МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ
ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»**

Кафедра «Автоматизированные системы управления»

А.В. ВОЛОСОВА

ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ И АЛГОРИТМЫ

Учебное пособие

МОСКВА
МАДИ
2020

УДК 004.9
ББК 32.81
В 683

Рецензенты:

д-р техн. наук, проф., заведующий кафедрой "АСУ" МАДИ Максимычев О.И.

д-р. техн. наук, проф., Министерство по делам Северного Кавказа Голубятников И.В.

Волосова, А.В.

В683 Параллельные методы и алгоритмы [Электронный ресурс]: учебное пособие/ А.В. Волосова. – М.: МАДИ, Электронные текстовые и графические данные (5,35 Мбайт). 2020. – 176 с. Учебное электронное издание комбинированного распространения: 1 CD-диск. – Систем. требования: PC 486 DX-33; Microsoft Windows XP; 2-скоростной дисковод CD-ROM; Adobe Reader 6.0. Официальный сайт Московского Автомобильно-дорожного государственного технического университета (МАДИ). Режим доступа: http://www. http:// lib.madi.ru //— Загл. с титул. экрана.

В учебном пособии рассматриваются вопросы эффективного решения больших задач на компьютерах с параллельной архитектурой: архитектуры параллельных вычислительных систем, численные методы решения задач, технологии параллельного программирования, проблемы современных параллельных вычислений

Учебное пособие может быть рекомендовано студентам по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.02 «Информационные системы и технологии», 38.03.01 «Экономика» и др.

УДК
004.9 ББК
32.81

Введение..6
Глава 1. Введение в параллельные вычисления..7
1.1. Параллелизм на уровне задач (многопоточность): 8
1.2. Параллелизм на уровне данных 8
1.3. Параллелизм на уровне алгоритмов 9
1.4. Параллелизм на уровне инструкций (команд)..9
Глава 2. Отображение параллельных структур алгоритмов и программ на архитектуру компьютера 15
2.1. Определение проблемных точек процесса реализации программы 15
2.2. Состав, принцип работы, временные характеристики (АЛУ)...........................18
Глава 3. Способы повышение производительности компьютера 27
3.1. Способы параллельной обработки данных...27
3.1.1. Параллельная обработка...28
3.1.2. Конвейерная обработка...29
Глава 4. Способы повышения эффективности работы программ 34
4.1. Разработка спецпроцессоров...34
4. 2. Улучшение архитектуры многопроцессорных конфигураций 35
4.3. Способы организации коммуникационных систем в суперкомпьютерах 37
4.4. Компьютеры с неоднородным доступом к памяти компьютеры с архитектурой NUMA (Non Uniform Memory Access)..39
4.5. Программное обеспечение параллельных компьютеров......................................40
Глава 5. Архитектура параллельных вычислительных систем.................................43
5.1. Классификация архитектур вычислительных систем...43
5.2. Векторно-конвейерные компьютеры (архитектура векторно-конвейерных супер-ЭВМ CRAY C90)..53
5.3. Параллельные компьютеры с общей памятью (HP Superdom) 58
(Cray T3D/T3E) 59
Глава 6. Большие задачи и параллельные вычисления 65
6.1. Решение больших задач..65
6.2. Граф алгоритма и параллельные вычисления 66
6.3. Концепция неограниченного параллелизма 67
6.4. Внутренний параллелизм 70
Глава 7. Параллельные алгоритмы..71

7.1. Классификация алгоритмов по типу параллелизма ... 73

7.2. Общая схема разработки параллельных алгоритмов ... 73

7.3. Простые параллельные операции ... 76

Глава 8. Базовые методы построения параллельных алгоритмов: метод сдваивания, геометрический параллелизм ... 78

8.1. Базовые методы построения параллельных алгоритмов ... 78

8.2. Построение параллельного алгоритма методом сдваивания ... 79

8.3. Метод геометрического параллелизма ... 83

Глава 9. Базовые методы построения параллельных алгоритмов конвейерный параллелизм; диффузная балансировка загрузки ... 87

9.1. Конвейерный параллелизм ... 87

9.2. Диффузная балансировка загрузки ... 91

Глава 10. Типы параллельных алгоритмов ... 94

Глава 11. Типы параллельных алгоритмов ... 101

11.1. Параллельный алгоритм поиска кратчайшего пути ... 101

11.2. Параллельный алгоритм поиска минимального остовного дерева ... 104

11.3. Генетические параллельные алгоритмы ... 106

Глава 12. Расширенные методы построения параллельных алгоритмов ... 114

12.1. Генерация псевдослучайных чисел ... 114

12.2. Декомпозиция сеточных графов ... 121

Глава 13. Расширенные методы построения параллельных алгоритмов ... 129

13.1. Динамическая балансировка загрузки процессоров ... 129

13.2. Визуализация сеточных данных ... 129

13.3. Сети сортировки ... 138

Глава 14. Технологии параллельного программирования: использование традиционных последовательных языков ... 141

14.1. Принципы анализа параллельных алгоритмов ... 141

14.2. Использование традиционных последовательных языков (системы программирования OpenMP, DVM, mpC). ... 141

14.2.1. OpenMP ... 142

14.2.2. Система программирования DVM ... 144

14.2.3. Система программирования mpC ... 147

Глава 15. Технологии параллельного программирования: системы программирования на основе передачи сообщений и другие системы ... 150

15.1. Системы программирования на основе передачи сообщений (системы параллельного программирования Linda, Message Passing Interface (MPI), MPI-2, DVM 150

15.2. Другие языки и системы программирования (Т-система, НОРМА). 156

Глава 16. Методы исследования и эквивалентные преобразования параллельных алгоритмов и программ 161

16.1. Информационная структура алгоритмов и программ 161

16.2. Графовые модели программ 163

16.3. Классы программ 170

16.4. Эквивалентные преобразования параллельных алгоритмов и программ 172

16.5. Оптимизация программ. Система V-Ray 173

Заключение 174

Список литературы 175

Введение

Рост сложных задач, решение которых связано с применением современных ИТ-технологий, ведет к необходимости использования параллельных вычислений. Параллельные вычисления носят междисциплинарный характер. Они затрагивают, в частности, такие области, как численные методы, структуры и алгоритмы обработки данных, аппаратное и программное обеспечение, системный анализ. Это позволяет применять знания, полученные при изучении параллельных вычислений, в различных сферах научно-практической деятельности.

В рамках учебного пособия рассматриваются методы и средства параллельных вычислений.

Глава 1. Введение в параллельные вычисления

В ИТ-технологиях термин "параллельный" связан с решением сложных задач. Использование нескольких компьютеров одновременно для реализации отдельных частей сложной задачи привело к возникновению терминов «параллельные компьютеры» и «параллельные вычислительные системы». Разработка математического и программного обеспечения для решения сложной задачи – к возникновению терминов «языки параллельного программирования», «параллельные численные методы», «параллельные алгоритмы». Важно запомнить , что термин «параллельный», применительно к решению сложных задач, заменяет слова «одновременный», «независимый» и означает, что части сложной задачи выполняются независимо друг от друга.

Часто вместо термина «параллельный» используется термин «параллелизм», имеющий такое же значение. Например, параллелизм данных, параллелизм задач и т.п.

Таким образом, параллельные вычисления охватывают вопросы, касающиеся создания ресурсов параллелизма в процессах решения задач и обеспечения его эффективной реализации. Целью любого параллелизма является повышение эффективности работы компьютера на различных уровнях, а именно на уровне:

- задач
- данных
- алгоритмов
- инструкций
- битов.

Рассмотрим каждый из уровней.

1.1. Параллелизм на уровне задач (многопоточность):

Данный вид параллелизма относится к парадигме параллельного программирования и предполагает разбиение задачи на подзадачи. Подзадачи реализуются одновременно. Многопоточность предполагает:

1. Обнаружение в программе вычислений, реализацию которых можно осуществлять параллельно.
2. Последующее распределение данных по модулям локальной памяти процессоров.
3. Последующее согласование параллелизма вычислений и распределения данных.

Параллелизм на уровне данных эффективен на однородных задачах – задачах, не допускающих возможности разбиения на подзадачи.

1.2. Параллелизм на уровне данных

Данный вид параллелизма реализуется компилятором и заключается в замене множества выполнений однотипных операций одной операцией над множеством данных. Обработка осуществляется на векторных процессорах. В программах, обрабатываемых компилятором, задаются директивы компиляции, что предполагает использования языков параллельных вычислений. Рассмотрим два независимых устройства, которые могут работать одновременно. Причем каждое устройство может выполнять суммирование двух чисел за 5 тактов. При этом устройство не может выполнять другую работу. Тогда вся работа выполнится одним устройством за 500 тактов. Постоянно загружая каждое устройство элементами входных массивов, например векторов, имеющих по 100 вещественных чисел каждый, можно получить искомую сумму уже за 250 тактов – ускорение выполнения работы в два раза. В случае 10 подобных устройств, время получения результата составит всего 50 тактов, а в общем случае система из N устройств затратит на суммирование время около 500/N.

Пример. В 50-х годах XX века академик А.А. Самарский занимался моделированием ядерных взрывов. При осуществлении расчетов он использовал одновременно несколько арифмометров, с каждым из которых работал оператор. Данный способ организации вычислений явился прообразом параллельных вычислений и позволил рассчитать эволюцию взрывной волны.

1.3. Параллелизм на уровне алгоритмов

Данный вид параллелизма предполагает замену последовательных алгоритмов некоторых вычислений на параллельные. Это касается алгоритмов поиска, сортировки и т.п. Организация процесса распараллеливания осуществляется за счет использования различных средств параллельного программирования, таких как, специальные библиотеки, переменные окружения, директивы компилятора и т.п. Примером может служить технология OpenMP, которая подробно рассматривается в главе 14.

1.4. Параллелизм на уровне инструкций (команд)

Под инструкциями будем понимать – команды, выполняемые процессором. Параллелизмом на уровне команд предполагает совмещение выполнения команд. Этот процесс не меняет результат программы.

Рассмотрим проявление рассматриваемого параллелизма на уровне команд на аппаратном уровне и на программном уровне:

1.4.1. Аппаратный уровень

На аппаратном уровне параллелизм реализуется в следующих формах:

1.4.1.1. Внеочередное исполнение машинных инструкций — технология, реализующая вычисления по мере наличия готовых к обработке данных в регистрах процессора. Тем самым исключаются простои процессора, вызванные ожиданием данных инструкций, выполняемых в порядке следования в машинном коде. При внеочередном исполнении

применяется технология **переименования регистров.** При внеочередном исполнении, когда данные в регистр записываются после выполнения двух и более команд. Следующая команда не может быть выполнена до завершения предыдущей. Если команды используют не связанные данные, то возможно появление **ложной зависимости**. Такая зависимость уменьшает производительность процессора. При использовании данной технологии ссылки на архитектурные регистры преобразовываются в ссылки на физические регистры, что снижает вероятность возникновения ложных зависимостей.

На данном уровне параллелизма используется также такая технология, как **конвейер,** которая позволяет увеличить число инструкций, выполняемых в единицу времени. Конвейеры используются в процессорах и контроллерах. **Пусть имеется** поток задач. Все задачи потока разбиваются на подзадачи, каждая из которых может быть обработана свободной от нагрузки частью вычислительной системы. Устройство, называемое **модуль прогнозирования ветвлений**, используемое в данной технологии, рассчитывает прогноз выполнения условного перехода в исполняемой программе. Инструкции, следующие за инструкцией условного перехода, загружаются и частично обрабатываются. Если прогноз не выполняется, то загружаются правильные инструкции. Здесь также имеет место **переименование регистров**.

1.4.1.2. Суперскалярный процессор – процессор, вычислительное ядро которого состоит из нескольких АЛУ, модулей операций с плавающей точкой, умножителей других подобных устройств. Вычислительное ядро осуществляет динамическое управление потоком инструкций.

Параллелизм на уровне команд позволяет отказаться от параллельного программирования.

1.4.2. Программный уровень

Программный параллелизм на уровне команд осуществляется компилятором. Компилятор формирует исполняемый код непосредственно под конкретный процессор.

С параллелизмом на уровне команд и данных связана классификация Флинна осуществляется в пределах потоков команд и данных и зависит от уровня параллелизма в этих потоках. Была предложена Майклом Флинном в 1966 году. Классификация относится к архитектурам ЭВМ, которые делятся на следующие классы:

ОКОД (SISD) – архитектура с одиночным потоком команд и одиночным потоком данных;

ОКМД (SIMD) – архитектура с одиночным потоком команд и множественным потоком данных;

МКОД (MISD) – архитектура со множественным потоком команд и одиночным потоком данных;

МКМД (MIMD) – архитектура со множественным потоком команд и множественным потоком данных.

1.5. Параллелизм на уровне битов

Все процессы, связанные с решением задачи, происходят на битовом уровне. Переход к битовому уровню и обратно осуществляется совокупностью аппаратных и программных средств. Качество реализации таких переходов непосредственно влияет на эффективность решения задачи. Таким образом, понимание устройства компьютера, а следовательно и механизмов реализации задачи на машинном уровне, может оказаться полезным в том случае, если нет возможности повлиять на эффективность решения задачи на более высоких уровнях, таких как уровень инструкций, данных и задач. Рассмотрим ключевые узлы архитектуры компьютера, работа которых оказывает влияние на эффективность решения задачи.

