четыре основные концепции, обсуждаемые в этом разделе: планировщик задач, арены задач, задачи и средства настройки вычислений с плавающей точкой. Низкоуровневые интерфейсы дают точный контроль над распространением исключений (глава 15), приоритетами (глава 4), изоляцией (глава 12) и привязкой (главы 13 и 20). Планировщик задач со всем, что к нему относится, лежит в основе всех TBB-алгоритмов (главы 2–16) и потоковых графов (главы 3 и 17–19).

Планировщик задач: класс task_scheduler_init

#include <**tbb**/task_scheduler_init.h>

Конструктор

```
class task_scheduler_init(
  int max_threads=automatic
 [ , stack_size_type thread_stack_size ]
)
```

Класс **tbb::task_scheduler_init** явно выражает интерес потока к службам планирования задач. TBB автоматически создает планировщик задач при первом использовании потока служб планирования и уничтожает его, когда последний такой поток завершается, а это означает, что явный вызов для задания нестандартных параметров должен иметь место до того, как служба планирования задач будет вызвана неявно. Кроме того, это означает, что явную инициализацию следует использовать только для отладки или если имеются особые причины управлять инициализацией самостоятельно.

Параметр `max_threads` может быть целым положительным числом. Он означает, что в каждый момент времени может работать не более `max_threads-1` рабочих потоков, созданных по инициативе вызывающего потока. Имеется также два специальных значения: `tbb::task_scheduler_init::automatic` означает, что TBB автоматически задаст `max_threads`, исходя из имеющегося оборудования (это значение подразумевается по умолчанию), а `tbb::task_scheduler_init::deferred` – что активацию следует отложить до вызова метода `initialize()` или автоматической инициализации по требованию TBB-алгоритма, пользующегося услугами планировщика задач. Значение параметра `max_threads` вполне может зависеть от вызывающего потока. Например, если поток A задает `max_threads=4`, а поток B – `max_threads=12`, то A ограничен 3 рабочими потоками, а B может иметь до 11 таких потоков. Поскольку рабочие потоки могут разделяться между A и B, общее количество рабочих потоков, созданных планировщиком, может достигать 11.

Необязательный параметр `thread_stack_size` задает размер стека каждого рабочего потока. Значение 0 означает, что нужно использовать размер стека по умолчанию. Первый созданный экземпляр `tbb::task_scheduler_init` определяет размер стека всех рабочих потоков

Инициализация

`void` **`initialize`**`( int max_threads=automatic )`

Активирует экземпляр `task_scheduler_init`, который не был активирован ранее

Деструктор

`~task_scheduler_init`

Если `task_scheduler_init` не активен, то не происходит ничего. В противном случае `task_scheduler_init` деактивируется следующим образом. Если в данном потоке нет других активных объектов `task_scheduler_init`, то поток освобождает память, занятую внутренними относящимися к этому потоку ресурсами, которые необходимы для планирования задач. Если ни в каком потоке нет других объектов `task_scheduler_init`, то все внутренние рабочие потоки останавливаются

Деактивация

`void` **`terminate`**`()`

Деактивирует `task_scheduler_init`, не уничтожая его. О том, что такое деактивация, см. в описании деструктора

Запросы

`static int` **`default_num_threads`**`()`

Возвращает число, на единицу большее количества рабочих потоков, создаваемых объектом `task_scheduler_init` по умолчанию

`bool` **`is_active`**`() const`

Возвращает `true`, если `task_scheduler_init` был инициализирован, в противном случае `false`

Рис. B.72 ❖ Планировщик задач: класс `task_scheduler_init`

Планировщик задачи: класс task_scheduler_init

```
#include <tbb/task_scheduler_init.h>

#define XYZZY(FLAG) \
  printf(FLAG " default threads = %d; ", \
         tbb::task_scheduler_init::default_num_threads()); \
  printf(my_init.is_active()?"TBB activate\n":"TBB not active\n");

int main( int argc, char *argv[] ) {
  {
    tbb::task_scheduler_init my_init(tbb::task_scheduler_init::deferred);
    XYZZY("AA"); my_init.initialize(10);
    XYZZY("AB");
  }
  {
    tbb::task_scheduler_init my_init;
    XYZZY("BB");
    // если раскомментировать следующую строку:
    // my_init.initialize(24);
    // то исполняющая среда аварийно завершится с ошибкой:
    // Assertion !my_scheduler failed...
    // Detailed description: task_scheduler_init already initialized
    // Abort trap: 6
    XYZZY("BC"); my_init.terminate();
    XYZZY("BD"); my_init.initialize(90);
    XYZZY("BE");
  }
}
```

```
AA default threads = 4; TBB not active
AB default threads = 4; TBB activate
BB default threads = 4; TBB activate
BC default threads = 4; TBB activate
BD default threads = 4; TBB not active
BE default threads = 4; TBB activate
```

Рис. В.73 ❖ Планировщик задач: пример `task_scheduler_init`

Арены задач: класс task_arena

#include <**tbb**/task_arena.h>

Класс tbb::task_arena представляет место, на котором потоки могут разделять и исполнять задачи.
Количество одновременно выполняемых на арене задач ограничено уровнем конкурентности. Уровень конкурентности влияет только на данную арену и не зависит от ранее созданных объектов task_scheduler_init. Общее количество потоков, доступных планировщику, ограничено наибольшим из двух чисел: количество потоков на данной машине по умолчанию и значение, заданное при первой инициализации планировщика задач. Поэтому количество задач, назначенных арене, никогда не превысит этого максимального значения, независимо от уровня конкурентности. Но арена задач может не получить заданного количества потоков, даже если оно меньше допустимого максимума. Экземпляр task_arena запоминает ссылку на внутреннее представление, но не полностью управляет его временем жизни. Внутреннее представление не может быть уничтожено, пока оно содержит задачи или на него ссылаются какие-то другие потоки.
Конструкторы task_arena не создают внутренний объект арены. Он может уже существовать в случае «присоединяющего» конструктора, а в противном случае создается явным вызовом функции task_arena::inititalize() или отложенно при первом использовании.
В пространстве имен this_task_arena находятся глобальные функции для взаимодействия с ареной (явно или неявно созданным объектом task_arena), которая в данный момент используется вызывающим потоком.

static const int automatic	Если конструктору передано это значение в аргументе max_concurrency, то уровень конкурентности арены будет автоматически установлен в зависимости от конфигурации оборудования
static const int not_initialized	Если это значение возвращено функцией, значит, активной арены не существует или объект арены еще не был инициализирован

Конструкторы, деструктор

Член	Описание

task_arena(
  int max_concurrency = automatic,
  unsigned reserved_for_masters = 1)
	Создает арену задач с некоторым предельным уровнем конкурентности (max_concurrency). Часть емкости арены может быть зарезервирована для потоков, созданных самим приложением, с помощью параметра reserved_for_masters. Количество зарезервированных мест не может превышать лимита конкурентности. Если и max_concurrency, и reserved_for_masters явно заданы равными и не меньшими 1, то рабочие потоки TBB никогда не зайдут на эту арену. В результате гарантия выполнения для стоящих в очереди потоков на такую арену не распространяется. Не следует использовать функции task_arena::enqueue и task::enqueue для арены, не имеющей рабочих потоков

task_arena(const task_arena&)	Копирует значения параметров из другого экземпляра task_arena
explicit task_arena(task_arena::attach)	Создает экземпляр task_arena, связанный с внутренней ареной, которая в данный момент используется вызывающим потоком. Если такой арены еще не существует, то создается объект task_arena с параметрами по умолчанию. В отличие от других конструкторов, этот автоматически инициализирует новый объект task_arena при подключении к уже существующей арене

Рис. В.74 ❖ Планировщик задач: класс task_arena (*начало*)

`void initialize()`
 	
`void initialize(`
`  int max_concurrency,`
`  unsigned reserved_for_masters = 1)`
 	
`void initialize(task_arena::attach)`
	Выполняет фактическую инициализацию внутреннего представления арены. Если аргументы заданы, то они переопределяют прежние параметры арены. Если в качестве аргумента задано значение `task_arena::attach` и существует внутренняя арена, используемая в данный момент вызывающим потоком, то метод игнорирует параметры арены и присоединяет объект `task_arena` к этой внутренней арене. Метод ничего не делает, если вызван для уже инициализированной арены. После обращения к `initialize` параметры арены фиксированы и не могут быть изменены

`void terminate()`	Удаляет ссылку на внутреннее представление арены, не уничтожая объект `task_arena`, который впоследствии может быть использован повторно. Небезопасен относительно конкурентных вызовов других методов
`~task_arena()`	Удаляет ссылку на внутреннее представление арены и уничтожает объект `task_arena`. Небезопасен относительно конкурентных вызовов других методов
Запрос, постановка в очередь, выполнение	
Член	**Описание**
`bool is_active() const`	Возвращает `true`, если арена инициализирована, иначе `false`
`int max_concurrency() const`	Возвращает уровень конкурентности для данной арены. Не требует, чтобы арена была инициализирована, и не инициализирует ее
`template<F> void enqueue(F&& f)`	Ставит задачу в очередь на арене для обработки заданного функтора (который копируется или перемещается в объект задачи), после чего сразу же возвращает управление. Метод не требует, чтобы вызывающий поток был присоединен к арене, т. е. сколько угодно потоков вне арены могут отправлять ей задачи без блокировки. Не гарантируется, что поставленные в очередь задачи будут выполняться одновременно с другими задачами на той же арене. Если внутри функтора возбуждается, но не перехватывается исключение, то поведение не определено
`template<F> void enqueue(F&& f,` `  priority_t)`	Ставит задачу в очередь с заданным приоритетом. Во всех остальных отношениях ведет себя так же, как **`enqueue(f)`**