Ошибка!

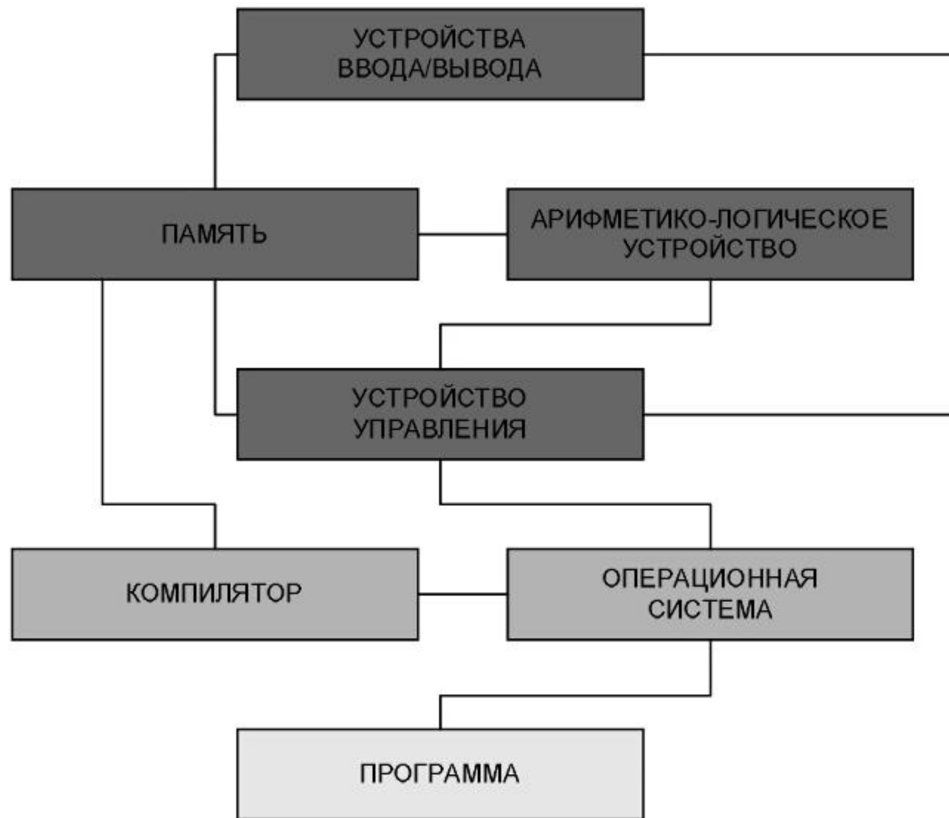


Рис. 1. Общая схема компьютера

Бит является минимальной единицей хранения информации. Упорядоченный набор битов называется **словом** и является основным информационным элементом компьютера. Слово может делиться на упорядоченные наборы из 8 битов - **байты**. **Длиной слова** называется количество байтов. Рассмотрим общую схему компьютера (рис. 1).

Память – устройство, в котором хранится все множество поименованных слов. Имя слова называется **адресом.** Устройство и организация памяти имеют решающее влияние на время решения задачи.

Арифметико-логическое устройство (АЛУ) - совокупность устройств, реализующих арифметические и логические операции**. Операция** – функция, выполняемая над содержимым слов. Аргумент функции или

значение аргумента или адрес слова, которое содержит аргумент называется **операндом функции.** Арифметические операции – операции над числами (сложение, вычитание, умножение и т. д.). Логические операции – операции булевой алгебры над битами слов (конъюнкция, дизъюнкция и т. д.). Выполнение последовательности функций над содержимым слов осуществляет преобразование хранящейся в памяти информации.

В состав АЛУ входят также **сумматоры -** устройства, выполняющие операции сложения/вычитания и **умножители – устройства,** выполняющие операции умножения. Операции с фиксированной запятой выполняются быстрее аналогичных операций с плавающей запятой. Операции сложения выполняются быстрее операций умножения. Любые логические операции реализуются быстрее операций сложения/вычитания с фиксированной запятой. На каждом конкретном компьютере длительности выполнения стандартных операций отличаются друг от друга не более, чем в несколько раз.

Программа – алгоритм, записанный на языке программирования

Компилятор - программные системы, осуществляющие перевод программ на языке высокого уровня (ЯВУ) в машинный код.

Машинный код - совокупность машинных команд из числа возможных для данного компьютера (**программа во внутреннем коде**). Во внутреннем коде описываются процессы преобразования информации программами. От качества преобразования программы на ЯВУ в машинный код зависит эффективность реализации машинного кода, которая кардинально влияет на эффективность процесса решения задач.

Устройства ввода - осуществляют ввод данных в память.

Устройство вывода осуществляют вывод данных из памяти.

Устройство управления (УУ) – совокупность устройств

(регистры, счетчики и т.п.), которые реализуют управление перемещением информации между памятью, АЛУ, УВВ и другими частями компьютера. Управление осуществляется на нескольких уровнях. Например,

уровень электронных схем, который осуществляют дешифрацию содержания текущей команд, выделяет код операции и адреса операторов, осуществляют выбор следующей команды или группы команд. За выполнение операций АЛУ отвечают схемы другого уровня.

Операционная система – совокупность программных систем, обеспечивающих управление работой компьютера.

Параллелизм на уровне битов предполагает изменение размера машинного слова.

Задания

1. Приведите примеры физических, электрических и механических устройств, имеющих p = 2, 3, 8, 10 устойчивых состояний.
2. На основе предложенных устройств постройте схемы выполнения арифметических операций над целыми числами.
3. Исследуйте любой калькулятор как компьютер. Каковы в нем центральный процессор, память, операционная система, форма представления чисел?
4. Какие задачи можно решать на компьютере, имеющем универсальный процессор и память объемом в 1 или 2 ячейки?
5. Рассмотрите любую систему машинных команд. С помощью команд, реализующих логические операции и операции безусловного перехода, составьте программу, реализующую операцию условного перехода.

Глава 2. Отображение параллельных структур алгоритмов и программ на архитектуру компьютера

2.1. Определение проблемных точек процесса реализации программы

Повысить эффективность решения задач на компьютере возможно следующими способами:

- улучшить математические модели,
- разработать новые численные методы,
- улучшить программу.

Обычно программист не оценивает возможности компилятора и операционной системы. Влияние этих компонентов на эффективность реализации пользовательских программ при создании и исполнении машинных кодов может быть достаточно велико, но выявить проблемные места созданного машинного кода и внести соответствующие изменения в код на языке высокого уровня достаточно затруднительно.

Даже в случае получения такой информации, ею редко удается воспользоваться при выборе путей модернизации как самой программы, так и реализуемых ею алгоритмов.

Системное программное обеспечение компьютера не содержит средств, позволяющих анализировать структуры программных кодов. Затруднительно найти информацию об эффективности работы ключевых для программы конструкций языка высокого уровня. Тем более важно выявить параллельных структуры алгоритмов и программ и спроецировать их на архитектуру компьютеров.

Главный вектор развития компьютеров – повышение производительности, т. е. возможности выполнять большее число операций в единицу времени.

Решающее влияние на производительность оказывает арифметико-логическое устройство. Кроме АЛУ существуют менее значимые устройства,

тоже влияющие на производительность компьютера. И часто это влияние превалирует над влиянием более значимых устройств, особенно при возрастании сложности решаемой задачи. В этом случае сбои происходят чаще. Данная проблема может быть решена путем смены вычислительных мощностей, что либо приводит к дополнительным расходам, либо заканчивается неудачей.

Таким образом, предпочтительнее решить проблему на том же компьютере. Для этого программист должен обладать знаниями о средствах реализации программы на низком уровне, таких как:

1. Арифметико-логическое устройство (АЛУ), в составе которого работают такие устройства, как сумматоры, умножители, сопроцессоры и т. п. Работа этих устройств не связано непосредственно с задачей, реализуемой программой. Текущей организацией работы производственного отдела занимается сектор оперативного управления.
2. Устройство управления (УУ), которое связано с памятью, устройство которой влияет на производительность. В работе с данными участвуют регистровая память, кэш-память, оперативная память, медленная память.
3. Компилятор, участвующий в формировании машинного кода.
4. Оборудование, обеспечивающее транспортировку данных - различные соединения, шины, каналы и т.д.
5. Операционная система, которой принадлежит организационная роль.

Время решения задачи определяется:

- мощностью оборудования АЛУ;
- скоростью работы оборудования, обеспечивающей транспортировку данных;
- качество сгенерированного компилятором машинного кода;

- эффективностью работы операционной системы с данными, перемещаемыми из памяти.

Базовая производительность компьютера определяется мощностью оборудования АЛУ. Суммарная производительность АЛУ называется **пиковой производительностью**. Это понятие не учитывает временные затраты от деятельности других устройств, участвующих в выполнении программы.

Существует дополнительное понятие – «**реальная производительность**» вычислительного устройства. Реальная производительность системы устройств – количество операций, реально выполненных в среднем за единицу времени. Эта производительность не может превышать пиковую. **Отношение реальной производительности к пиковой называется эффективностью работы системы.**

Следовательно, пиковая производительность оказывает решающее влияние на величину эффективности работы вычислительной системы.

Рассмотрим способы увеличения пиковой производительности:

- **наращивание однотипного оборудования;**
- **организация конвейеров;**
- **комбинация перечисленных способов.**

При детализации задачи увеличения пиковой производительности имеет смысл повысить производительность памяти, АЛУ, оборудования, обеспечивающего транспортировку данных, УУ.

Значение эффективности работы компьютера (на большинстве программ) – **[0,5; 1]** можно считать хорошим. При меньших значениях необходимо искать узкие места процесса функционирования компьютера и его программного окружения в целом, в которых теряется производительность. Они в первую очередь могут определяться:

- составом, принципом работы и временными характеристиками арифметико-логического устройства;
- составом, размером и временными характеристиками памяти;

- структурой и пропускной способностью коммуникационной среды;
- компилятором, создающим неэффективные коды;
- операционной системой, организующей неэффективную работу с памятью, особенно медленной.

2.2. Состав, принцип работы, временные характеристики (АЛУ)

Состав АЛУ:

- УУ, задающее последовательность микрокоманд. В зависимости от уровня поступления, микрокоманды делятся на внутренние и внешние;

операционное устройство, реализующее последовательность микрокоманд.

АЛУ выполняет следующие группы команд:

- операции двоичной арифметики для чисел с фиксированной точкой;
- операции двоичной (или шестнадцатеричной) арифметики для чисел с плавающей точкой;
- операции десятичной арифметики;
- операции индексной арифметики (при модификации адресов команд);
- операции специальной арифметики;
- операции над логическими кодами (логические операции);
- операции над алфавитно-цифровыми полями.

Принцип работы АЛУ

На рис. 2 представлен принцип работы АЛУ. В зависимости от информации, поступающей на входы ***S*** и ***M,*** выполняется одна из 32 возможных операций.

Состояние входов S				Состояние входа M	
S3	S2	S1	S0	M=1	M=0 (C=0)
0	0	0	0	A	A\/1
0	0	0	1	A+B	(A+B)\/1
0	0	1	0	A+B	(A+B)\/1
0	0	1	1	0	0
0	1	0	0	A+B	(A\/A)x(B\/1)
0	1	0	1	B	(A+B)\/A(B\/1)
0	1	1	0	AxB\/AxB	A-B
0	1	1	1	AxB	AxB
1	0	0	0	A+B	A\/AxB\/1
1	0	0	1	AxB+AxB	A\/B\/1
1	0	1	0	B	(A+B)\/AxB\/1
1	0	1	1	AxB	AxB
1	1	0	0	1	A\/A\/1
1	1	0	1	A+B	(A+B)\/A1\/1
1	1	1	0	A+B	(A+B)\/A\/1
1	1	1	1	A	A

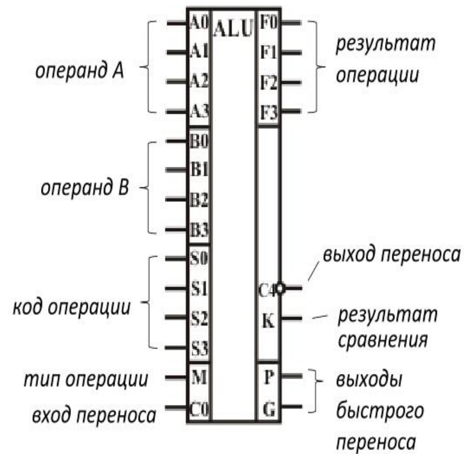


Рис. 2. Принцип работы АЛУ

Временные характеристики АЛУ различают:

- синхронные характеристики характеризуют синхронные АЛУ, выполняющие за один такт каждую операцию.
- асинхронные характеристики характеризуют не тактируемые АЛУ, которые выполняются на комбинационных схемах.

2.3. Состав и временные характеристики памяти

Компьютерная память обеспечивает длительное хранение информации и является ключевыми звеньями архитектуры фон Неймана. На рис. 3 представлен состав памяти.