`template<F> auto execute(F&) ->`
`  decltype(f())`
 	
`template<F> auto execute(const F&) ->`
`  decltype(f())`
	Выполняет указанный функтор на арене. Функция возвращает значение, возвращенное функтором.
	Вызывающий поток присоединяется к арене, если это возможно, и исполняет функтор. После возврата он восстанавливает состояние планировщика и параметры плавающей точки для предыдущей задачи.
	Если присоединиться к арене невозможно, то вызов обертывает функцию задачей, ставит ее в очередь на арене и с помощью объекта синхронизации ОС ждет возможности присоединиться. Заканчивает работу после завершения задачи.
	Исключение, возбужденное внутри функтора, перехватывается и возбуждается повторно. Если механизм распространения точной копии исключения отсутствует или выключен, то исключение обертывается объектом `tbb::captured_exception`, даже если оно возникло в том же самом потоке

Рис. B.74 (*окончание*)

Члены пространства имен this_task_arena	
`int current_thread_index()`	Возвращает индекс потока на арене, которая в данный момент используется вызывающим потоком, или `task_arena::not_initialized`, если поток еще не инициализирован планировщиком задач. Индекс потока – целое число от 0 до уровня конкурентности арены. Индексы назначаются как потокам, созданным приложением (мастер-потокам), так и рабочим потокам при заходе на арену и сохраняются до выхода с арены. Индексы потоков на одной арене уникальны (т. е. никакие два потока на одной и той же арене не могут одновременно иметь одинаковые индексы), но необязательно должны быть последовательными числами. Поскольку поток может покинуть арену в любой момент времени, если он не занят исполнением задачи, то индекс потока может быть разным в двух задачах, пусть даже они принадлежат одной и той же группе задач или алгоритму. Потоки на разных аренах могут иметь одинаковые индексы. Присоединение к вложенной арене в функции `execute()` может изменить текущее значение индекса, но при возврате на внешнюю арену будет восстановлено прежнее значение. Этот метод используется, например, в ознакомительном расширении `task_scheduler_observer` для привязки потоков, заходящих на некоторую арену, к конкретному оборудованию
`int max_cocurrency()`	Возвращает уровень конкурентности арены задач, которая в данный момент используется вызывающим потоком. Если поток еще не инициализирован планировщиком задач, то возвращает уровень конкурентности, автоматически определяемый для текущей конфигурации оборудования

`template<F> auto isolate(F&) -> decltype(f())`
 	
`template<F> auto isolate(const F&) -> decltype(f())`
	Выполняет заданный функтор в изоляции, разрешив вызывающему потоку обрабатывать только задачи, запланированные в области видимости функтора (называемой также изолированным регионом). Функция возвращает значение, возвращенное функтором. Объект, возвращенный функтором, не может быть ссылкой, вместо этого можно использовать класс `std::reference_wrapper`. В изолированном регионе такие асинхронные параллельные конструкции, как граф потоков или группа потоков, следует использовать с осторожностью.
	Если `graph::wait_for_all` или `task_group::wait` выполняется в изолированном регионе, то задачи, запланированные в результате вызова метода `try_put()` потокового графа или метода `task_group::run()`, доступны, только если запланированы в том же самом изолированном регионе или из задачи, ранее запущенной в этом регионе. В противном случае возможно снижение производительности и даже взаимоблокировка

Рис. B.75 ❖ Планировщик задач: члены пространства имен `this_task_arena`

Класс задачи (task)

#include <tbb/task.h>

Класс task является базовым для всех классов задач.
Его подкласс empty_task представляет задачу, которая не делает ничего. Он полезен в качестве продолжения родительской задачи, когда продолжение должно только дождаться завершения предшествующих задач.

Атрибуты задачи

У каждого экземпляра задачи имеются атрибуты, которые не видны напрямую, но важны для понимания того, как используются объекты задач. Вообще говоря, методы выделения должны вызываться до запуска любой из выделенных задач. Исключение составляет метод allocate_additional_child_of(t), который можно вызывать, даже если задача t уже выполняется.

Атрибут	Описание
successor	Либо null, либо указатель на следующую задачу, чье поле refcount будет уменьшено на единицу после завершения данной задачи. Как правило, successor указывает на задачу, которая выделила данную, или на задачу, выделенную как продолжение данной. В методах класса task задача **successor** называется родительской (parent), а предшествующая ей – дочерней (child), потому что такое использование наиболее типично. Но по мере развития библиотеки наличие отношения ребенок–родитель между предшествующей и последующей задачами стало необязательным.
refcount	Количество задач, для которых данная является родителем. Увеличение и уменьшение refcount производятся НЕ атомарно. На заре развития TBB это, быть может, и не было проблемой, но теперь стало. В современной TBB все функции, кроме allocate_additional_child_of, сделаны более эффективными благодаря предположению о том, что они будут использоваться только для настройки, предшествующей запуску. Поскольку это нормальный способ использования, мы выигрываем в эффективности, и при этом все работает правильно, коль скоро придерживаться правила «сначала выдели, потом запускай». Когда задача уже работает, дополнительных потомков следует создавать с помощью allocate_additional_child_of

Выделение задач

Выделять память для объектов задач следует только с помощью одного из перечисленных ниже перегруженных операторов new. Методы allocate_x не конструируют задачу, а возвращают прокси-объект, который можно передать в качестве аргумента одному из перегруженных вариантов оператора new, предоставляемых библиотекой.

Член	Описание
new(task::**allocate_root**(task_group_context& group)) T	Выделяет задачу типа T с заданной группой для отмены. Для выполнения задачи использовать метод spawn_root_and_wait
new(task::**allocate_root**()) T	Работает как task::**allocate_root**(task_group_context&)), но в качестве группы для отмены используется самая внутренняя текущая группа
new(x.**allocate_continuation**()) T	Выделяет и конструирует задачу типа T и передает преемника successor от x новой задаче. Никакие счетчики ссылок не изменяются
new(x.**allocate_child**()) T	Выделяет задачу, для которой преемником successor является this. Если используется явная передача управления через продолжение, то вызывать этот метод должно продолжение, а не преемник, только тогда преемник будет установлен правильно. Если количество задач может быть достаточно велико и не фиксировано, то рассмотрите возможность сначала создать список task_list предшественников и запускать одним вызовом task::spawn. Если задача должна запускать некоторых предшественников, до того как все сконструированы, то она должна пользоваться методом task::allocate_additional_child_of (*this), поскольку этот метод атомарно увеличивает refcount, так что дополнительный предшественник будет правильно учтен. Но в таком случае следует защититься от преждевременного обнуления refcount, используя паттерн блокировки задачи

Рис. В.76 ❖ Планировщик задач: класс task (*начало*)

`new( task::task::`**`allocate_additional_child_of`**`(y) ) T`	Выделяет задачу как предшественника другой задачи `y`. Задача `y` должна уже работать сама или иметь других работающих предшественников. Поскольку у `y` могут быть работающие предшественники, увеличение `y.refcount` производится атомарно (в отличие от других методов выделения). Если программист добавляет предшественника к задаче с другими работающими предшественниками, то должен позаботиться о том, чтобы `refcount` преемника не оказался преждевременно равен 0, в результате чего преемник начнет выполняться, прежде чем будет добавлен предшественник

Уничтожение задачи (явное)

Обычно задача уничтожается автоматически планировщиком после возврата из метода `execute`. Но иногда объект задачи используется идиоматически (например, для подсчета ссылок) вообще без вызова `execute`. Такие задачи следует уничтожать методом `destroy`.

Член	Описание
`static void `**`destroy`**`( task& victim )`	Вызывает деструктор и освобождает память, занятую задачей `victim`. Если `victim.parent` не равно `null`, то атомарно уменьшает `victim.parent->refcount`. Родитель не помещается в пул готовых, если его счетчик ссылок обратился в ноль.

Рециклинг задач

Бывает так, что более эффективно повторно использовать объект задачи, чем выделять память для новой. Зачастую родитель становится продолжением или одним из предшественников. Правило совмещения: рециклированная задача `t` не должна подвергаться риску повторного выполнения `t.execute()`, когда ранее вызванный `t.execute()` еще работает. Отладочная версия библиотеки обнаруживает некоторые нарушения этого правила. Методы `recycle_x` должны вызываться только тогда, когда метод `execute()` еще работает.

Член	Описание
`void `**`recycle_as_continuation`**`()`	Счетчик ссылок рециклированной задачи должен быть установлен в `n`, где `n` – количество предшественников задачи-продолжения. Вызывающая сторона должна гарантировать, что `refcount` задачи не обращается в 0 раньше, чем вернет управление метод `execute()`, в противном случае будет нарушено правило совмещения. Если дать такую гарантию невозможно, то используйте метод `recycle_as_safe_continuation()` и устанавливайте `refcount` равным `n+1`. Для задачи `t` возможно возникновение гонки, если: – `t.execute()` рециклирует `t` как продолжение; – продолжение имеет предшественников, но все они завершаются до возврата из `t.execute()`. В таком случае рециклированная `t` будет неявно перезапущена, когда исходный вызов `t.execute()` еще работает, чем нарушит правило совмещения. В ситуациях, когда используется метод `recycle_as_continuation()`, этой гонки обычно удается избежать, сделав так, чтобы `t.execute()` возвращал указатель на одного из предшественников вместо явного запуска этого предшественника. Планировщик неявно запускает этого предшественника после возврата из `t.execute()`, гарантируя тем самым, что рециклированная `t` не будет перезапущена преждевременно. Последствия: `this` не уничтожается после возврата из метода `execute()`

Рис. B.76 (*продолжение*)

`void` **`recycle_as_safe_continuation`**`()`	Счетчик ссылок для рециклированной задачи должен быть установлен равным `n+1`, где `n` – количество предшественников задачи-продолжения. Дополнительная +1 учитывает задачу, подлежащую рециклингу. Последствия: `this` не уничтожается после возврата из метода `execute()`. Этот метод позволяет избежать гонки, возможной для метода `recycle_as_continuation()`, поскольку дополнительная +1 не дает задаче-продолжению начать выполнение до того, как завершится исходный вызов `execute()`
`void` **`recycle_as_child_of`**`(` `task& new_successor)`	Последствия: `this` становится предшественником `new_successor` и не уничтожается после возврата из метода `execute()`

Синхронизация задач

Запуск задачи `t` приводит либо к тому, что вызывающий поток вызывает `t.execute()`, либо к помещению `t` в пул готовых. Любой поток, участвующий в планировании задач, может затем забрать себе задачу и вызвать `t.execute()`.
Некоторые методы различают запуск корневых и некорневых задач. Корневой называется задача, созданная методом `allocate_root`.
Примечание: задача не должна запускать предшествующую задачу до тех пор, пока не вызовет метод `set_ref_count`, чтобы указать количество предшественников и обозначить свое желание использовать один из методов `wait_for_all`.