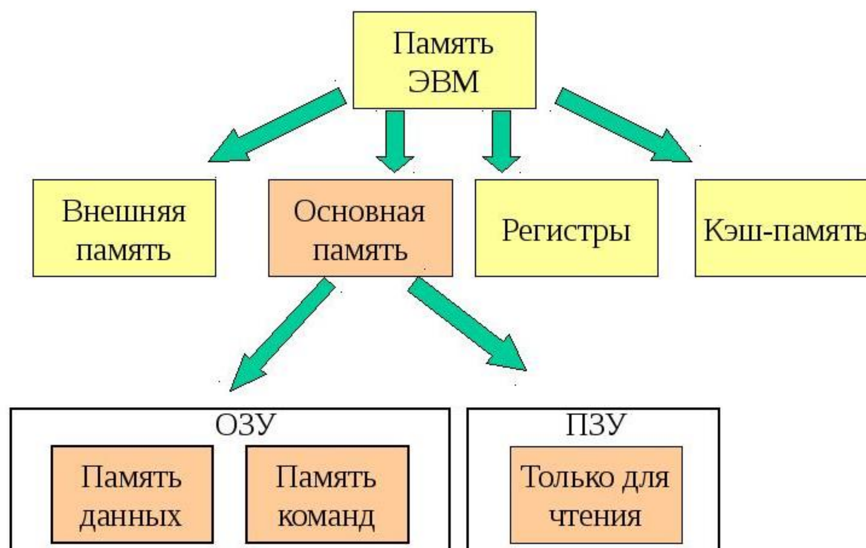


Рис. 3. Состав памяти компьютера

Внешняя память – память, предназначенная для долговременного хранения информации на различных носителях.

Основная память – устройство для хранения информации.

Оперативной памятью (ОП) будем называть адресуемую часть памяти, время доступа к которой значительно меньше времени выполнения операций – быструю память. Память позволяет записывать и считывать информацию.

Оперативное запоминающее устройство (ОЗУ) – техническое устройство, соответствующее оперативной памяти.

Неадресуемая часть памяти – **медленная память**. Эта часть памяти недоступна пользователю. Она делится на **постоянную память** и **сверхбыструю память**.

Постоянная память допускает только считывание информации - команды запуска компьютера, различные служебные программы и т. п.

Сверхбыстрая память – неадресуемая часть памяти, имеющая 1-2 уровня. Управление сверхбыстрой памятью осуществляет устройство управления.

Регистровая память (регистры) – самый быстрый уровень сверхбыстрой памяти. Обычный объем регистровой памяти – единицы или десятки слов. В регистровой памяти хранятся результаты выполнения операции, необходимые для реализации команды, следующей за исполняемой. Регистровая память входит в состав АЛУ.

Кэш-память – связующим звеном между регистровой памятью и оперативной памятью. Объем кэш-памяти может достигать миллиона слов. В кэш памяти содержаться результаты выполнения операции, которые необходимы для реализации команд, следующих за исполняемыми. Иногда компилятор в процессе перевода программ в машинный код может выстраивать команды так, чтобы был достигнут максимальный эффект от использования сверхбыстрой памяти.

Виртуальная память - метод управления памятью, которая реализуется с использованием аппаратного и программного обеспечения компьютера. Виртуальная память отображает используемые программами виртуальные адреса в физические адреса в памяти компьютера. Реализацию виртуальной памяти осуществляют алгоритмы операционной системы.

Временные характеристики памяти

Быстродействие памяти характеризуется двумя параметрами:

1. Временем доступа - промежуток времени между формированием запроса на чтение информации из памяти и моментом поступления из памяти запрошенного машинного слова
2. Длительностью цикла памяти – минимальным допустимым временем между двумя последовательными обращениями к памяти.

Для минимизации времени доступа к оперативной памяти, можно минимизировать время прохождения управляющих сигналов к ее элементам. Это время зависит в свою очередь от длины соединений. При большом числе соединений эти длины нельзя сделать равномерно малыми. Размеры неравномерности прямо пропорциональны количеству элементов. Неоднородность памяти влияет на время доступа. Для уменьшения большого

разброса времени доступа в ОЗУ используют иерархию, которая образуется при делении памяти на кубы, блоки, секции, страницы и т. п. Высота уровня иерархии обратно пропорциональна разбросу времен доступа к отдельным словам одного раздела памяти данного уровня. На максимальном уровне существуют группы слов с последовательными физическими адресами или адресами, меняющимися с постоянным шагом, доступ к которым может быть осуществлен или одновременно, или с минимальной разностью времен.

Увеличение объема оперативной памяти неизбежно приводит к усложнению ее структуры, что в свою очередь увеличивает разброс времен доступа к отдельным словам. Операционные системы и компиляторы поддерживают наилучший режим работы с оперативной памятью для определенных программ. Различают два типа задач, с учетом этого режима, которые позволяют с точки зрения использования этого режима следует различать **два типа задач**:

1. **Задачи, при решении которых используется только оперативная память**

Компьютеры особенно быстро осуществляют доступ к словам с последовательными физическими адресами. Пусть, например, программа составляется на языке Фортран. В пределах этого языка нет разницы в обработке элементов массивов по строкам или столбцам. В первом случае осуществляется преобразование элементов матриц. Во втором случае - элементов, транспонированных к ним. Массивы в Фортране располагаются в физической памяти последовательно по адресам; столбцы располагаются друг за другом. В случае алгоритм решения систем линейных алгебраических уравнений методом Гаусса, время работы программы будет определяться ее трехмерными циклами. Внутренние циклы программ могут осуществлять обработку, как уже упоминалось выше, по строкам и столбцам. В зависимости от того, расположена матрица системы в массиве по столбцам или строкам, внутренние циклы программы осуществляют обработку

элементов массивов по столбцам или строкам. Время реализации для этих двух случаев может значительно отличаться.

2. Задачи, при решении которых используется медленная память

Увеличение объема памяти вызывает необходимость задействовать, наряду с оперативной, медленную память, которая часто реализуется на жестких дисках. Объемы информации, необходимые при решении больших задач, требуют хранения большей ее части в медленной памяти. С учетом того, что операции выполняются только в оперативной памяти, необходим многократный обмен данными между оперативной и медленной памятью. Таким образом, время решения задачи складывается из времени выполнения операций алгоритма и времени осуществления обменов между медленной и оперативной памятью. При неудачной организации обменов время обменов может превысить время выполнения операций алгоритма в тысячи раз.

Организация обменов между медленной и быстрой памятью должна осуществляется с учетом следующих требований:

- осуществление минимально возможного числа обращений к медленной памяти;
- осуществление переноса максимально возможного количества данных за обмен;
- достижение максимально возможного времени обработки данных при переносе в быструю память.

Для реализации обмена, с учетом этих требований, можно использовать виртуальную память, которая либо однородна, либо имеет некоторую структуру. В последнем случае для памяти разрабатывается система правил предпочтительного размещения данных. Универсальные алгоритмы, реализующие память практически не учитывают особенности конкретных программ.

Для проверки организации обменов существует следующая методика:

- определить пример, отражающий существо решаемой большой задачи;
- замерить в монопольном режиме астрономическое время реализации примера на компьютере;
- подсчитать время выполнения необходимых операций;
- найти отношение величин времени реализации примера и времени выполнения необходимых операций;
- провести анализ полученных величин; если первая величина отличается от второй не более чем в несколько раз, то для рассматриваемого класса задач организация обмена является удовлетворительной, иначе – обмен следует организовывать самостоятельно. Существуют задачи, для которых нельзя минимизировать время обменов с медленной памятью.

Пример [1, с. 44]

В середине 60-х годов прошлого столетия в нашей стране были широко распространены вычислительные машины типа М-20, построенные коллективом, возглавляемым академиком С. А. Лебедевым. По тем временам это были вполне современные компьютеры, обладающие производительностью около 20 тыс. операций в секунду. Оперативная память машин этого типа была малой и позволяла решать системы линейных алгебраических уравнений с плотной матрицей порядка всего лишь 50—100. Требования практики заставляли искать способы решения систем значительно большего порядка. В качестве медленной памяти на этих машинах использовались магнитные барабаны. Они играли тогда такую же роль, какую сейчас в персональном компьютере играют жесткие диски. Естественно, с поправкой на объем хранимой информации. Под этот тип медленной памяти была разработана специальная блочная технология решения больших алгебраических задач. Она позволяла с использованием только 300 слов оперативной памяти решать системы практически любого порядка. Точнее, такого порядка, при котором матрица и правая часть могли целиком разместиться в медленной памяти. При этом системы решались

почти столь же быстро, как будто вся информация о них на самом деле была размещена в оперативной памяти. Созданные на основе данной технологии программы были весьма эффективны. В частности, на машинах типа М-20 они позволяли решать системы 200-го порядка всего за 9 минут.

В конце 60-х годов прошлого столетия появилась машина БЭСМ-6. Это была одна из лучших вычислительных машин в Европе. Она обладала производительностью около одного миллиона операций в секунду, т. е. была быстрее машин типа М-20 примерно в 50 раз. Разработана БЭСМ-6 была коллективом под руководством академика С. А. Лебедева, его заместителей В. А. Мельникова и Л. Н. Королева. Среди математического обеспечения машин БЭСМ-6 были и стандартные программы для решения системы линейных алгебраических уравнений большого порядка с использованием медленной памяти. Ожидалось, что с их помощью можно решать большие системы, если и не в 50, то хотя бы в несколько раз быстрее. Однако опытная проверка дала удивительные результаты. Система 200-го порядка решалась за 20 минут. Данный результат явился следствием небрежного использования программистами медленной памяти.

Задания

1. Пусть один землекоп может за час вырыть яму размером 1х1х1 $м^3$ и способен работать в таком режиме достаточно долго. За какое время бригада из 5, 10, 20 землекопов выроет яму размером 2х2 $м^2$ и глубиной 1 м?
2. Постройте график времени выполнения работы в зависимости от числа землекопов в бригаде.
3. Повторите задания п.п. 1, 2 для ямы размером 10х10 $м^2$, глубиной 1 м и бригады из 10, 100 землекопов.
4. Чем принципиально различаются эти варианты?
5. Пусть предпринимается попытка увеличить реальную мощность производственного отдела из иллюстративной модели

компьютера путем наращивания однотипного оборудования. Не кажется ли вам, что возникнет ситуация, аналогичная рассмотренной в пп. 1 - 4?

6. Для создания очень мощного компьютера часто объединяют вместе большое число персональных компьютеров. Пусть их будет n. Для обмена информацией между компьютерами необходимо создать коммуникационную сеть. Теоретически самый простой способ – соединить каждый компьютер с каждым прямыми связями, т. к. в этом случае информация вроде бы должна передаваться наиболее быстро. Всего потребуется n(n-1) таких связей. Рассмотрите подобные коммуникационные сети для n = 3, 5, 8.
7. В реальных компьютерах величина n может достигать 10^4–10^5. Можете ли вы представить всю совокупность связей в этом случае, если они организованы согласно п. 6?
8. Пусть снова сверхмощный компьютер создается как объединение персональных компьютеров. Построим неориентированный граф, в котором вершины символизируют персональные компьютеры, а ребра – прямые связи. Будем считать, что граф связный, т. е. для каждой пары вершин существует связывающая их цепь ребер. Такой граф называется коммуникационным. Исследуйте коммуникационный граф из п. 6.
9. Пусть в коммуникационном графе существуют не все ребра. Будем считать, что передача информации между персональными компьютерами может осуществляться вдоль любой цепи коммуникационного графа, связывающей соответствующие вершины. Постройте различные коммуникационные графы, уменьшая общее число ребер.
10. Решите на своем компьютере систему линейных алгебраических уравнений 200-го порядка методом Гаусса и измерьте время ее

решения. Далее сравните отношение производительностей и отношение времен решения систем для обоих компьютеров. Оцените результаты сравнения.

11. Решите предыдущую задачу для системы большого порядка с использованием жесткого диска в качестве медленной памяти. Оцените результаты.

Глава 3. Способы повышение производительности компьютера

Для повышения производительности программы можно:

- минимизировать конфликты в памяти;
- оптимизировать распределение вычислительной нагрузки между процессорами;
- оптимизировать распределение данных;
- оптимизировать структуру программы;
- заменить последовательные части алгоритма на параллельные;
- решать проблемы, характерные для последовательных компьютеров и традиционного последовательного программирования.

Следует учесть, что скорость развития параллельных вычислений всегда опережала скорость развития вычислительной техники для осуществления этих вычислений. Путь совершенствования устройства микросхем менее результативен по сравнению с эффектом применением параллелизма в архитектуре компьютера.

Рассмотрим возможности оптимизации распределения данных.

3.1. Способы параллельной обработки данных

Рассмотрим два способа параллельной обработки данных:

1. Параллельная обработка
2. Конвейерность

3.1.1. Параллельная обработка

Пример [1, с. 45]. Требуется найти сумму двух векторов, состоящих из 100 вещественных чисел каждый. Некоторое устройство выполняет суммирование пары чисел за пять тактов работы компьютера. Устройство блокируется на все время выполнения операции. Операция выполняется за 500 тактов (рис. 4). Если добавить еще одно такое устройство, то при их независимой работе операция будет выполнена за 250 тактов (рис. 5). Для ***n*** время выполнения операции суммирования составит 500/N тактов.

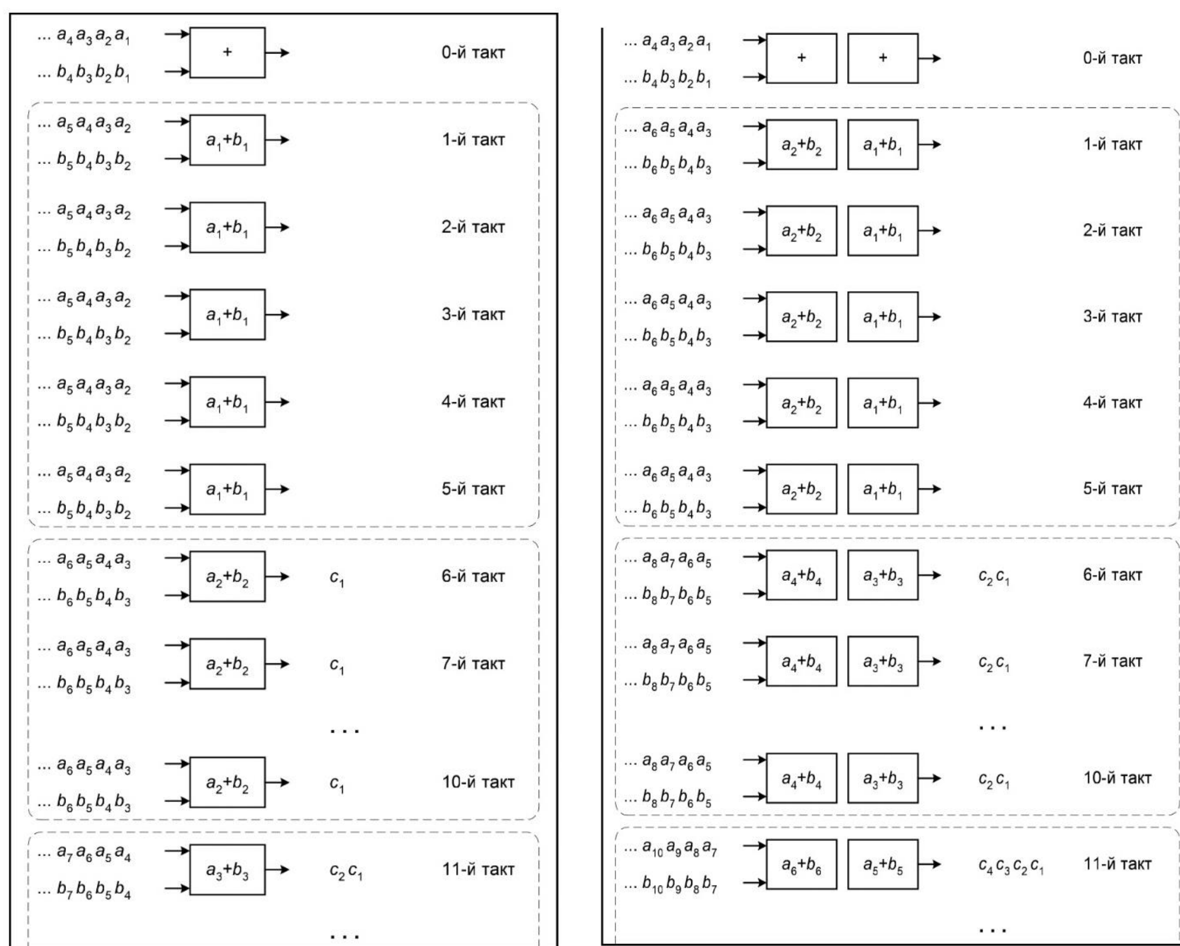


Рис. 4. Суммирование векторов C = A + B: а) с помощью последовательного устройства, выполняющего одну операцию за пять тактов, б) с помощью двух последовательных устройств, выполняющего одну операцию за пять тактов каждое

3.1.2. Конвейерная обработка

Другой способ организации процесса обработки всех элементов входных массивов заключается в использовании освободившихся в процессе сложения микроопераций, из последовательности которых состоит операция. Это микрооперации сравнение порядков, выравнивание порядков, сложение мантисс, нормализация. Использование освободившихся микроопераций становится возможным так как порядок их выполнения при обработке входных данных всегда один и тот же. Поэтому освободившаяся операция может быть использована для обработки следующей пары чисел на входе устройства. Устройство, использующее подобный механизм, размещает каждую микрооперацию в отдельной части и после окончания работы микрооперации передает обработанные данные следующей части. В то время как освободившаяся микрооперация может начать обрабатывать следующую порцию данных. Когда будет исчерпан входной набор данных, устройство выдает результат своей работы. Подобное устройство называется **конвейером**. Каждая часть устройства называется **ступенью конвейера**, а общее число ступеней – **длиной конвейера**. Время выполнения контейнером одной операции равно суммарному времени работы каждой ступени. При реализации суммирования векторов на конвейере на выполнение операции потребуется 104 такта (рис. 5). Таким образом, при наличии m ступеней в конвейере время обработки n операций, которые не зависят друг от друга – ***(m + n – 1)*** единиц (время работы одной ступени – 1 единица). По сравнению с последовательным устройством при больших значениях n время обработки уменьшится приблизительно в m раз.

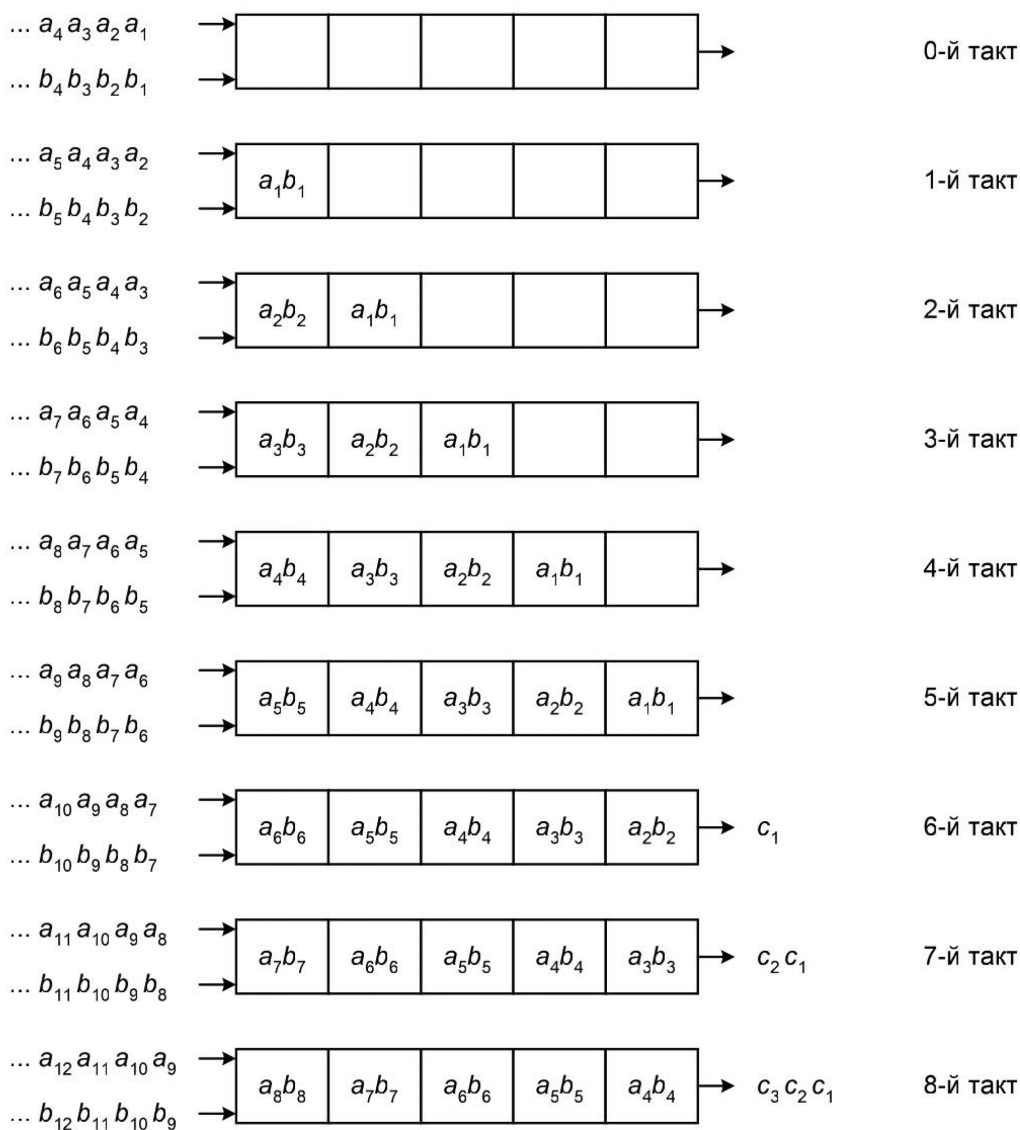


Рис. 5. Суммирование векторов C = A + B с помощью конвейерного устройства. Каждая из пяти ступеней конвейера срабатывает за один такт.

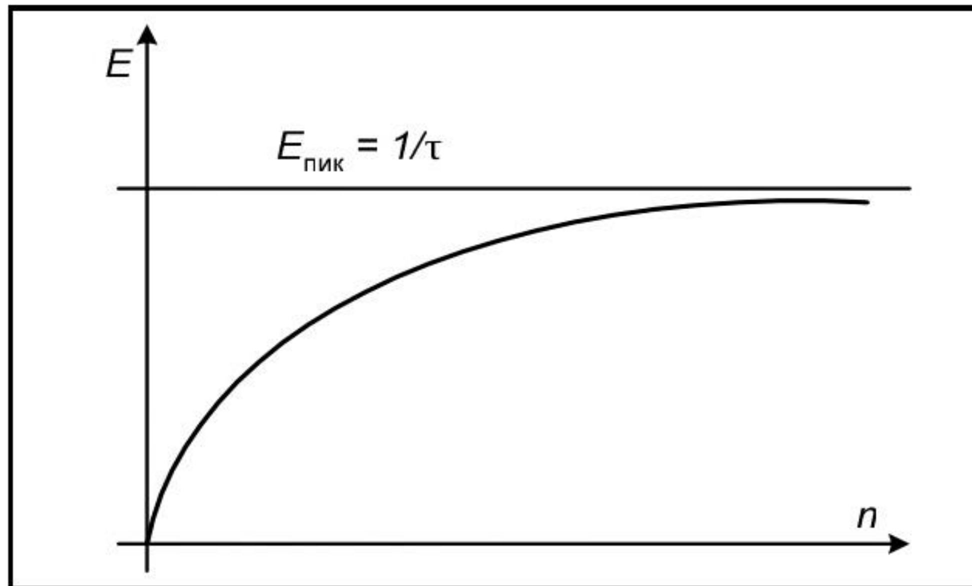


Рис. 6. Кривая зависимости производительности конвейерного устройства от длины входных данных

При возрастании числа входных данных реальная производительность E конвейерного устройства приближается к пиковой (рис.6). τ - время такта работы компьютера.

Рассмотрим оптимизацию распределения данных. Одним из способов такой оптимизации является иерархическая организация памяти. Часто используемые данные оптимально хранить на регистрах. Скорость работы процессора всегда согласуется со временем выполнения операций на регистрах. Но в случае увеличения объемов данных привлекается уровень кэш-памяти. Далее, в случае повторения ситуации, привлекаются следующие уровни памяти (разные уровни кэш-памяти, оперативная, дисковая и т. д.). В связи с высокой стоимостью, объем высших уровней иерархии ограничен.

Выполнение стандартных вычислений зависит от времени обращения к запоминающему устройству. Использование сверхбыстродействующей дополнительной памяти небольшой емкости позволяет уменьшить это время. Например, цикл с телом из одного оператора и большим числом итераций является фрагментом программы с высокой локальностью вычислений. В этой связи тело цикла помещается в более быструю память. Такой шаг

становится возможным, так как каждая итерация выполняет одни и те же команды.

Помимо цикла существуют следующие причины снижения степени локальности (данных и вычислений) в программах:

- вызов подпрограмм и функций,
- косвенную адресацию массивов,
- неудачную работу с многомерными массивами
- сложные структуры данных,
- использование условных операторов и т.п.

Использование иерархии памяти позволяет повысить производительность вычислительных устройств.

Еще одним способом повышения производительности современных компьютеров является организация параллельной работы нескольких машин. Подобная возможность обеспечивается общим дополнительным устройством управления и передачей кодов чисел с одной машины на другую. В вычислительной системе существует общая основная память, в которой хранятся числа и команды, обеспечивающие решение задачи. Данная информация поступает на ряд сверхбыстродействующих запоминающих устройств небольшой емкости. Каждое устройство имеет свое АЛУ и индивидуальное управление.

При расчете производительности следует учитывать необходимость согласования скорости работы процессора со временем выборки данных из памяти. В качестве решения проблемы использовалось расслоение памяти, развитая регистровая структура, многоуровневая память, кэш-память и т.д.