Член	**Описание**
`void` **`set_ref_count`**`( int count )`	Требования: `count >= 0`. Если мы намереваемся впоследствии запустить *n* предшественников и ждать, то *count* должно быть равно *n*+1. В противном случае *count* должно быть равно *n*. Последствия: устанавливает атрибут `refcount` равным `count`
`void` **`add_ref_count`**`( int count )`	Атомарно прибавляет `count` к атрибуту `refcount`. Возвращает новое значение атрибута `refcount`
`void` **`increment_ref_count`**`()`	Атомарно увеличивает атрибут `refcount` на 1. Возвращает новое значение атрибута `refcount`
`void` **`decrement_ref_count`**`()`	Атомарно уменьшает атрибут `refcount` на 1. Возвращает новое значение атрибута `refcount`
`void` **`wait_for_all`**`()`	Выполняет задачи, находящиеся в пуле готовых, пока `refcount` не станет равен 1. Затем оставляет `refcount=1`, в объекте `task_group_context`, ассоциированном с задачей, задан режим `concurrent_wait`, иначе устанавливает `refcount` равным 0. Кроме того, `wait_for_all()` автоматически сбрасывает состояние отмены в контексте `task_group_context`, неявно ассоциированном с задачей, если выполнены все перечисленные ниже условия: • при выделении задачи контекст не задавался; • вызывающий поток создан пользователем, а не является рабочим потоком TBB; • это самый внешний из всех вызовов `wait_for_all()`, произведенных данным потоком
`static void` **`spawn`**`(task& t)`	Помещает задачу `t` в пул готовых и сразу же возвращает управление. Если преемник `t` не равен `null`, то перед запуском дочерних задач для преемника следует вызвать метод `set_ref_count`, поскольку если задача начала выполняться, то по ее завершении `refcount` преемника автоматически будет уменьшен на 1 асинхронно. Отладочная версия библиотеки часто обнаруживает, что `set_ref_count` не был вызван вообще или он был вызван слишком поздно

Рис. В.76 (*продолжение*)

`static void spawn(task_list& list)`	Эквивалентно запуску каждой задачи в списке `list` с последующей очисткой списка, но может оказаться эффективнее. Если список `list` пуст, то ничего не делает
`void spawn_and_wait_for_all( task& t)`	Аналогично `{ spawn(t); wait_for_all(); }`, но часто эффективнее. Кроме того, гарантирует, что задача выполняется в текущем потоке. Иногда это ограничение упрощает синхронизацию
`void spawn_and_wait_for_all( task_list& list)`	Аналогично `{ spawn(list); wait_for_all(); }`, но часто эффективнее
`static void spawn_root_and_wait( task& root)`	Устанавливает атрибут `parent` задачи `root` в неопределенное значение и выполняет задачу `root`. Затем уничтожает `root`, если она не была рециклирована
`static void spawn_root_and_wait( task_list& root_list)`	Для каждой задачи `t` в списке `root_list` выполняет метод `spawn_root_and_wait()`, потенциально параллельно
`static void enqueue( task& )`	Задача планируется и будет в конечном итоге выполнена рабочим потоком, даже если никакой поток явно не ждет ее завершения.Если общее количество рабочих потоков рвано 0, то создается специальный дополнительный рабочий поток для выполнения поставленных в очередь задач. Поставленные в очередь задачи выполняются приблизительно (но не точно) в порядке «первым пришел – первым обслужен». Постановка задач в очередь в случае рекурсивного параллелизма может привести к высокому потреблению памяти, потому что рекурсия распространяется в ширину. Для рекурсивного параллелизма пользуйтесь обычным запуском задач (`spawn`). Следует избегать явного ожидания поставленной в очередь задачи, т. к. не исключено, что сначала нужно будет обработать задачи, поставленные в очередь в других частях программы, не связанных с данной. Рекомендуется, чтобы поставленная в очередь задача асинхронно сигнализировала о своем завершении, например отправив сообщение потоку, поставившему ее в очередь
Контекст задачи	
Методы, описанные ниже, раскрывают связи между разными объектами задач, а также между задачами и физическими потоками.	
Член	**Описание**
`static task& self()`	Возвращает ссылку на самую глубоко вложенную задачу из числа выполняемых вызывающим потоком. Задача считается выполняемой, если работает ее метод `execute()`, `note_affinity()` или деструктор. Если вызывающий поток создан пользователем и не выполняет никакую задачу, то `self()` возвращает ссылку на неявную фиктивную задачу, ассоциированную с потоком
`task* parent() const`	Возвращает значение атрибута `successor`. Результат не определен, если задача была выделена функцией `allocate_root` и в данный момент работает под контролем `spawn_root_and_wait`
`void set_parent(task* p)`	Требования: обе задачи должны принадлежать одной группе, за исключением случая `p == NULL`. Например, для задачи `t` должно быть `t.group() == p->group()`. Последствия: устанавливает указатель на родительскую задачу равным `p`

Рис. В.76 (*продолжение*)

`bool is_stolen_task() const`	Возвращает `true`, если задача работает в потоке, отличном от того, который ее запустил. *Примечание*: задачи, поставленные в очередь методом `task::enqueue()`, никогда не считаются заимствованными
`task_group_context* group()`	Возвращает дескриптор группы задач, которой принадлежит данная
`void change_group( task_group_context& ctx )`	Перемещает задачу из текущей группы в ту, что задана аргументом `ctx`

Отмена задач

Задачей называется работа, которая может быть отменена или выполнена до конца. Отмененная задача не выполняет свой метод `execute()`, если он еще не начался. В противном случае отмена не оказывает непосредственного воздействия на задачу. Но задача может вызвать функции `task::is_cancelled()` и узнать, была ли запрошена отмена, после того как началось ее выполнение.

Член	Описание
`bool cancel_group_execution()`	Запрашивает отмену всех задач в своей группе и подчиненных ей. Возвращает `false`, если группа данной задачи уже получала запрос об отмене, иначе `true`
`bool is_cancelled() const`	Возвращает `true`, если группа данной задачи уже получила запрос об отмене, иначе `false`

Приоритеты задач

Приоритет может быть задан для отдельных задач или для групп задач. Библиотека поддерживает три уровня приоритета (низкий, обычный, высокий) и два вида приоритетов: (1) статический для задач, поставленных в очередь; (2) динамический для групп задач.
Приоритет первого вида задается с помощью необязательного аргумента метода `task::enqueue()`, распространяется только на одну задачу и не может быть изменен впоследствии. Задачи с более высоким приоритетом выбираются из очереди раньше, чем задачи с низким приоритетом. Приоритет второго вида применяется ко всем задачам в группе и может быть изменен в любой момент с помощью ассоциированного объекта `task_group_context` или через любую задачу группы. Относящиеся к приоритетам методы в классе `task_group_context` описаны в соответствующем разделе. Планировщик задач отслеживает самый высокий приоритет готовых к выполнению задач (запущенных и поставленных в очередь) и откладывает выполнение задач с низким приоритетом до завершения всех задач с более высоким приоритетом. По умолчанию всем задачам и группам задач назначается обычный приоритет.

Член	Описание
`void enqueue( task& t, priority_t p) const`	Ставит задачу `t` в очередь с приоритетом `p`. *Примечание*: приоритет поставленной в очередь задачи не влияет на приоритет группы задач, из области видимости которой вызван метод `task::enqueue()`, т. е. той группы, которой принадлежит задача, возвращаемая методом `task::self()`
`void set_priority( priority_t p )`	Изменяет приоритет группы задач, которой принадлежит данная задача
`priority_t p group_priority() const`	Возвращает приоритет группы задач, которой принадлежит данная задача

Привязка задач

Эти методы оптимизируют привязку задач к кешу. Они сообщают библиотеке, что последующую задачу было бы желательно выполнять в том же потоке, что другую задачу, выполненную раньше.

Член	Описание
`affinity_id`	Тип `task::affinity_id` – определенный библиотекой целочисленный тип без знака. Значение 0 означает отсутствие привязки. Другие значения представляют привязку к определенному потоку. Не следует делать никаких предположений о семантике ненулевых значений. Соответствие между ненулевыми значениями и потоками – внутреннее дело библиотеки TBB

Рис. B.76 (*продолжение*)

`virtual void note_affinity( affinity_id id )`	Планировщик задач вызывает метод `note_affinity` перед вызовом `execute()`, если: • у задачи нет привязки, но она будет выполнена в потоке, отличном от того, который ее запустил; • у задачи есть привязка, но она будет выполнена в потоке, отличном от того, который указан в привязке. Этот метод можно переопределить, так чтобы `id` запоминался и в дальнейшем использовался в качестве аргумента метода `set_affinity(id)` для последующей задачи
`void set_affinity( affinity_id id )`	Устанавливает привязку данной задачи к указанному `id`. Значение `id` должно быть равно 0 или получено от метода `note_affinity`
`affinity_id affinity() const`	Возвращает привязку данной задачи, установленную методом `set_affinity`

Список задач (класс task_list)

Список объектов `task`.

Член	Описание
`task_list``()`	Конструирует пустой список
`~task_list``()`	Уничтожает список. Не уничтожает сами объекты задач
`bool` **`empty`**`() const`	Возвращает `true`, если список пуст, иначе `false`
`push_back``( task& task )`	Вставляет ссылку на `task` в конец списка
`task& task` **`pop_front`**`()`	Извлекает ссылку на `task` из начала списка. Возвращает извлеченную ссылку
`void` **`clear`**`()`	Удаляет все ссылки на задачи из списка. Не уничтожает сами объекты задач

Отладка задач

Описанные ниже методы полезны для отладки. В будущих реализациях могут измениться.