Для оценки времени выполнения программ используют свойства локальности вычислений и локальности использования данных.

Таким образом, использование иерархии памяти является одним из способов повышения производительности. В целом производительность программы зависит от:

- поддержки параллелизма в аппаратно-программной среде вычислительной системы, а именно от его использования в операционной системе, компиляторе;
- технологии параллельного программирования и системы времени исполнения программ, поддержке параллелизма в процессоре и особенности работы с памятью.

Задания [1, с. 59]

1. Каким соотношением связаны между собой время такта и тактовая частота компьютера?
2. Если время такта компьютера равно 2,5 нс, и за каждый такт он может выполнять две операции, чему равна пиковая производительность этого компьютера?
3. Конвейерное устройство состоит из пяти ступеней. Времена срабатываний ступеней равны 1, 1, 2, 1 и 3 такта соответственно. С какой максимальной частотой на выходе данного устройства будут появляться результаты, если на его вход аргументы поступают без перебоев?
4. Меняется ли ответ задачи 3 в зависимости от того, в каком порядке идут ступени в таком устройстве?
5. За какое минимальное число тактов может быть выполнено 70 операций, если в распоряжении есть устройство, описанное в задаче 3?
6. В компьютере есть 7 параллельно работающих устройств, каждое из которых может выполнить операцию за 7 единиц времени. За какое минимальное время этот компьютер обработает 7 независимых операций?
7. Конвейерное устройство состоит из *к* ступеней, срабатывающих за n_1, n_2, .. n_k тактов соответственно. За какое минимальное число тактов может быть выполнено m операций на таком устройстве?

8. В системе есть два универсальных процессора, выполняющих как операцию сложения, так и операцию умножения за один такт. Предположим, что операции чтения/записи данных происходят мгновенно и всегда есть место для сохранения промежуточных результатов. За какое минимальное число тактов в такой системе может быть выполнен следующий фрагмент программы?

a = c * d+ e b = v*t + u f= x * y + z

9. Почему пиковой производительности конвейерного компьютера нельзя точно достичь на практике?
10. Какие уровни в иерархии памяти используются в современных компьютерах? Каковы их объемы?

Глава 4. Способы повышения эффективности работы программ

Возможны следующие способы повышения эффективности работы программ: использование спецпроцессоров, различные способы оптимизации архитектуры многопроцессорных конфигураций.

4.1. Разработка спецпроцессоров

Спецпроцессоры находят применение при решении таких практических задач, как распознавании речи, анализ изображений, обработка сейсмологических данных, обработка сигналов, и т. д. Сопроцессоры объединяют от сотен до десятков тысяч элементарных функциональных устройств в процессе их параллельной работы, что обеспечивает высочайшую скорость вычислений. Необходимость разработки спецпроцессоров обусловлена сложностью вычислений при решении задач на параллельных компьютерах. Подобные вычисления используют вещественные числа с плавающей точкой. Единица производительности параллельного компьютера - **флопс** определяется количеством арифметических операций в секунду, выполненных над вещественными числами с плавающей точкой.

При разработке спецпроцессоров используется **параллелизм на уровне команд,** который рассматривался выше.

Рассмотрим два подхода к построению архитектуры процессоров, реализующих параллелизм на уровне машинных команд: суперскалярные процессоры и VLIW-процессоры.

Суперскалярные процессоры

При таком подходе задача обнаружения параллелизма в машинном коде и определение последовательности исполнения команд решается на аппаратном уровне. Программа для суперскалярного процессора не содержит информацию о параллелизме.

VLIW-процессоры (Very Large Instruction Word) работают по правилам фоннеймановского компьютера. Команда, исполняемая процессором, определяет несколько операций. Каждое из полей команды отвечает за реализацию одной из стандартных операций процессора: работу с памятью, работу регистрами, работу с функциональными устройствами и т.д. Программа для процессора всегда содержит информацию о параллелизме.

Оба подхода относятся к увеличению производительности отдельных процессоров, на основе которых, строятся многопроцессорные конфигурации.

4. 2. Улучшение архитектуры многопроцессорных конфигураций

В зависимости от способа организации взаимодействия множества процессоров с памятью можно выделить два типа компьютеров:

1. **Компьютеры с общей памятью - мультипроцессорные системы** или мультипроцессоры - SMP-класс (Symmetric Multy Processors/ Shared Memory Processors).

Эти системы объединяют несколько процессоров с равными правами и равноправным доступом к общей памяти. Все процессоры используют общую память, работают с единым адресным пространством. Эти компьютеры решают задачу параллельных вычислений. Но существуют следующие проблемы: вопросы эффективного использования таких систем,

расходы на организацию взаимодействия параллельно работающих процессоров, упрощение разработки параллельных программ. Единственным способом программирования подобных систем является обмен сообщениями, например (PVM, MPI). Мультипроцессор представлен на рис. 7.

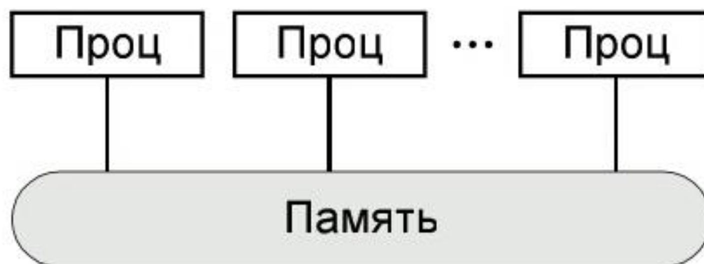


Рис. 7. Мультипроцессор

2. **Компьютеры с распределенной памятью - мультикомпьютерные системы**

Вычислительный узел представляет собой вычислительное устройство, включающее процессор, память, подсистему ввода/вывода, операционную систему. Компьютеры с распределенной памятью позволяют строить системы с максимальной производительностью. Эффективная работа таких систем зависит от качества программного обеспечения. Разработка эффективных программ является одной из основных задач параллельных вычислений. Мультикомпьютер представлен на рис. 8.

Можно выделить проблему, общую для систем обоих типов – система коммутации процессора с модулями памяти и процессоров между собой. Рассмотрим возможные способы организации коммуникационных систем.

Рис. 8. Параллельные компьютеры с распределенной памятью

4.3. Способы организации коммуникационных систем в суперкомпьютерах

1. Использование общей шины

При таком способе к шине подключаются процессоры и память (рис. 9). Передача адресов, данных и управляющих сигналов между процессорами и памятью осуществляется через линии, из которых состоит шина. Для предотвращения одновременного обращения нескольких процессоров к памяти, устройству, захватившее шину, обеспечивается монопольное владение шиной. Увеличение числа устройств в подобных системах может сказаться на производительность системы. Программное обеспечение для рассматриваемых систем отличается невысокой сложностью по сравнению с программным обеспечением для систем с распределенной памятью.

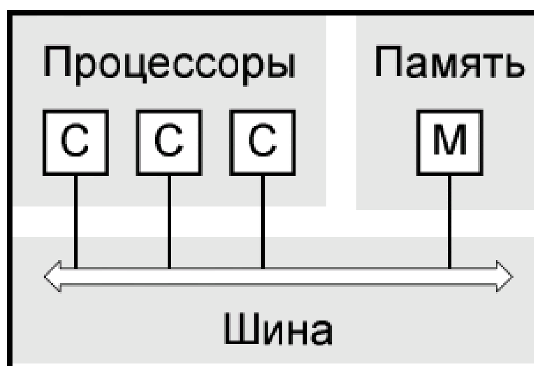


Рис. 9. Использование общей шины

2. Разделение памяти на независимые модули

Данный способ обеспечивает одновременный доступ разных процессоров к различным модулям (рис. 10). Такая возможность появляется за счет:

- матричного коммутатора;
- каскадных переключателей;

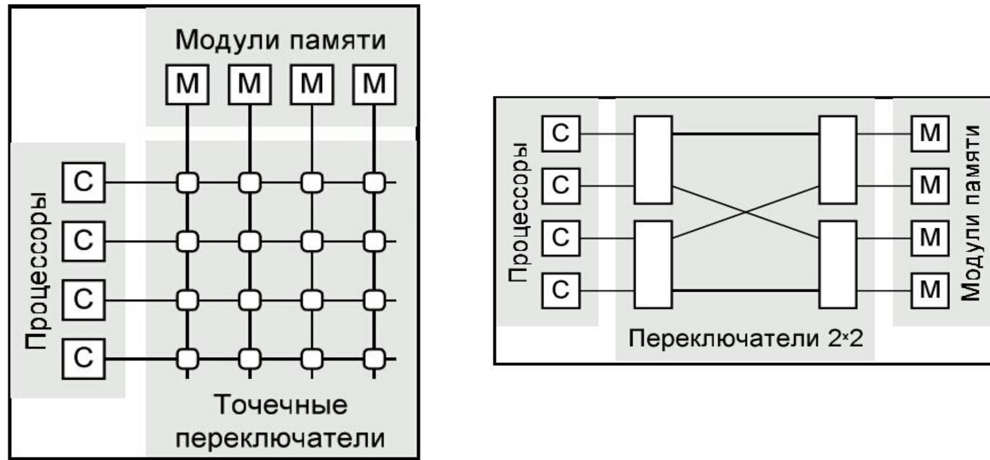


Рис. 10. Разделение памяти: а) мультипроцессор с матричным коммутатором, б) мультипроцессор с каскадным переключателем

Ошибка!

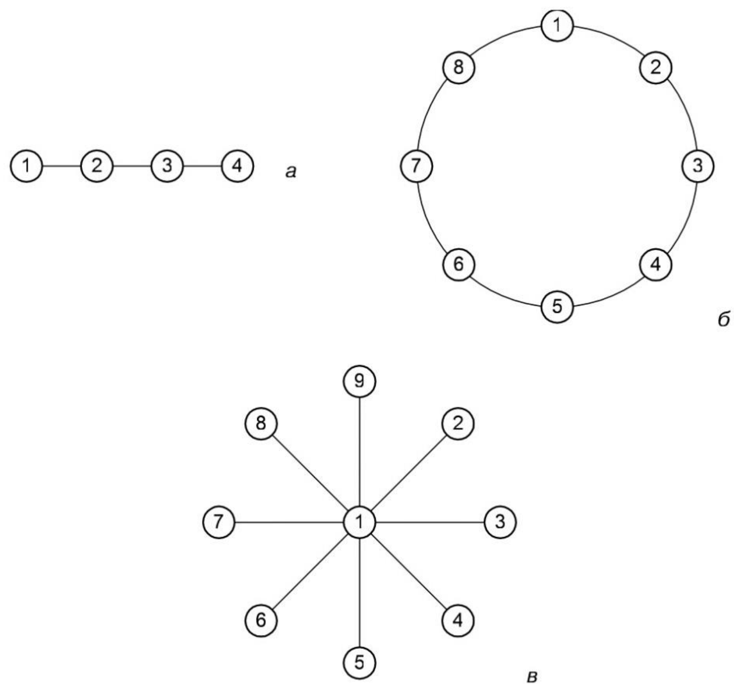


Рис. 11.Мультикомпьютеры с топологиями связи: а) линейка, б) кольцо, в) звезда

Возможные топологии соединения вычислительных узлов в мультикомпьютерных системах представлены на рис. 11.

Выбор топологии связи процессоров в конкретной вычислительной системе обусловлен разными причинами: стоимость, технологическая реализуемость, простота сборки и программирования, надежность, минимальность средней длины пути между узлами, наименьшее из максимальных расстояний между узлами и т.д. Варианты представлены на рис. 12.

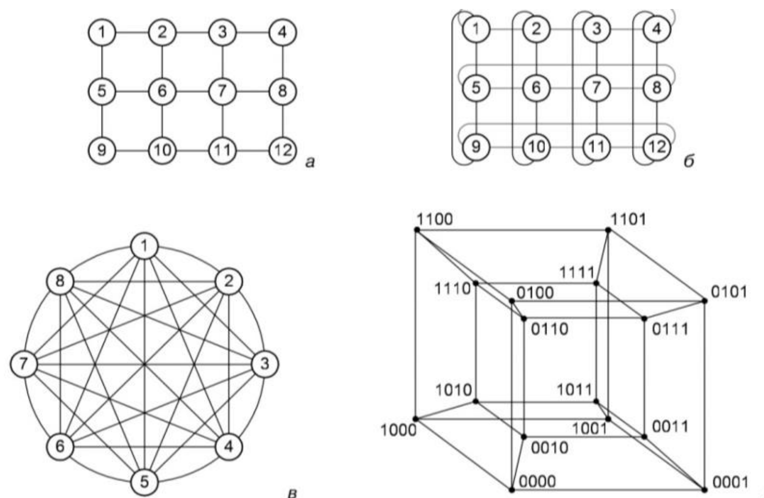


Рис. 12. Варианты топологий связи процессоров: а) решетка, б) 2-тор,
в) полная связь, г) гиперкуб

4.4. Компьютеры с неоднородным доступом к памяти компьютеры с архитектурой NUMA (Non Uniform Memory Access)

Рассматриваемая архитектура позволяет использовать достоинства как систем с общей памятью, так и систем с распределенной памятью.

Компьютер данного типа представляет собой множество соединенных через межкластерную шину кластеров. Кластер обычно состоит из процессора, контроллера памяти, модуля памяти, но может включать устройства ввода/вывода и соединяющую их локальную шину. Когда процессору нужно выполнить операции чтения или записи, он посылает запрос с нужным адресом своему контроллеру памяти. Контроллер

анализирует старшие разряды адреса, по которым и определяет, в каком модуле хранятся нужные данные. Если адрес локальный, то запрос выставляется на локальную шину, в противном случае запрос для удаленного кластера отправляется через межкластерную шину. В таком режиме программа, хранящаяся в одном модуле памяти, может выполняться любым процессором системы. Единственное различие заключается в скорости выполнения. Все локальные ссылки отрабатываются намного быстрее, чем удаленные. Поэтому и процессор того кластера, где хранится программа, выполнит ее на порядок быстрее, чем любой другой.

Проблема согласования содержимого кэш-памяти – кэш-память, которая помогает значительно ускорить работу отдельных процессоров, для многопроцессорных систем оказывается «узким» местом.

Для решения данной проблемы разработана специальная модификация NUMA-архитектуры — ccNUMA (cach coherent NUMA) – архитектура.

4.5. Программное обеспечение параллельных компьютеров

Программное обеспечение параллельных компьютеров нельзя рассматривать отдельно от аппаратного обеспечения. Разработка эффективного параллельного программного обеспечения является центральной проблемой параллельных вычислений.

Основные подходы

Использования в традиционных языках программирования специальных комментариев, добавляющих "параллельную" специфику в изначально последовательные программы (стандарт OpenMP). Это - спецкомментарии для компилятора. Использование спецкомментариев добавляет возможность параллельного исполнения и полностью сохраняет исходный вариант программы.

Расширение существующих языков программирования.

Механизм предполагает использование дополнительных операторов и новых элементов описания переменных. Это дает пользователю возможность

непосредственно реализовывать параллельные вычисления в программе и управлять ее исполнением. Языки HPF (hight performance Fortran) и mpC позволяют создавать эффективное программное обеспечение для неоднородных вычислительных систем.

Существуют специальные языки параллельного программирования, которые могут точно отразить либо специфику архитектуры параллельных систем, либо свойства какого-то класса задач некоторой предметной области - Occam, Sisal, НОРМА.

Также имеются библиотеки и интерфейсы, поддерживающие взаимодействие параллельных процессов - интерфейс MPI (Message Passing Interface), библиотеки Lapack, ScaLapack, HP Mathematical Library, система Linda.

Существуют специализированные пакеты и программные комплексы, для использования которых пользователю необходимо указывать входные данные с тем, чтобы правильно использовать возможности пакета.

В настоящее время параллельные компьютеры - системы, состоящие из сотен и тысяч процессоров, каждый из которых – сложная параллельная система.

Задания [1, с. 77]

1. Две вычислительные системы отличаются только версиями установленных на них операционных систем. Могут ли на таких компьютерах отличаться времена работы одной и той же программы?
2. Возможна ли такая ситуация: в программе, содержащей 10 000 строк исходного текста, изменили только один символ, а время ее работы выросло в 10 раз?
3. Возможна ли такая ситуация: в программе, содержащей 10 000 строк исходного текста, изменили только один символ, после

чего производительность, с которой ее выполнял компьютер, выросла в 10 раз?

4. Предположим, что спецпроцессор является матрицей синхронно работающих однобитных процессоров. Какие классы задач можно эффективно решать на таких спецпроцессорах?
5. Какого вида операции должен уметь быстро выполнять спецпроцессор, проектируемый для реализации быстрого преобразования Фурье?
6. Какие еще виды параллелизма на уровне машинных команд, кроме суперскалярности и идей VLIW-обработки, используются в современных микропроцессорах?
7. Известно, что программа хорошо выполняется на суперскалярном процессоре с некоторым набором независимых функциональных устройств. Означает ли это, что та же самая программа будет хорошо выполняться на УУ\У-процессоре, имеющем тот же набор устройств? Верно ли обратное?
8. Компилятор для какого процессора: суперскалярного или VLIW должен быть более "интеллектуальным" для генерации эффективных программ?
9. Почему общую шину не используют для объединения большого числа процессоров?
10. Почему средняя длина пути между двумя узлами системы, в которой n процессоров соединены по топологии "линейка", равна $n/3$?
11. Приведите пример алгоритма, структура связей которого хорошо соответствует топологии "кольцо".

Глава 5. Архитектура параллельных вычислительных систем

Процесс решения задачи на параллельном компьютере состоит из следующих этапов:

1. Формулировка задачи
2. Выбор метода ее решения
3. Составление алгоритма
4. Выбор технологии программирования
5. Создание программы
6. Выполнение.

Несоответствие структуры программы особенностям ее архитектуры приводит производительность к падению производительности.

5.1. Классификация архитектур вычислительных систем

Целью классификации является решение задачи эффективного отображения алгоритмов на архитектуру вычислительных систем.

5.1.1. Классификация М. Флинна

В 1966 году классификация предложена М. Флинном и основана на обрабатываемой процессором информации (команд или данных), называемой потоком. Можно выделить четыре класса архитектур: SISD, SIMD, MIST,MIMD.

1.1. **Класс SISD** (Single Instruction stream/ Single Data stream) - одиночный поток команд и одиночный поток данных.

К классу SISD относятся машины фон-неймановского типа. Такие машины характеризуются одним потоком команд, выполняющихся последовательно. Каждой команде соответствует одна скалярная операция. Конвейерная обработка может применяться для увеличения скорости обработки команд и скорости выполнения арифметических операций. В этот класс можно включить и векторно-конвейерные машины (вектор - одно

неделимое данное для соответствующей команды) системы Cray-1, CYBER 205, машины семейства FACOM VP и другие.

Иллюстрация класса представлена на рис. 13 а

ПР – это один или несколько процессорных элементов,

УУ – устройство управления,

ПД – память данных.

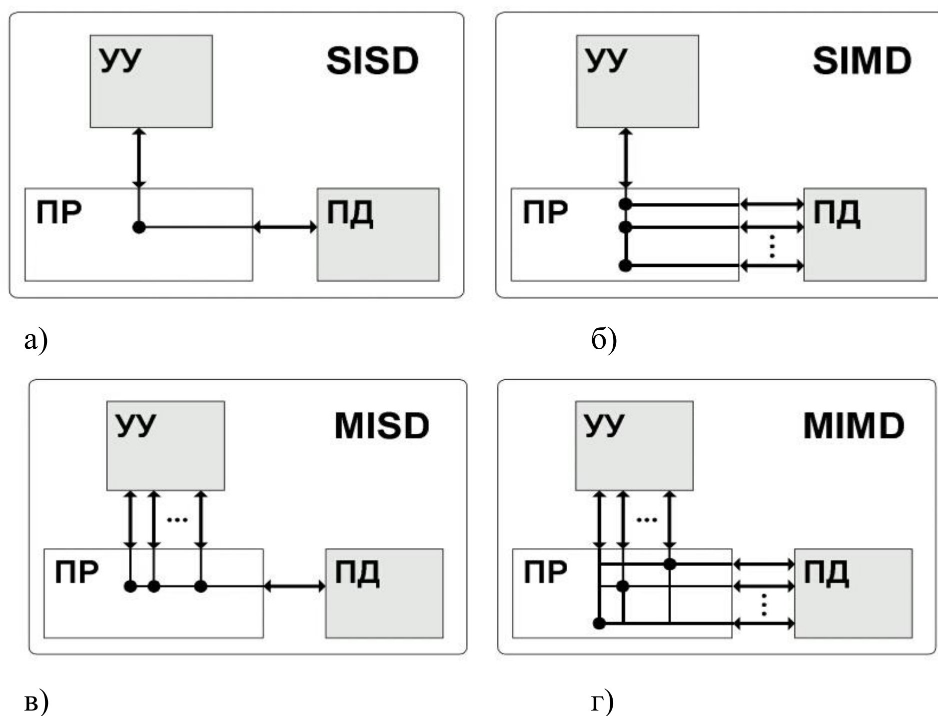


Рис. 13. Классификация М. Флинна: а) класс SISD, б) класс SIMD, в) класс MIST, г) класс MIMD

1.2. **Класс SIMD (**Single Instruction stream/ Multiple Data stream) – одиночный поток команд и множественный поток данных. Имеется один поток команд, включающий векторные команды. Это делает возможным применение одной арифметической операцию к нескольким данными, Примером таких данных могут служить элементы вектора. Обработка элементов вектора может производиться процессорной матрицей (ILLIAC IV) или конвейерно (Cray-1). В данном случае единое УУ осуществляет контроль над множеством процессорных элементов. В фиксированный момент времени каждый процессорный элемент получает от УУ одинаковую

команду, которую выполняет над своими локальными данными. Матрицы процессоров: ILLIAC IV, ICL DAP и т. п. Иллюстрация класса представлена на рис. 13 б.

1.3. **MISD** (Multiple Instruction stream/ Single Data stream)- поток команд и одиночный поток данных. Класс пуст, так как не представлена реально существующая вычислительная система, построенной на данном принципе. Иллюстрация класса представлена на рис. 13 в.

1.4. **MIMD** (Multiple Instruction stream/ Multiple Data stream)- множественный поток команд в сочетании с множественным потоком данных. В вычислительной системе есть несколько устройств обработки команд. Эти команды объединены в один комплекс. Каждое устройство работает с собственным потоком команд и данных. Если конвейерная обработка выполняется над множественным скалярным потоком, то в данный класс можно включить векторно-конвейерные компьютеры. Мультипроцессорные системы: Cm*, C.mmp, Cray Y-MP, Denelcor HEP, BBN Butterfly, Intel Paragon, Cray T3D. Иллюстрация класса представлена на рис. 13 г.

Достоинство

Данная классификация дает четкое понятие об архитектуре компьютера, отнесенного к конкретному классу.

Недостатки

Некоторые архитектуры, такие как dataflow и векторно-конвейерные машины, четко не вписываются в данную классификацию.

Часто в один класс, по классификации Флинна, попадают вычислительные системы, отличающиеся аппаратной и программной реализацией (например, MIMD). Таким образом, следует рассмотреть более детальные классификации.

5.1.2. Классификация Р. Хокни

Более детальная систематизация компьютеров, попадающих в класс MIMD по систематике М. Флинна. Возможны следующие способы обработки множественного потока команд: либо одно конвейерное устройство работает в режиме разделения времени для отдельных потоков, либо каждый поток обрабатывается отдельным устройством. Первая возможность используется в MIMD-компьютерах (процессорные модули в Denelcor HEP или компьютеры семейства Tera MTA). Архитектуры, использующие второй способ, делятся на два класса:

- MIMD-компьютеры, в которых процессоры могут напрямую связываться друг с другом при помощи переключателя.

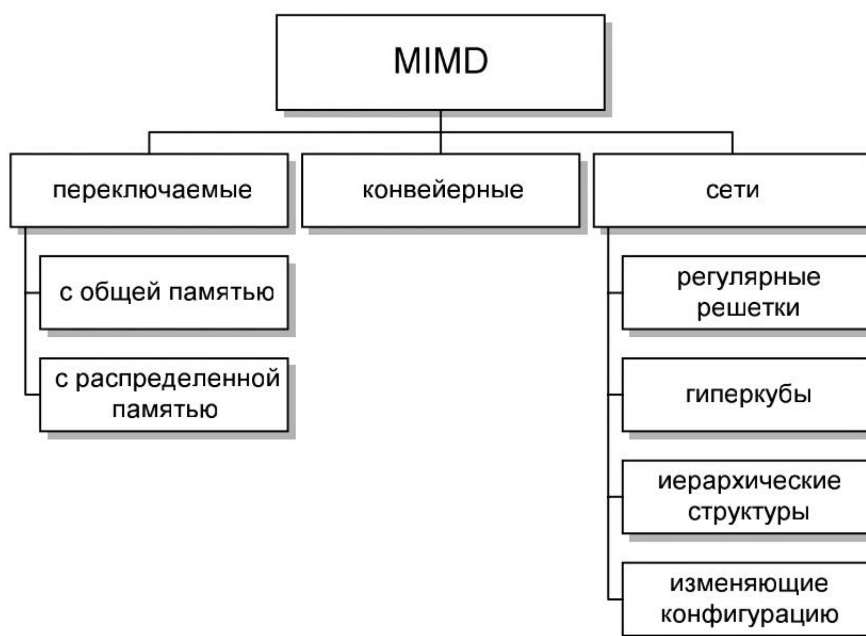


Рис. 14. Классификация Р.Хокни

-

- MIMD-компьютеры, в которых взаимодействие удаленных процессоров поддерживается специальной системой маршрутизации и каждый процессор может напрямую связываться только с ближайшими соседями по сети. Классификация представлена на рис. 14.

5.1.3. Классификация Т. Фенга (1972 г.) осуществляется на основе двух характеристик:

- число *n* бит в машинном слове, обрабатываемых параллельно при выполнении машинных инструкций. Практически во всех современных компьютерах это число совпадает с длиной машинного слова.
- число слов *m*, обрабатываемых одновременно данной вычислительной системой или ширина битового слоя (параллельная обработку *n* битовых слоев, на каждом из которых независимо преобразуются *m* бит.