Член		Описание
`state_type state() const`		Текущее состояние задачи. В таблице ниже перечислены возможные значения. Любое другое значение свидетельствует о повреждении памяти, например об использовании задачи, память которой уже освобождена
	Значение	**Описание**
	`allocated`	Задача только что выделена или рециклирована
	`ready`	Задача находится в пуле готовых и будет уничтожена после возврата из метода `execute()`
	`executing`	Задача работает и будет уничтожена после возврата из метода `execute()`
	`freed`	Задача находится во внутреннем списке свободных или в процессе перемещения в него или из него
	`reexecute`	Задача работает и будет повторно запущена после возврата из метода `execute()`
`int ref_count() const`		Значение атрибута `refcount`

Рис. В.76 (*продолжение*)

Настройки плавающей точки

Для приложений, которым необходимо управлять зависящими от процессора настройками вычислений с плавающей точкой, предлагается два способа распространить нужные настройки на задачи, выполняемые планировщиком TBB:

- на этапе инициализации арены задач или планировщика задач для данного потока приложения запоминаются текущие настройки плавающей точки, действующие в потоке;
- в классе `task_group_context` имеется метод для запоминания текущих настроек плавающей точки.

По умолчанию рабочие потоки пользуются настройками плавающей точки, запомненными на этапе инициализации явной или неявной арены задач, связанной с потоком приложения. Эти настройки применяются ко всем параллельным вычислениям, выполняемым на этой арене или начатым потоком приложения, – до тех пор, пока поток не уничтожит свой экземпляр планировщика задач. Если впоследствии поток повторно инициализирует планировщик задач, то будут запомнены новые настройки.

Для более точного контроля над поведением плавающей точки поток может запомнить текущие настройки в контексте группы задач. Это можно сделать в момент создания контекста, передав специальный флаг конструктору:

```
task_group_context ctx(
  task_group_context::isolated,
  task_group_context::default_traits |
  task_group_context::fp_settings );
```

или обратившись к методу `capture_fp_settings`:

```
task_group_context ctx;
ctx.capture_fp_settings();
```

Затем контекст группы задач можно передать большинству параллельных TBB-алгоритмов (включая `tbb::flow::graph`), тогда все относящиеся к алгоритму задачи будут пользоваться указанными в нем настройками плавающей точки. Допускается выполнение параллельных алгоритмов с различными настройками плавающей точки, запомненными в разных контекстах, – и даже в одно и то же время. Настройки, запомненные в контексте группы задач, имеют приоритет над настройками, запомненными в момент инициализации планировщика. Таким образом, если параллельному алгоритму передан контекст, то будут использованы настройки плавающей точки, запомненные в этом контексте. Если же в контексте настройки не были запомнены или контекст не указан явно, то используются настройки, заданные при инициализации планировщика.

Во вложенном вызове параллельного алгоритма без указания контекста, в котором явно запомнены настройки плавающей точки, используются настройки из внешнего уровня. Если ни в одном из внешних контекстов настройки плавающей точки не были запомнены, то используются настройки, заданные при инициализации планировщика.

Гарантируется, что:

- настройки плавающей точки, запомненные в контексте группы задач или на этапе инициализации планировщика, применяются ко всем задачам, выполняемым планировщиком задач;
- вызов параллельного TBB-алгоритма не оказывает видимого влияния на настройки плавающей точки вызывающего потока, даже если алгоритм выполняется с другими настройками.

Эти гарантии действуют *только* при следующих условиях:

- пользовательский код внутри задачи либо не изменяет настройки плавающей точки, либо внесенные модификации отменяются путем восстановления предыдущих настроек до завершения задачи;
- в наблюдателях за планировщиком задач настройки плавающей точки не модифицируются.

В противном случае указанные гарантии не действуют и поведение, относящееся к настройкам плавающей точки, не определено.

Исключения

Обработка исключений рассмотрена в главе 15; здесь же приводится краткий обзор исключений, которые могут возбуждать компоненты TBB, и подкласса `std::exception`, называемого `tbb::tbb_exception`, который в TBB используется для распространения исключений между задачами с целью сделать обработку исключений в программе, где используется TBB, естественной.

TBB распространяет исключения вдоль логических путей в дереве задач. Поскольку эти пути пересекают стеки задач, необходимо каким-то образом обеспечить перемещение исключений из одного стека в другой.

Если исключение выходит за пределы задачи, его перехватывает исполняющая среда TBB и обрабатывает следующим образом:

1) если группа отмены для данной задачи уже получила запрос об отмене, то исключение игнорируется;
2) в противном случае исключение запоминается и группа отменяется;
3) запомненное исключение повторно возбуждается в корне группы отмены, после того как все задачи группы завершены или успешно отменены.

В современных версиях TBB (собранных компилятором, поддерживающим C++11) запоминается точная копия исключения. Если же компилятор не поддерживает C++11, то в TBB используется механизм обратной совместимости – класс `tbb::captured_exception`, аппроксимирующий исходное исключение. Этот класс уже уходит в прошлое и потому в книге не рассматривается.

Исключения, возбуждаемые компонентами TBB **#include <tbb/**tbb_exception.h>	
Имя исключения	**Возбуждается, когда...**
bad_last_alloc	1. Операция pop объекта concurrent_queue или concurrent_bounded_queue соответствует операции push, возбудившей исключение. 2. Операция над concurrent_vector не может быть выполнена, потому что предыдущая операция возбудила исключение
improper_lock	Поток пытается захватить уже захваченную критическую секцию или reader_writer_lock
invalid_multiple_scheduling	task_group или structured_task_group пытается повторно запустить task_handle
missing_wait	task_group или structured_task_group уничтожена раньше, чем вызван метод wait()
user_abort	Операция push или pop объекта concurrent_bounded_queue была прервана пользователем

Рис. В.77 ❖ Исключения, возбуждаемые TBB

Пример обработки исключения

```
#include <tbb/tbb_exception.h>
#include <tbb/parallel_for.h>
#include <vector>
#include <iostream>

std::vector<int> Data;

struct Update {
  void operator()( const tbb::blocked_range<int>& r ) const {
  for( int i=r.begin(); i!=r.end(); ++i )
     Data.at(i) += 1;
  }
};

int main( int argc, char *argv[] ) {
  int vecsize = 1000;
  int vecwalk = vecsize;
  Data.resize(vecsize);
  printf("Vector of size %d being traversed up to element %d\n",
     vecsize,vecwalk);
  try {
    tbb::parallel_for( tbb::blocked_range<int>(0, vecwalk), Update());
  } catch( std::out_of_range& ex ) {
    std::cout << "Exception caught was out_of_range: " <<
      ex.what() << std::endl;
    exit(0);
  }
  printf("No exception detected.\n");
  return 0;
}
```

```
$ ./example 1000
Vector of size 1000 being traversed up to element 1000
No exception detected
$ ./example 1001
Vector of size 1000 being traversed up to element 1001
Exception caught was out_of_range: vector
```

Рис. В.78 ❖ Исключения: пример обработки исключения

Класс tbb_exception и movable_exception

#include <tbb/tbb_exception.h>

В параллельной среде исключения иногда необходимо распространять между потоками. Класс tbb::tbb_exception, наследующий std::exception, предназначен именно для этой цели. Класс tbb::movable_exception предлагает удобный способ реализовать подкласс tbb_exception, который распространяет произвольные данные, допускающие конструирование копированием.

```
namespace tbb {
  class tbb_exception: public std::exception {
    virtual tbb_exception* move() = 0;
    virtual void destroy() throw() = 0;
    virtual void throw_self() = 0;
    virtual const char* name() throw() = 0;
    virtual const char* what() throw() = 0;
  };
}
```

В производных классах абстрактные виртуальные методы должны быть определены следующим образом:

- **move()** должен создать указатель на копию исключения, которая будет существовать дольше оригинала. Он может переместить содержимое оригинала;
- **destroy()** должен уничтожить копию, созданную move();
- **throw_self()** должен возбудить *this;
- **name()** обычно возвращает RTTI-имя первоначально перехваченного исключения;
- **what()** возвращает завершающуюся нулем строку, описывающую исключение.

Члены movable_exception	Описание
movable_exception(const ExceptionData& src)	Конструирует объект **movable_exception**, содержащий копию src
ExceptionData& data() **throw**()	Возвращает ссылку на внутренние данные
const ExceptionData& **data**() const throw()	Возвращает константную ссылку на внутренние данные

Рис. В.79 ❖ Исключения: классы tbb_exception и movable_exception

Потоки

TBB поддерживает переносимый API «потока», который на самом деле не что иное, как обертка вокруг платформенного потока. Этот API появился раньше, чем класс std::thread в C++, и не имеет никаких преимуществ по сравнению с последним.

В этой книге мы не раз говорили об опасностях перегруженности процессорных ядер, а в предисловии кратко упомянули о положительных сторонах осмотрительной перегруженности. Дело в том, что TBB предназначена для поддержки кода, ориентированного на интенсивные вычисления. Но стоит отметить, что конкурентность также можно эффективно использовать для маскировки задержки операций ввода/вывода. Задачи TBB – неподходящее место для размещения ввода/вывода, потому что TBB не вытесняет потоки, ожидающие завершения ввода/вывода, – эту функцию уже обеспечивают современные операционные системы. Поэтому в TBB добавлен переносимый API для создания дополнительных потоков, в которых вычислений немного и которые могут выиграть от перегруженности ядра. Разработчики TBB были среди тех, кто лоббировал включение этой возможности в C++, что и произошло в стандарте C++11.

Обсуждение вопроса об управлении количеством потоков, используемых TBB, см. в главе 11. А здесь мы сделаем несколько замечаний о реализации в TBB класса `std::thread` для поддержки унаследованных приложений. В новом коде мы рекомендуем использовать реализацию `std::thread` из стандарта C++11.