Максимальной степенью параллелизма вычислительной системы **С, которую можно описать парой чисел (*n, m*)**, (по Фенгу) называется произведение P= n • m, которое определяет интегральную характеристику потенциала параллельности архитектуры. Данное значение есть про производительность, выраженная в других единицах.

По классификации Фенга все вычислительные системы можно разделить на следующие четыре класса:

Разрядно-последовательные, пословно-последовательные (n = m = 1). В каждый момент времени такие компьютеры обрабатывают только один двоичный разряд. Представителем данного класса служит давняя система MINIMA с естественным описанием (1, 1).

Разрядно-параллельные, пословно-последовательные (n > m, m = 1). Большинство классических последовательных компьютеров, так же как и многие вычислительные системы, используемые сейчас, принадлежат к данному классу: IBM 701 с описанием (36, 1), PDP-11 с описанием (16, 1), IBM 360/50 и VAX 11/780 — обе с описанием (32, 1).

Разрядно-последовательные, пословно-параллельные (n = 1, m > 1). Такие системы состоят из множества одноразрядных процессорных элементов. Обработка данных каждым процессором осуществляется независимо процессорный элемент обрабатывать свои данные. Типичными примерами служат STARAN (1, 256) и MPP (1, 16384) фирмы Goodyear

Aerospace, прототип известной системы SOLOMON (1, 1024) и ICL DAP (1, 4096).

Разрядно-параллельные, пословно-параллельные ($n > 1$, $m > 1$). Такие системы, как ILLIAC IV (64, 64), TI ASC (64, 32), CDC 6600 (60, 10), BBN Butterfly GP1000 (32, 256) способны обрабатывать одновременно $m \cdot n$ двоичных разрядов. Такие системы принадлежат именно к этому классу.

Достоинство - введение единой числовой метрики для всех типов компьютеров, позволяющей сравнить любые два компьютера между собой.

Недостаток: В пределах классификации трудно осознать специфику той или иной вычислительной системы.

При вычислении ширины битового слоя не делается никакого различия между процессорными матрицами, векторно-конвейерными и многопроцессорными системами. Нет информации об аппаратном обеспечении процессов параллельной обработки информации (конвейерность, набор функциональных устройств, набор независимых процессоров). В системе из N независимых процессоров, каждый их которых содержит P конвейерных функциональных устройств с длиной конвейера h, ширина битового слоя составит $N \cdot p \cdot h$.

5.1.4. Классификация В. Хендлера

Данная классификация использует различия между тремя уровнями обработки данных в процессе выполнения программ:

- уровень выполнения программы, на котором устройство управления (УУ) производит выборку и дешифрацию команд программы. Механизм предполагает использование счетчика команд и других регистров;
- уровень выполнения команд, на котором АЛУ исполняет выданную устройством управления команду;
- уровень битовой обработки, на котором происходит разбиение на группы все элементарных логических схем процессора, которые используются для выполнения операций над одним двоичным разрядом.

Вычислительная система содержит какое-то число процессоров, каждый со своим устройством управления. Число УУ – k. Каждое устройство управления связано с несколькими арифметико-логическими устройствами (d), исполняющими одну и ту же операцию в кажд9ый конкретный момент времени. Каждое АЛУ объединяет несколько групп элементарных логических схем (ЭЛС), ассоциированных с обработкой одного двоичного разряда. Число групп ЭЛС (w) - длина машинного слова. Без учета возможности конвейеризации систему **C** можно рассматривать как тройку C=<k, d, w>.

5.1.5. Классификация Л. Шнайдера (1988 г.)

Классификация Л. Шнайдера основана на разделении этапов выборки и непосредственно исполнения в потоках команд и данных (рис. 15). Это позволило описать такие архитектуры, как компьютеры с длинным командным словом, систолические массивы и т.д. Можно выделить следующие классы:

1. $I_{ss}D_{ss}$ — классические машины фоннеймановского типа;
2. $I_{ss}D_{sc}$ — фоннеймановские машины, в которых заложена возможность выбирать данные, расположенные с разным смещением относительно одного и того же адреса и над которыми будет выполнена одна и та же операция. Примером могут служить компьютеры, имеющие команды типа одновременного выполнения двух операций сложения над данными в формате полуслова, расположенными по указанному адресу;
3. $I_{ss}D_{sm}$ – SIMD-компьютеры, в которых отсутствует возможность получения уникального адреса для данных в процессорных элементах;
4. $I_{ss}D_{mm}$ – SIMD-компьютеры с возможность. независимой модификации адресов операндов в каждом процессорном элементе;
5. $I_{sc}D_{cc}$ – вычислительные системы, обрабатывающие одновременно несколько команд, доступ к которым осуществляется по одному адресу.

6. $I_{mm}D_{mm}$ – класс, включающий все MIMD-компьютеры.

В пределах классификации *c, s, m* - описатели архитектуры реальных компьютеров.

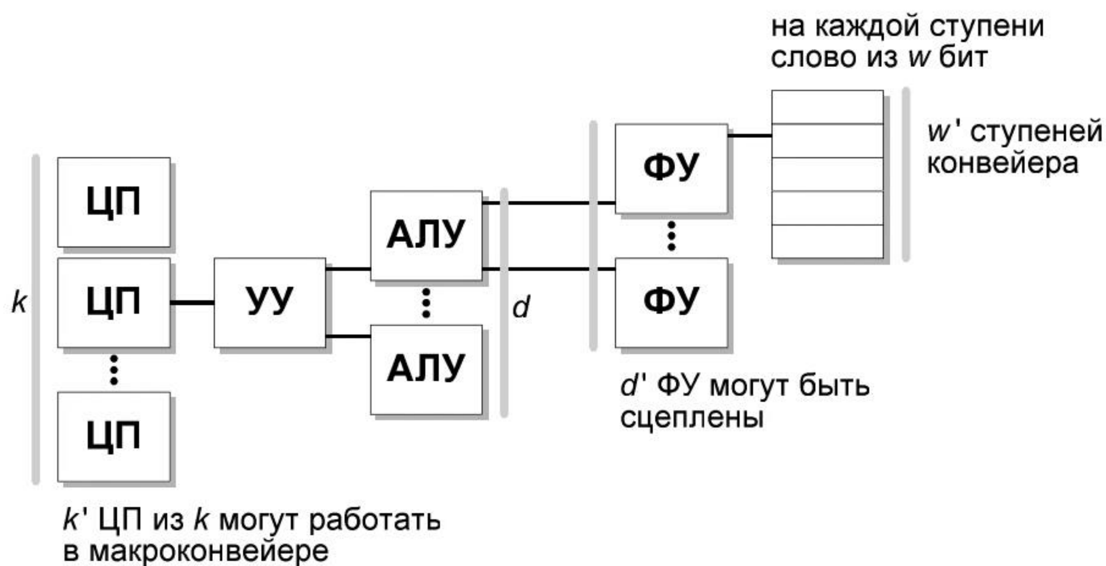


Рис. 15. Классификация Л. Шнайдера

Достоинства:

- тщательная систематизация SIMD-компьютеров;
- возможность описания нетрадиционных архитектур типа систолических массивов или компьютеров с длинным командным словом.

Недостаток: критерий классификации, основанный на потоках команд и данных без учета распределенности памяти и топологии межпроцессорной связи, приводит к попаданию в один класс вычислительные системы типа М1МБ.

5.1.6. Классификация Д. Скилликорна (1989 г.)

Является подходом для описания свойств многопроцессорных систем и некоторых нетрадиционных архитектур, в частности, dataflow (рис. 16). Архитектура компьютера рассматривается как абстрактная структура, состоящая из четырех компонентов:

Процессор команд (IP – Instruction Processor) – функциональное устройство, работающее как интерпретатор команд; в системе, может отсутствовать;

Процессор данных (DP – Data Processor) – функциональное устройство, работающее как преобразователь данных в соответствии с арифметическими операциями;

Иерархия памяти (IM – Instruction Memory, DM – Data Memory) - запоминающее устройство, в котором хранятся данные и команды, пересылаемые между процессорами;

Переключатель – абстрактное устройство, обеспечивающее связь между процессорами и памятью.

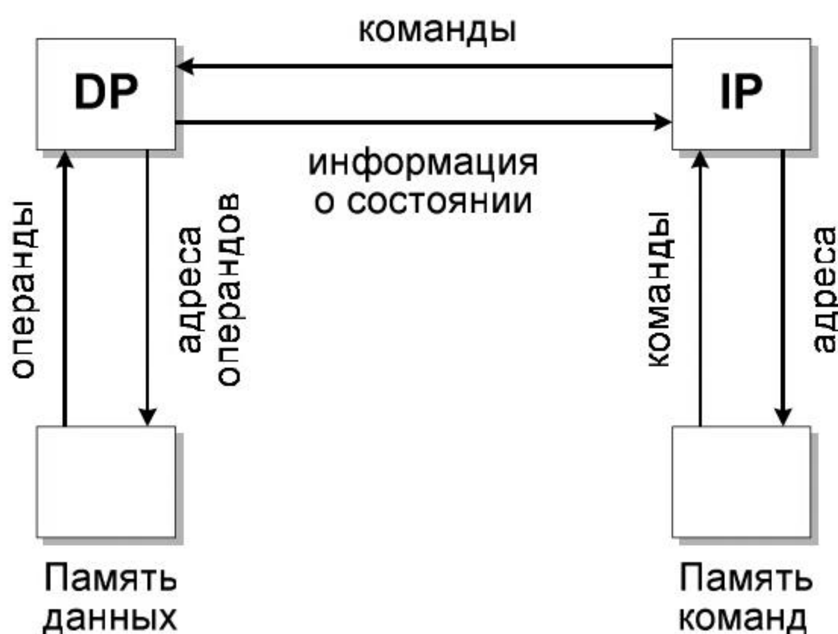


Рис. 16. Классификация Д. Скилликорна

Для описания сложных параллельных вычислительных систем Д. Скилликорн зафиксировал четыре типа переключателей для соединения устройств:

1. 1-1 - переключатель такого типа связывает пару функциональных устройств.
2. n-n - переключатель связывает каждое устройство из одного множества устройств с соответствующим ему устройством из другого множества, т. е. фиксирует попарную связь.
3. 1-n - переключатель соединяет одно выделенное устройство со всеми функциональными устройствами из некоторого набора.
4. n • n - каждое функциональное устройство одного множества можетбыть связано с любым устройством другого множества, и наоборот.

Классификация Д. Скилликорна строится на основе следующих восьми характеристик:

- количество процессоров команд IP;
- число запоминающих устройств (модулей памяти) команд IM;
- тип переключателя между IP и IM;
- количество процессоров данных DP;
- число запоминающих устройств (модулей памяти) данных DM;
- тип переключателя между DP и DM;
- тип переключателя между IP и DP;
- тип переключателя между DP и DP.

На рис. 17 представлен компьютер Connection Machine 2, который можно описать в рамках классификации: (1,1, 1 - 1, n, n, n - n, 1 - n, n x n).

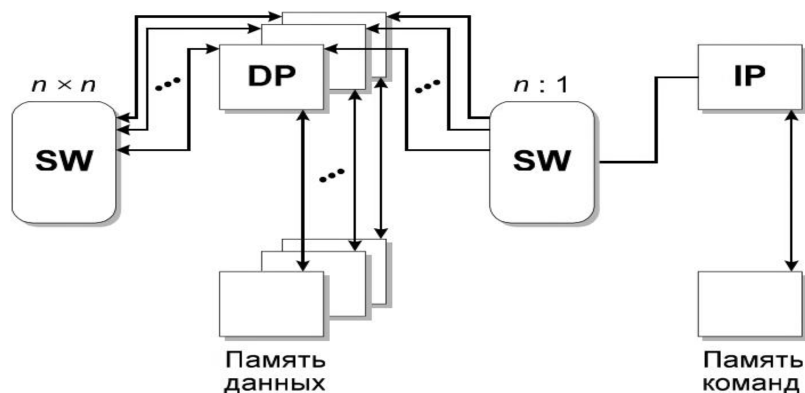


Рис. 17. Компьютер Connection Machine 2

5.2. Векторно-конвейерные компьютеры (архитектура векторно-конвейерных супер-ЭВМ CRAY C90)

Первый компьютер Cray-1 появился в 1976 году. Рассмотрим архитектуру Cray C90, представленную на рис.18.