Различия между классом thread в C++11 и TBB

Оба класса называются `std::thread`. Какой именно используется, зависит от включенного заголовка.

std::thread в C++11	**Потоки TBB**
`#include <thread>`	`#include <tbb/compat/thread.h>`
`template<class Rep, class Period> std::this_thread::sleep_for( const std::chrono::duration <Rep, Period>& rel_time`	`std::this_thread::sleep_for( const tbb::ticj_count::interval_t& )`
`std::thread::id` может быть хеширован шаблонным классом `std::hash`	`std::thread::id` может быть хеширован шаблонными классами `tbb::tbb_hash_compare` и `tbb::tbb_hash`
ссылочные параметры `rvalue`	параметр изменен на простое значение или функция исключена – зависит от ситуации
конструктор `std::thread` принимает произвольное число аргументов	конструктор `std::thread` принимает от 0 до 3 аргументов
деструктор `std::thread` вызывает `terminate()`, если метод `joinable()` потока возвращает `true`	деструктор `std::thread` вызывает `detach()`, если метод `joinable()` потока возвращает `true`

Рис. В.80 ❖ Исключения: сравнение классов потока в C++11 и в TBB

Parallel STL

Parallel STL обсуждается в главе 4, а здесь приведены некоторые справочные сведения.

Поскольку Parallel STL лишь недавно вошла в состав C++17, во время написания книги реализации были сравнительно новые, как и оптимизированные версии. Компания Intel включила свою реализацию Parallel STL в состав Intel Parallel Studio XE и Intel System Studio, а также в двоичные дистрибутивы TBB.

Parallel STL, являющаяся частью TBB, уже поддерживает все средства, включенные в C++17, а также те, что, вероятно, войдут в C++2x (конкретно, неупорядоченная (unsequenced) политика выполнения, определенная в выпущенной комитетом версии 2 Технической спецификации параллелизма от февраля 2018 года). Библиотека Parallel STL предлагает эффективную поддержку как параллельного, так и векторизованного выполнения алгоритмов. Что касается последовательного выполнения, то она опирается на реализацию, уже имеющуюся в стандартной библиотеке C++. По мере того как содержимое стандарта C++2x постепенно приобретает все более четкие очертания, поддержка Parallel STL будет развиваться.

К моменту публикации этой книги Intel сопровождает открытый исходный код Parallel STL в виде проекта на github (https://github.com/intel/parallelstl), и ведется открытое обсуждение возможного включения Parallel STL в libstdc++ (gnu) и libc++ (LLVM).

Пока же для включения оптимизированной версии Parallel STL в свой код необходимо добавить директиву `#include <pstl/execution>` и некоторые из следующих директив в зависимости от типа алгоритма STL, который мы намереваемся использовать:

- **`#include`** `<pstl/algorithm>`;
- **`#include`** `<pstl/numeric>`;
- **`#include`** `<pstl/memory>`.

```
// оригинальный код
#include <algorithm>
using namespace std;
sort(v.begin(), v.end());

// для добавления политики параллельного выполнения 'par' (parallel)
// нужно всего лишь поступить следующим образом
#include <algorithm>
#include <pstl/execution>
#include <pstl/algorithm> // ради std::sort
using namespace pstl::execution; // или std::execution
sort(par, v.begin(), v.end());
```

Рис. B.81 ❖ Parallel STL: упрощенный фрагмент кода

Неупорядоченная политика (`unseq` и `par_unseq`) – довольно странное создание: она означает, что вызовы функций абсолютно *не упорядочены* друг относительно друга, а главное следствие отсюда заключается в том, что они *могут* перемежаться в одном потоке выполнения. Во всех остальных случаях стандарт C++ настаивает на неопределенной упорядоченности вызовов (не могут перемежаться в одном потоке). Поэтому код не должен выполнять никаких других небезопасных относительно векторизации операций. В общем случае любое действие в одном вызове, которое нуждается в синхронизации с другим вызовом, небезопасно относительно векторизации. Это относится, в частности, к операциям выделения и освобождения памяти и захвата мьютекса.

Политики выполнения в Parallel STL

Parallel STL расширяет стандартную библиотеку C++ (STL) путем добавления нового аргумента – политики выполнения, – который определяет степень многопоточности и векторизации каждого алгоритма.

Политика выполнения **Имя класса**	**Стандарт**	**Семантика**	**По-простому**	**Может использоваться в**
`seq`, `sequenced_policy`	C++17	Последовательное выполнение **обязательно**	последовательное (без параллелизма)	–
`unseq`, `unsequenced_policy`	Предложено включить в C++2x	Неупорядоченное выполнение SIMD-команд **разрешено**. В этой политике необходимо, чтобы все функции были SIMD-безопасны	векторизованное	Компиляторы для SIMD
`par`, `parallel_policy`	C++17	Параллельное выполнение несколькими потоками **разрешено**. В этой политике необходимо, чтобы все функции были безопасны относительно конкурентности	многопоточное	TBB для задач
`par_unseq`, `parallel_unsequenced_policy`	C++17	Комбинация unseq и par. Требования политики являются объединением требований к `unseq` и `par`	многопоточное и векторизованное	TBB для задач Компиляторы для SIMD
`vec` – многообещающее предложение, но на момент написания этой книги оно еще не поддерживалось в версии Parallel STL, поставляемой вместе с TBB, потому что это добавление активно обсуждается в комитете по стандартизации.				
`vec`, `vector_policy`	Предложено включить в C++2x	Похоже на unseq, но допускает опережающие зависимости (с некоторым ограниченным расстоянием)	векторизованное	Компиляторы для SIMD

Рис. B.82 ❖ Parallel STL: политики выполнения

Глоссарий

Atom расхваливают как настраиваемый текстовый редактор XXI века с открытым исходным кодом. Рафа говорит: «Я люблю и emacs, но теперь его место на моем Mac'е занял Atom». Сравним с редакторами vi и emacs.

EFLOPs (экзафлоп) = 10^{18} операций с плавающей точкой.

EFLOPs/c (экзафлопс) = 10^{18} операций с плавающей точкой в секунду.

emacs. Лучший текстовый редактор в мире (если верить Джеймсу) и его открытый исходный код. Сравним с редактором vi. «emacs» – первый пакет, который Джеймс устанавливает на любой компьютер, с которым работает.

FLOPs (флопс). Количество операций с плавающей точкой.

FLOPs/c (флопс). Количество операций с плавающей точкой в секунду.

GFLOPs (гигафлоп) = 10^{9} операций с плавающей точкой.

GFLOPs/c (гигафлопс) = 10^{9} операций с плавающей точкой в секунду.

GPU (графический процессор). Вычислительное устройство, предназначенное для выполнения вычислений, связанных с графикой: освещения, преобразований, обрезки и рендеринга. Вычислительные возможности GPU первоначально проектировались исключительно для использования в «графическом конвейере», и сами GPU располагались между центральным процессором общего назначения (ЦП) и дисплеями. Поддержка программных средств вычислений без отправки результатов на дисплей и последующие расширения дизайна GPU привели к появлению у многих GPU более общих возможностей. OpenCL и CUDA – две популярные модели программирования, в которых используются вычислительные средства GPU. *См.* также **Гетерогенные платформы**.

OpenCL (Open Computing Language). Система для написания программ, исполняемых на гетерогенных платформах. OpenCL определяет API хоста для управления разгрузкой и присоединенными устройствами, а также расширения C/C++ для написания кода, работающего на присоединенном ускорителе (GPU, ЦСП, ППВМ и т. д.), но обладающего возможностью переходить на ЦП, если на этапе выполнения оказывается, что присоединенного устройства не существует или оно недоступно.

OpenMP. API, поддерживающий программирование на многоплатформенных многопроцессорных системах с разделяемой памятью на языках C, C++ и Fortran. Реализован для большинства процессорных архитектур и операционных систем. Включает набор директив компилятора, библиотечных процедур и переменных среды, влияющих на поведение во время выполнения. OpenMP координируется некоммерческим технологическим консорциумом OpenMP Architecture Review Board и совместно определяется группой основных производителей аппаратного и программного обеспечения компьютеров (http://openmp.org).

PFLOPs (петафлоп) = 10^{15} операций с плавающей точкой.

PFLOPs/c (петафлопс) = 10^{15} операций с плавающей точкой в секунду.

SIMD. Single-instruction-multiple-data (один поток команд, несколько потоков данных) – способность обрабатывать несколько порций данных (например, элементов массива) в одной операции. SIMD – одна из компьютерных архитектур в широко известной классификации Флинна, впервые предложенной в 1966 году.

SPMD. Single-program-multiple-data (одна программа, несколько потоков данных) – способность обрабатывать несколько порций данных (например, элементов массива) в одной и той же программе, в отличие от более ограничительной архитектуры SIMD. SPMD чаще всего ассоциируется с программированием передачи сообщений в компьютерных архитектурах с разделяемой памятью. SPMD – подкатегория MIMD (несколько потоков команд, несколько потоков данных) в широко известной классификации Флинна, впервые предложенной в 1966 году.

STL (Standard Template Library) – часть стандарта C++.

TBB. *См.* **Threading Building Blocks (TBB)**.

TFLOPs (терафлоп) = 10^{12} операций с плавающей точкой.

TFLOPs/c (терафлопс) = 10^{12} операций с плавающей точкой в секунду.

Threading Building Blocks (TBB). Самое популярное абстрактное решение для параллельного программирования на C++. TBB – проект с открытым исходным кодом, созданный компанией Intel и перенесенный на многие операционные системы и процессоры разных производителей. OpenMP и TBB редко конкурируют между собой на практике. TBB более популярна, чем OpenMP, если считать по количеству использующих разработчиков, но она популярна у программистов на C++, тогда как OpenMP используют в основном те, кто пишет на языках Fortran и C.

TLB. Аббревиатура Translation Lookaside Buffer (буфер ассоциативной трансляции). Это специализированный кеш, используемый для трансляции виртуальных адресов в физические. От количества элементов в TLB зависит, сколько страниц памяти можно одновременно использовать с удовлетворительной эффективностью. Доступ к странице, отсутствующей в TLB, называется непопаданием в TLB. Это приводит к срабатыванию ловушки в операционной системе и обновлению содержимого TLB.

vi. Текстовый редактор, входящий в состав большинства систем UNIX и BSD. Написан Биллом Джоем и популярен только среди тех, кто еще не открыл для себя emacs (если верить Джеймсу). И да, его исходный код открыт. Не выдерживает сравнения с emacs и Atom. Да, Рон, Джеймс проглядел книжку «vi in a nutshell», которую ты ему дал, но… по-прежнему настаивает на том, что работал с vi достаточно долго и заслужил право установить emacs.

Абстрагирование. В случае TBB под абстрагированием понимается разделение работы на две части: ту, что возлагается на программиста, и ту, что лучше оставить исполняющей среде. Цель такого абстрагирования – обеспечить масштабируемую высокую производительность в различных многоядерных и многократноядерных системах и даже на гетерогенных платформах без

переписывания кода. При таком разделении обязанностей за программистом остается выявление возможностей для распараллеливания, а на исполняющую среду возлагается отображение этих возможностей на оборудование. Код, написанный с соблюдением идей абстрагирования, не будет содержать параметров, задающих размеры кешей и количество ядер, и даже не будет заниматься согласованием производительности различных процессорных устройств.

Алгоритм – термин, который в TBB ассоциируется с общим допускающим повторное использование решением типичных проблем параллельного программирования. Поэтому в TBB и в этой книге применяется термин «параллельный алгоритм», хотя «параллельный паттерн» был бы уместнее.

Аппаратный поток. Аппаратная реализация задачи с отдельным потоком управления. Несколько аппаратных потоков можно реализовать с помощью нескольких ядер или исполнять одновременно в одном ядре, чтобы скрывать задержки, например включив режим гипертрединга в процессорном ядре. В последнем случае (гипертрединг или одновременная многопоточность, SMT) говорят, что физическое ядро включает несколько логических ядер (или аппаратных потоков).

Атомарной называется операция, которая гарантированно выглядит неделимой и не может быть прервана другими потоками. Например, процессор может предоставлять команду инкремента ячейки памяти. Для ее выполнения значение нужно прочитать из памяти, увеличить на единицу и записать обратно в память. Атомарная команда инкремента гарантирует, что конечное значение будет таким, как если бы между чтением и записью все прочие операции с этой ячейкой памяти были запрещены.

Балансировка нагрузки. Назначение ресурсов задачам в условиях неравного размера задач. Оптимально было бы загрузить работой все вычислительные устройства с минимальными потерями на накладные расходы.

Барьер. Если вычисление разбито на этапы, то часто бывает необходимо, чтобы все потоки завершили один этап, прежде чем хотя бы одному будет разрешено перейти к следующему. Барьер – механизм синхронизации, решающий эту задачу: потоки, подошедшие к барьеру, ждут прибытия последнего потока, а затем все сразу продолжают работу. Барьер можно реализовать с помощью атомарных операций. Например, каждый поток мог бы инкрементировать разделяемую переменную и перейти в состояние ожидания, если значение этой переменной не равно количеству потоков, синхронизируемых барьером. Последний прибывший поток сбрасывает барьер в ноль и пробуждает все заблокированные потоки.

Блок. Этот термин употребляется в двух смыслах: (1) состояние, в котором поток не может продолжить работу, потому что ждет некоторого события синхронизации (по-русски употребляется слово «блокировка», а не «блок»), и (2) область памяти. Второй смысл тесно связан с разбиением цикла на множество параллельных задач подходящей зернистости, каждая из которых работает со своим блоком памяти.

Блокировка. Механизм реализации взаимного исключения. Прежде чем войти в область взаимного исключения, поток должен попытаться захватить бло-

кировку на эту область. Если блокировка уже захвачена другим потоком, то текущий поток должен подождать (заблокироваться), либо приостановив работу, либо войдя в цикл активного ожидания. Когда блокировка освободится, текущий поток может захватить ее. Блокировки могут быть реализованы с помощью атомарных операций, которые сами по себе являются формой взаимного исключения простейших операций и реализованы аппаратно.

Векторизация. Преобразование кода, так чтобы можно было осуществлять одновременные вычисления на векторном оборудовании. Векторным оборудованием пользуются такие расширения системы команд в многопроцессорных системах, как MMX, SSE, AVX, AVX2 и AVX-512, но существует и векторное оборудование вне ЦП, и оно тоже может быть объектом векторизации. Векторизация программы обычно приводит к повышению производительности, потому что одной командой удается обработать больше данных.

Векторные операции. Низкоуровневые операции, которые могут воздействовать сразу на несколько элементов данных. *См*. **SIMD**.

Векторный параллелизм. Механизм реализации параллелизма аппаратными средствами с применением одного потока управления к нескольким элементам данных.

Взаимоблокировка. Программная ошибка. Возникает, когда по крайней мере две задачи ждут друг друга и ни одна не может продолжить работу, пока не продолжит другая. Это легко может случиться, если программа захватывает несколько мьютексов. Например, каждая задача может удерживать мьютекс, нужный другой задаче.

Взаимодействие. Любой обмен данными или синхронизация между программными задачами или потоками. Понимание того, что затраты на взаимодействие часто являются фактором, лимитирующим масштабируемость, критически важно для параллельного программирования.

Виртуальная память. Разрывает связь между адресами, используемыми программой, и физическими адресами в реальной памяти. Трансляция виртуальных адресов в физические происходит на уровне оборудования, которое инициализируется и управляется операционной системой.

Временная локальность. Близость, измеряемая в терминах времени. Ср. с **пространственной локальностью**. Под временной локальностью понимается поведение программы, при котором данные с большой вероятностью будут снова использованы в ближайшее время. Алгоритмы с хорошей временной локальностью данных могут получить выигрыш от кеширования данных, широко распространенного в современных компьютерах. Нередко бывает возможно добиться одновременно временной и пространственной локальности данных. Компьютерная система, обладающая таким свойством, скорее достигнет оптимальной производительности, отсюда и интерес к проектированию таких алгоритмов.

Высокопроизводительные вычисления (HPC). Вычисления с самой высокой производительностью на данном историческом отрезке времени. На сегодняшний день это как минимум петафлопс. Термин HPC иногда употребляется как синоним вычислений на суперкомпьютере, хотя суперкомпьютерами,

пожалуй, следует называть системы с еще большей производительностью. HPC постепенно проникает во все новые отрасли промышленности, но чаще всего ассоциируется с решением самых трудных научных и технических задач. Высокопроизводительные методы анализа данных, зачастую с использованием искусственного интеллекта (ММ) и машинного обучения (МО), рассматриваются как HPC в самом крупномасштабном воплощении и нередко сочетаются с традиционными рабочими HPC-нагрузками.

Гетерогенная платформа состоит из смеси разнородных вычислительных устройств, в отличие от однородной совокупности центральных процессоров. Гетерогенные вычисления обычно используются, чтобы добиться ускорения за счет присоединенных устройств: GPU, ЦСП, ППВМ и других. См. также `OpenCL`.

Гипертрединг. Многопоточность на одном процессорном ядре, имеющая цель полнее задействовать функциональные возможности ядра с изменением последовательности команд посредством выполнения большего числа команд в единицу времени, чем можно было бы выполнить в одном программном потоке. В режиме гипертрединга несколько аппаратных потоков могут работать в одном ядре и разделять ресурсы, но распараллеливание или конкурентность все равно дают некоторые преимущества. Обычно каждый гиперпоток имеет, по крайней мере, собственный набор регистров и счетчик команд, поэтому переключение между гиперпотоками обходится сравнительно дешево. Слово «гипертрединг» обычно ассоциируется с процессорами Intel, см. также **одновременная многопоточность**.

Детерминированный. Так называют алгоритм, поведение которого предсказуемо. При одинаковых данных на входе алгоритм всякий раз порождает одинаковые данные на выходе. Определение того, что такое «одинаковые», может быть важно из-за ограниченной точности математических операций и потому, что некоторые оптимизации, в т. ч. распараллеливание, могут изменять порядок операций. Это может приводить к разным ошибкам округления, когда порядок математических операций, необходимых для вычисления ответа, в первоначальной программе и ее конкурентной версии отличается. Конкурентность – не единственная причина **недетерминированности** алгоритмов, но на практике самая частая. Применение моделей программирования с последовательной семантикой и исключение гонок за данные с помощью соответствующих средств управления доступом, вообще говоря, устраняют все главные последствия конкурентности, кроме различия в ошибках округления.

Естественный параллелизм. Свойство алгоритма, который может быть разложен на большое число независимых задач почти или совсем без синхронизации либо взаимодействия.

Задача. Облегченная единица потенциального параллелизма с собственным потоком управления. Обычно реализуется на уровне пользователя, в противоположность управляемым ОС потокам. В отличие от потоков, задачи обычно последовательно выполняются на одном ядре от начала до конца. От «потока» задача отличается тем, что не делается никаких предположений о том, где она будет работать, тогда как между программными и аппаратными потоками существует взаимно однозначное соответствие. Потоки – это механизм параллельного выполнения задач, а задачи – единицы работы, которые просто

обеспечивают возможность параллельного выполнения; сами по себе задачи не являются механизмом параллельного выполнения.

Задержка – время, необходимое для завершения задачи, т. е. время между началом и окончанием задачи. Задержка измеряется в единицах времени. Ее масштаб может варьироваться от нескольких наносекунд до нескольких дней. В общем случае чем задержка меньше, тем лучше.

Закон Амдала. Ускорение ограничено нераспараллеливаемым последовательным участком программы. Программа, в которой две трети можно распараллелить, а одна треть остается последовательной, может быть ускорена не более, чем в три раза в предположении, что объем работы остается постоянным. Если при масштабировании проблемы нагрузка на параллельные части программы возрастает, то закон Амдала не так плох, как может показаться. См. **закон Густафсона–Барсиса**.

Закон Густафсона–Барсиса предлагает альтернативный взгляд на **закон Амдала**, который учитывает тот факт, что с ростом проблемы процентная доля последовательной части вычислений уменьшается.

Закон Мура. Наблюдение, согласно которому в истории развития полупроводниковых технологий количество транзисторов в плотной интегрированной схеме удваивалось примерно каждые два года.

Замощение. Разбиение цикла на множество параллельных задач подходящей зернистости. В общем случае замощение сводится к применению нескольких шагов к меньшей части проблемы, вместо того чтобы выполнять каждый шаг для всей проблемы по очереди. Цель замощения – увеличить степень повторного использования данных в кеше. Замощение может дать впечатляющее повышение производительности, когда вся проблема не помещается в кеш.

Зернистость. В таких терминах, как «крупнозернистый» и «мелкозернистый» параллелизм, означает, «сколько работы» может быть сделано до перехода к новой задаче или синхронизации. Программа лучше масштабируется, когда зернистость (степень детализации) как можно больше (так что потоки могут работать независимо), но при этом достаточно мала, чтобы полностью загрузить все вычислительные ресурсы (балансировка нагрузки). Эти два фактора конфликтуют между собой, поэтому приходится выбирать оптимальную степень детализации. TBB старается автоматизировать разбиение на задачи, но мир не совершенен, так что программист всегда может помочь в настройке для получения максимальной производительности с учетом знаний о конкретных алгоритмах.

Интерфейс передачи сообщений (MPI). Промышленный стандарт систем с передачей сообщений, спроектированный для обмена данными в разнообразных параллельных компьютерах.

Кеш – быстродействующая часть подсистемы памяти, в которой временно хранятся копии данных. При последующем обращении эти данные можно будет обработать быстрее, чем если бы пришлось выбирать данные из более далекой от процессора части памяти. Кеши обычно работают автоматически и предназначены для ускорения программ с временной и (или) пространственной локальностью. В современных компьютерах система кеширования обычно многоуровневая.

Кеш-дружелюбным называют приложение, производительность которого возрастает с увеличением размера задачи, но затем выходит на плато после достижения максимальной пропускной способности.

Кеш-недружелюбным называют приложение, в котором потребление памяти должно оптимально зависеть от рабочей нагрузки. В таком случае мы видим, что производительность остается постоянной или увеличивается до достижения оптимального размера рабочей нагрузки, а затем начинает падать. Для таких рабочих нагрузок, очевидно, существует «зона максимального благоприятствования».

Кеш-независимый алгоритм. Любой алгоритм, который хорошо работает без модификации на машинах с разной организацией памяти, в т. ч. с разным количеством уровней и разными размерами кеша. Слово «независимый» здесь означает, что такие алгоритмы ничего не знают о параметрах подсистемы памяти, таких как размеры или относительные скорости кешей. Это прямая противоположность прежним усилиям тщательно разрабатывать алгоритмы под конкретное оборудование кешей.

Кластер. Набор компьютеров с распределенной памятью, обменивающихся данными по высокоскоростным межсоединениям. Отдельные компьютеры часто называют **узлами**. TBB используется на уровне узла кластера, хотя несколько узлов часто программируют с применением TBB, а затем соединяют (обычно с помощью MPI).

Компонуемость. Способность использовать два компонента сообща, не вызывая отказов или необоснованных конфликтов (а лучше вообще без конфликтов). Если ограничения на компонуемость существуют, то оптимально было бы выявлять их на этапе сборки, без добавочного тестирования. Проблемы компонуемости, проявляющиеся только на этапе выполнения, – самая серьезная неприятность в некомпонуемых системах. Может относиться к системным средствам, к моделям программирования и к программным компонентам.

Конкурентность означает, что какие-то действия происходят логически одновременно. Две задачи, логически активные в один и тот же момент времени, считаются конкурентными. Между конкурентностью и **параллельностью** есть различия.

Ложное разделение. Две разные задачи, исполняемые двумя разными ядрами, могут писать в разные ячейки памяти, но если эти ячейки случайно оказались в одной строке кеша, то оборудование когерентности кешей попытается сделать эти строки когерентными, что ведет к лишнему взаимодействию между процессорами и снижению производительности, хотя на самом деле у задач нет никаких общих данных.

Локальность. Стремление использовать близкие, а не отдаленные ячейки памяти. При этом достигается максимальная эффективность строк кеша, страниц памяти и т. д. вследствие их повторного использования. Обеспечение высокой локальности ссылок – ключ к масштабируемости.

Масштабируемость. Мера увеличения производительности как функция от количества оборудования, которую можно использовать параллельно.

Масштабируемый. Приложение называется масштабируемым, если его производительность возрастает по мере добавления параллельных аппаратных ресурсов. Термином **сильная масштабируемость** обозначают масштабируемость, которая имеет место, даже когда размер проблемы не изменяется вместе с добавлением ресурсов. **Слабая масштабируемость** требует масштабирования проблемы при добавлении вычислительных ресурсов. См. **масштабируемость**.

Мегагерцевая эра. Исторический период, в течение которого тактовая частота процессоров и количество транзисторов в схеме удваивались примерно с одинаковой скоростью – приблизительно каждые два года. Такой быстрый прирост тактовой частоты прекратился в 2004 году, немного не достигнув 4 ГГц. После этого проектировщики стали направлять усилия на увеличение количества ядер, что знаменовало начало **многоядерной эры**.

Многократноядерный процессор. Многоядерный процессор, в котором число ядер настолько велико, что на практике даже не указывается – говорят просто «уйма». Этот термин употребляется в основном для процессоров с 32 и более ядрами, но точного определения не существует.

Многоядерная эра. Период времени, когда в проектировании процессоров произошел сдвиг от быстрого повышения тактовой частоты к увеличению количества ядер. Эта эра началась примерно в 2005 году.

Многоядерный. Так называют процессор с несколькими подпроцессорами (ядрами), каждый из которых поддерживает по меньшей мере один аппаратный поток.

Недерминированный. Не демонстрирующий детерминированного поведения, вследствие чего могут получаться разные результаты при разных прогонах алгоритма. Конкурентность – не единственная причина недетерминированности алгоритмов, но на практике самая частая. Дополнительные пояснения см. в статье «**Детерминированный**».

Неравномерный доступ к памяти (Non-Uniform Memory Access – NUMA). Относится к характеристикам подсистемы памяти в архитектуре с распределенной памятью. NUMA = задержка доступа к памяти различна для разных видов памяти. UMA = задержка доступа к памяти одинакова для всей памяти. См. главу 20.

Одновременная многозадачность. Многозадачность в пределах одного процессорного ядра. См. также **гипертрединг**.

Операции с плавающей точкой. В их число входят сложение, умножение, вычитание и другие операции, выполняемые над числами с плавающей точкой.

Отдаленная память. В системах с архитектурой NUMA это память, время доступа к которой больше, чем к ближней памяти. Какая память является ближней, а какая отдаленной, зависит от процесса, в котором выполняется код. В главе 20 мы также называем эту память нелокальной (в отличие от локальной).

Параллелизм данных. Подход к параллелизму, ориентированный на данные, а не на задачи. На практике успешные стратегии разработки параллельных алгоритмов основаны преимущественно на параллелизме данных, потому что декомпозиция данных (порождение задач для разных блоков данных) масшта-

бируется, а функциональная декомпозиция (порождение различных задач для разных функций) – нет. *См.* **Закон Амдала** и **Закон Густафсона–Барсиса**.

Параллелизм задач. Попытка классифицировать параллелизм как нечто, ориентированное больше на задачи, чем на данные. Мы сознательно избегаем употребления этого термина, поскольку его смысл изменчив. В частности, в других контекстах под «параллелизмом задач» могут пониматься задачи, порожденные функциональной декомпозицией, или нерегулярные задачи, порожденные все той же декомпозицией по данным. В этой книге любой параллелизм, порожденный декомпозицией по данным, регулярный или нет, считается параллелизмом данных.

Параллелизм потоков. Механизм реализации параллелизма аппаратными средствами, когда для каждой задачи создается отдельный поток управления.

Параллелизм. Выполнение нескольких действий одновременно. Существует много попыток классификации параллелизма.

Параллельный. Происходящий одновременно. Про две задачи, которые выполняют работу в один и тот же момент времени, говорят, что они работают параллельно. Проводя различия между конкурентностью и параллельностью, нужно прежде всего обращать внимание на то, может ли работа в принципе выполняться одновременно. Мультиплексирование одного процессорного ядра, десятилетиями практиковавшееся в многозадачных операционных системах, обеспечивало конкурентность, даже когда истинно одновременное выполнение было невозможно в силу наличия единственного процессора.

Паттерн. Общее, допускающее повторное использование решение типичной задачи. Исторически в ТВВ употребляется термин «параллельный алгоритм», хотя «параллельный паттерн» был бы уместнее.

Последовательный. Неконкурентный и непараллельный.

Поток. Может относиться к программному или аппаратному потоку. В общем случае «программный поток» – это любая единица параллельной работы с независимым потоком управления, а «аппаратный поток» – любое аппаратное устройство, способное выполнять один поток управления (в частности, устройство, поддерживающее один счетчик команд). Сравнивая «поток» с «задачей», важно иметь в виду, что в задаче не делается никаких предположений о том, где она будет работать, тогда как между программными и аппаратными потоками существует взаимно однозначное соответствие. Потоки – механизм реализации задач. Многозадачная или многопоточная операционная система мультиплексирует несколько программных потоков на один аппаратный благодаря чередованию выполнения, для чего ОС выделяет каждому потоку кванты времени. Многоядерный или многократноядерный процессор содержит несколько ядер, каждое из которых может выполнять по меньшей мере один независимый поток благодаря дублированию оборудования. Многопоточное или гиперпоточное процессорное ядро мультиплексирует одно ядро для выполнения нескольких программых потоков, применяя аппаратные механизмы чередования потоков.

Потоковый граф. Способ описания алгоритма с помощью нотации графов. Граф состоит из вычислительных узлов, соединенных ребрами, которые обозначают возможный поток управления.

Поточно-локальная память. Данные, память для которых специально выделена так, чтобы доступ к ней был только у одного потока, по крайней мере во время конкурентных вычислений. Цель состоит в том, чтобы избежать синхронизации в самые напряженные моменты вычислений. Классический пример поточно-локальной памяти – вычисление частичных сумм при сложении всех элементов большого массива. Для этого массив разбивается на участки, для каждого из которых вычисляется локальная частичная сумма (приватизированные переменные). В силу своей локальности эти переменные не требуют синхронизации.

ППВМ (программируемая пользователем вентильная матрица, англ. **FPGA).** Устройство, объединяющее много вентилей (а зачастую и конструкции более высокого уровня, например ЦСП, процессоры плавающей точки или сетевые контроллеры), которые не связаны между собой до тех пор, пока устройство не запрограммировано. Первоначально микросхема программировалась целиком и предполагалось, что это делается один раз при запуске системы, но современные ППВМ поддерживают частичное изменение конфигурации и зачастую допускают динамическую загрузку новых программ. Традиционно ППВМ рассматривались как способ объединить большое число отдельных микросхем в единую матрицу – обычно для экономии места на плате, энергопотребления и общей стоимости. Как следствие ППВМ программировались с помощью средств, похожих на те, что используются при проектировании схем на уровне платы или одного кристалла, – языков высокоуровневого описания (например, VHDL или Verilog). Но сравнительно недавно ППВМ стали использоваться как вычислительные движки в рамках модели программирования OpenCL.

Привязка. Совокупность методов, призванных ассоциировать конкретный программный поток с конкретным аппаратным потоком, обычно с целью добиться лучшей или более предсказуемой производительности. Спецификация привязки включает идеи максимальной удаленности, чтобы уменьшить состязание (разведение), или плотной упаковки (уплотнение), чтобы минимизировать расстояние, на котором происходит взаимодействие. OpenMP поддерживает развитый набор средств управления привязкой на разных уровнях – от абстрактного до полностью ручного. В языке Fortran 2008 средства управления не специфицированы, но Intel воспользовалась такими средствами из OpenMP для реализации конструкции «do concurrent». Библиотека TBB предоставляет абстрактную привязку цикла к циклу.

Придушенная масштабируемость (strangled scaling). Программная ошибка, в результате которой производительность параллельного кода страдает от высокого уровня состязания или накладных расходов, причем может даже стать хуже, чем у последовательного кода.

Программный поток. Виртуальный аппаратный поток. Иными словами, один поток выполнения в программе, который предполагается отобразить на аппаратный поток. Обычно операционная система допускает существование гораздо большего количества программных потоков, чем имеется аппаратных, и сопоставляет программному потоку аппаратный по мере необходимости. Ситуацию, когда программных потоков больше, чем аппаратных, называют **перегруженностью** (oversubscription).

Пропускная способность. Скорость завершения задач из заданного множества. Измеряет скорость вычислений и выражается в задачах в единицу времени. *См*. **Задержка**.

Пространственная локальность. Близость в смысле расстояния (разность между адресами в памяти). *Ср*. с **Временной локальностью**. Под пространственной локальностью понимают поведение программы, при котором использование одного элемента данных означает, что находящиеся рядом данные, зачастую следующий элемент, вскоре могут понадобиться. Алгоритмы с хорошей пространственной локальностью использования данных могут получить выигрыш от организации строк кеша и аппаратуры предвыборки; то и другое – стандартные компоненты современных компьютеров.

Прямая масштабируемость. Программа или алгоритм уже масштабированы с помощью потоков и (или) векторов, так что если степень параллелизма в будущем оборудовании возрастет, то нужно будет только перекомпилировать код новым компилятором или скомпоновать его с новой библиотекой. Использование правильных абстракций для выражения параллелизма обычно является ключом к прямой масштабируемости параллельной программы.

Равномерный доступ к памяти (Uniform Memory Access – UMA). Относится к характеристикам подсистемы памяти в архитектуре с распределенной памятью. UMA = задержка доступа к памяти одинакова для всей памяти. NUMA = задержка доступа к памяти различна для разных видов памяти. См. главу 20.

Разгрузка. Перенос части вычислений на присоединенное устройство, например ППВМ, GPU или иной ускоритель.

Разделяемая память. Когда две параллельно выполняемые единицы работы могут обращаться к данным по одному и тому же адресу. Обычно, чтобы такая операция была безопасной, необходима синхронизация. Таким способом могут разделять данные любые единицы работы: процессы, потоки, задачи, если только физическая организация подсистемы памяти это допускает. Однако процессы по умолчанию не разделяют память, для этого нужны специальные обращения к операционной системе.

Распараллеливание. Преобразование кода, так чтобы он мог одновременно выполнять несколько действий. В результате распараллеливания хотя бы некоторые части программы начинают работать параллельно.

Распределенная память. Память, физически размещенная в разных компьютерах. Для доступа к памяти на удаленном компьютере необходим непрямой интерфейс, например передача сообщений, тогда как к локальной памяти можно обращаться непосредственно. Распределенная память обычно поддерживается кластерами, которые мы, с этой точки зрения, рассматриваем как совокупность компьютеров. Поскольку память сопроцессоров обычно также недоступна главному процессору напрямую, ее можно, с точки зрения функциональности, тоже считать формой распределенной памяти.

Рекурсия. Повторный вход в функцию, когда другой экземпляр этой функции еще активен в том же потоке выполнения. В простейшем и самом распространенном случае функция напрямую вызывает сама себя, хотя в рекурсии могут участвовать и промежуточные функции. Для поддержки рекурсии состояние

частично выполненных функций хранится в динамически выделяемой памяти, например в стеке, хотя если поддерживаются функции высшего порядка, то может понадобиться более сложная схема выделения памяти. Ограничение уровня рекурсии может быть необходимо для предотвращения чрезмерного потребления памяти.

С заделом на будущее. Компьютерная программа, написанная так, что при изменении архитектуры компьютера она не утрачивает актуальность, но при этом не требуется вносить существенные изменения в код. Вообще говоря, чем более абстрактен метод программирования, тем больше шансов у программы сохранить актуальность в будущем. Низкоуровневые методы программирования, которые так или иначе отражают детали компьютерной архитектуры, вряд ли смогут пережить новации без изменений. Однако при написании абстрактных программ с заделом на будущее иногда приходится приносить в жертву эффективность.

Сериализация имеет место, когда задачи в потенциально параллельном алгоритме выполняются в определенном последовательном порядке, обычно из-за ограниченности ресурсов. Противоположность распараллеливанию.

Симметричный мультипроцессор (SMP). Многопроцессорная система с разделяемой памятью, работающая под управлением одной операционной системы.

Синхронизация. Координирование задач или потоков с целью получить желаемый порядок выполнения. Обычно используется, чтобы избежать нежелательных состояний гонки.

Сокрытие задержки. Планирование вычислений на процессорном ядре, в то время когда остальные задачи, использующие то же ядро, ждут завершения длительных операций, например обмена данными с памятью или диском. На самом деле задержка никуда не девается, потому что каждой отдельной задаче нужно фиксированное время для завершения, но зато в течение одного и того же времени можно выполнить больше задач, поскольку ресурсы разделяются более эффективно и пропускная способность возрастает.

Состояние гонки. Недетерминированное поведение в параллельной программе, обычно считающееся программной ошибкой. Гонка возникает, когда конкурентные задачи выполняют операции с одной и той же ячейкой памяти без надлежащей синхронизации, и одна из этих операций – запись. Программа, содержащая гонку, иногда работает правильно, а иногда ошибочно.

Строки кеша – единицы обмена данными между кешем и основной памятью. Для задействования пространственной локальности их длина обычно превышает одно слово. Общая тенденция – увеличивать длину строки кеша; как правило, она достаточна для хранения по меньшей мере двух чисел с плавающей точкой двойной точности, но редко превышает размер восьми таких чисел. Чем больше строка кеша, тем эффективнее обмен с основной памятью, но при этом возникают проблемы, например ложное разделение, которые в общем случае снижают производительность.

Узел (кластера). Компьютер с разделяемой памятью, часто на одной плате с несколькими процессорами, который соединен с другими узлами, так что в совокупности они образуют кластерный компьютер, или суперкомпьютер.

Ускорение. Отношение задержки при решении задачи на одном процессоре к задержке при решении той же задачи параллельно на нескольких процессорах.

Циклически порожденная зависимость возникает, когда некоторый элемент данных (например, элемент [3] массива) записывается на одной итерации цикла и читается на другой его итерации. Если циклически порожденных зависимостей нет, то цикл можно векторизовать или распараллелить. Если же такая зависимость есть, то следует учитывать направление (предыдущая или будущая итерация, иначе говорят – обратная и прямая) и расстояние (количество итераций, разделяющих чтение и запись).

ЦСП (цифровой сигнальный процессор, англ. **DSP).** Вычислительное устройство, специально предназначенное для цифровой обработки сигналов, в частности требуемой в радиосвязи: фильтры, быстрое преобразование Фурье, аналого-цифровые преобразования. Вычислительные возможности ЦСП в сочетании с ЦП привели к появлению первых гетерогенных платформ и различных расширений языков программирования для управления ЦСП и взаимодействия с ними. OpenCL – модель программирования, призванная поставить на службу программе вычислительные средства ЦСП. *См.* также **Гетерогенные платформы**.

Число с плавающей точкой. Формат чисел в компьютерах, для которого характерен компромисс между широким диапазоном представления и сниженной точностью – благодаря разделению имеющихся разрядов на мантиссу и показатель степени – величину сдвига, обозначающего, в каком месте слева или справа от фиксированной позиции находится точка. Напротив, в представлениях с фиксированной точкой явного показателя степени нет, поэтому все доступные разряды используются для самого числа (играют роль мантиссы).

Ядро. Отдельный подпроцессор многоядерного процессора. Ядро должно поддерживать по крайней мере один отдельный поток управления, отличающийся от потоков в других ядрах того же процессора.

Предметный указатель

К

Л

М

Н

О

П

P

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Майкл Восс, Рафаэль Асенхо, Джеймс Рейндерс

Параллельное программирование на C++ с помощью библиотеки TBB

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 54,76. Тираж 200 экз.

Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1

Веб-сайт издательства: **www.dmkpress.com**