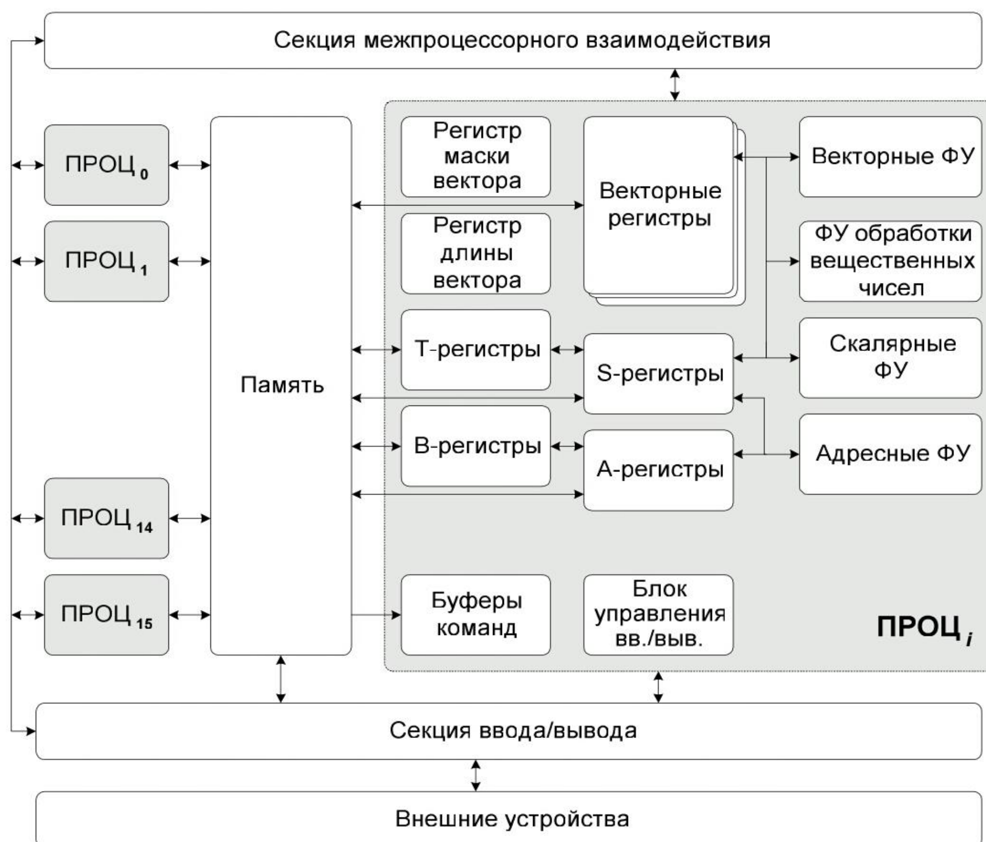


Рис. 18. Общая схема компьютера Cray C90

Оперативная память

В Cray C90 оперативная память разделяется всеми процессорами и секцией ввода/вывода. Слово оперативной памяти состоит из 80-ти разрядов: 64 разряда отводится для хранения данных и 16 вспомогательных разрядов используется для коррекции ошибок. Память разделена на банки, которые могут работать одновременно.

Оперативная память связана с процессором через четыре порта. Каждый порт за один такт обрабатывает два слова. Из четырех портов один связан с секцией ввода/вывода, один выполняет операцию записи. Данная архитектура обеспечивает выполнение векторных операций с не более чем двумя входными векторами.

В максимальной конфигурации реализовано *расслоение* памяти (рис. 19) компьютера на 1024 банка: каждая из 8 секций разделена на 8 подсекций, а каждая подсекция на 16 банков. Последовательные адреса идут с чередованием по каждому из данных параметров [2]:

адрес 0 - в 0-й секции, 0-й подсекции, 0-м банке;

адрес 1 - е 1-й секции, 0-й подсекции, 0-м банке;

адрес 2- е 2-й секции, 0-й подсекции, 0-м банке;

...

адрес 8 — в 0-й секции, 1-й подсекции, 0-м банке;

адрес 9 — в 1-й секции, 1-й подсекции, 0-м банке;

...

адрес 63 — в 7-й секции, 7-й подсекции, 0-м банке;

адрес 64 — в 0-й секции, 0-й подсекции, 1-м банке;

адрес 65 — в 1-й секции, 0-й подсекции, 1-м банке;

Данная память способна быстро обрабатывать стандартные вычислительные задачи.

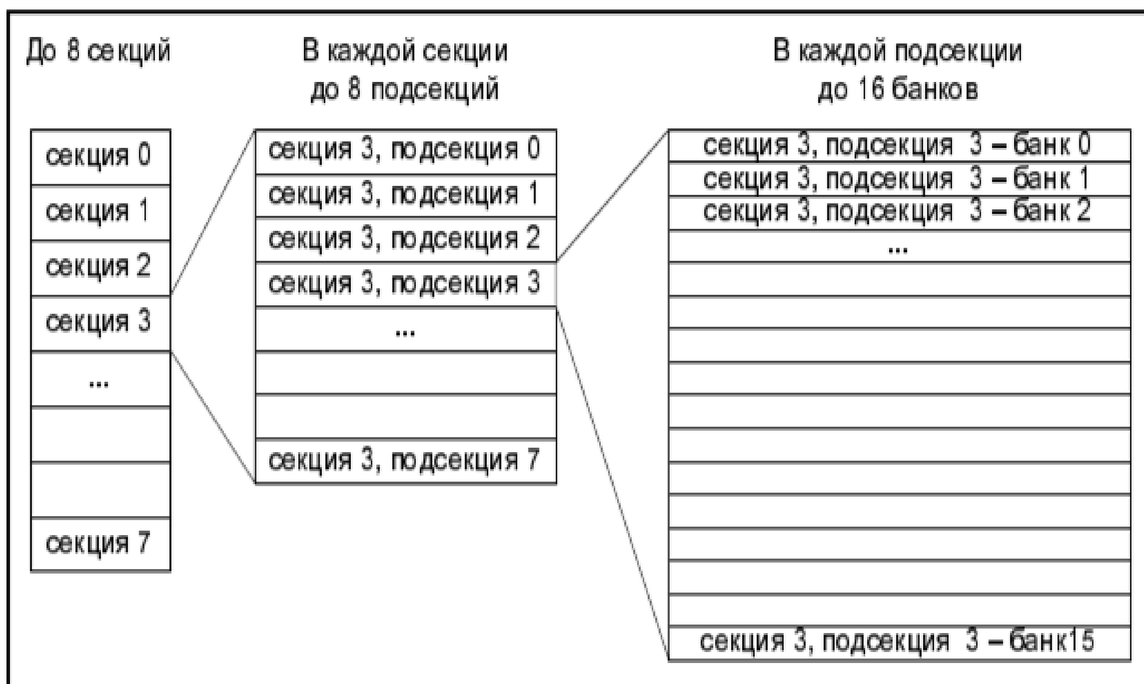


Рис. 19. Расслоение памяти компьютера Cray C90

Секция ввода/вывода

Секция ввода/вывода поддерживает три типа каналов для работы с внешними устройствами, которые различаются скоростью передачи данных:

Low-speed(LOSP) channels — 6 Мбайт/с;

High-speed(HISP) channels — 200 Мбайт/с;

Very high-speed (VHISP) channels — 1800 Мбайт/с.

Секция межпроцессорного взаимодействия осуществляет передачу данных и управляющей информации между процессорами для синхронизации их совместной работы организации взаимодействия друг с другом. Секция межпроцессорного взаимодействия содержит разделяемые регистры и семафоры, объединенные в одинаковые группы — кластеры. Каждый кластер состоит из восьми 32-разрядных разделяемых адресных регистров (SB), восьми 64-разрядных скалярных регистров (ST) и 32

однобитовых семафоров. Число кластеров в системе определяется конфигурацией компьютера.

Все процессоры имеют одинаковую вычислительную секцию, состоящую из регистров и функциональных устройств (ФУ). Различные регистры и функциональные устройства могут хранить и обрабатывать три класса данных: адреса, скаляры и векторы.

Регистровая структура процессора

Каждый процессор имеет набор основных (адресные регистры A, скалярные регистры S и векторные регистры V) и набор промежуточных регистров B и T выполняющие роль промежуточного хранилища между памятью и основными регистрами, взаимодействующими с памятью и функциональными устройствами. Адресные и скалярные регистры взаимодействуют с промежуточными регистрами.

В структуре компьютера предусмотрено 8 адресных регистров в основном наборе A, и 64 регистра в промежуточном наборе B. **Адресные регистры** предназначены для хранения и вычисления адресов, индексации, указания величины сдвигов, числа итераций циклов и т. д. Все регистры данной группы имеют по 32 разряда.

Скалярные регистры имеют 64 разряда и сохраняют аргументы, результаты арифметических операций, данные для векторных команд.

Векторные регистры выполняют векторные команды. В процессоре - 8 векторных регистров, размером до 128 слов. Размер слова составляет 64 разряда.

Функциональные устройства

Все функциональные устройства – конвейерные. Они способны выдавать результат вычислений за один такт. Различают четыре группы данных устройств: адресные, скалярные, векторные и функциональные устройства, выполняющие операции над вещественными числами.

Секция управления процессора

Команды выбираются из оперативной памяти блоками и заносятся в буфера команд, откуда затем они выбираются для исполнения. Если необходимой для исполнения команды нет в текущем буфере, она ищется в других буферах. Если требуемой команды в буферах не оказалось, то происходит выборка очередного блока.

Особенности архитектуры:

Конвейеризация выполнения команд. Конвейерные команды обеспечивают выполнение основных операций процессора: обращение в память, обработка команд и выполнение инструкций функциональными устройствами.

Независимость функциональных устройств. Функциональные устройства в Cray C90 являются независимыми, поэтому несколько операций могут выполняться одновременно.

Векторная обработка. Векторная обработка увеличивает скорость и эффективность обработки за счет того, что обработка целого набора (вектора) данных выполняется одной командой. Скорость выполнения операций в векторном режиме может быть в 10—15 раз выше скорости скалярной обработки.

Зацепление функциональных устройств. Возможность выполнения нескольких векторных операций в режиме "макроконвейера" дает дополнительный выигрыш в скорости их обработки.

Многопроцессорная обработка. В максимальной конфигурации компьютер может содержать до 16 независимых процессоров. Эти процессоры могут быть использованы по-разному. В частности, они могут выполнять несколько независимых программ, но могут и все быть назначены на выполнение одной программы.

Векторно-конвейерные компьютеры Cray занимают первые места по простоте достижения относительно высокой реальной производительности в сравнении с другими архитектурами.

5.3. Параллельные компьютеры с общей памятью (HP Superdom)

Компьютер появился в 2000 году

Количество процессоров в HP Superdom (стандартная комплектация): от 2 до 64 процессоров с возможностью последующего расширения системы. Все процессоры имеют доступ к общей памяти, организованной в соответствии с архитектурой ccNUMA. Все процессы могут работать в едином адресном пространстве, адресуя любой байт памяти посредством обычных операций чтения/записи. Доступ к локальной памяти в системе осуществляется быстрее, чем доступ к удаленной памяти. Проблемы возможного несоответствия данных, вызванные кэш-памятью процессоров, решены на уровне аппаратуры. В архитектуре компьютера могут использоваться несколько типов микропроцессоров. Основа архитектуры - вычислительные ячейки, связанные иерархической системой переключателей. Каждая ячейка является симметричным мультипроцессором (рис. 20), реализованным на одной плате, в котором есть компоненты:

- процессоры (до 4-х);
- оперативная память (до 16 Гбайт);
- контроллер ячейки;
- преобразователи питания;
- связь с подсистемой ввода/вывода (опционально).

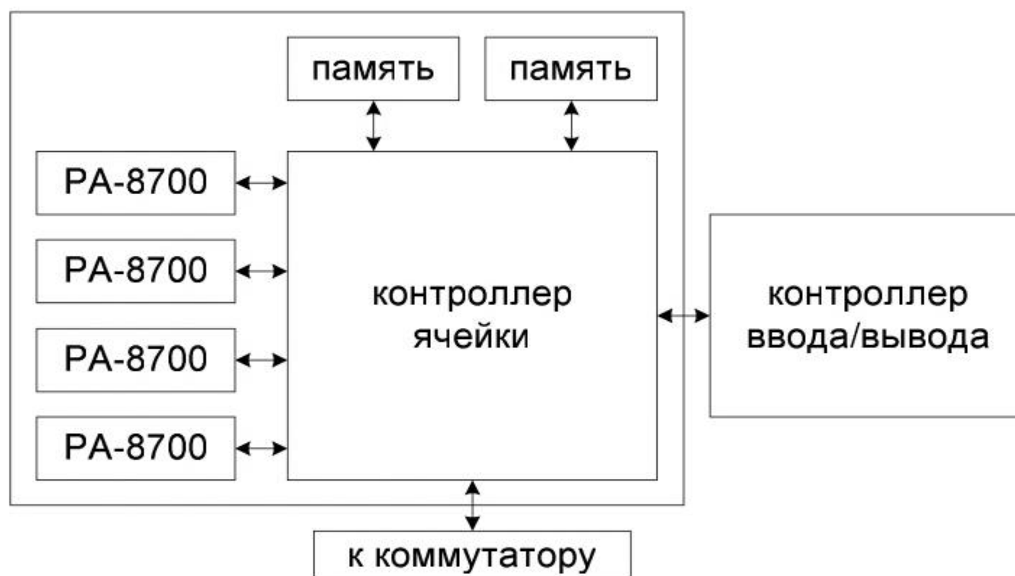


Рис. 20. Структура ячейки компьютера HP Superdom

В компьютере HP Superdom возможны три вида задержек (разница во времени при обращении процессора к локальным и удаленным ячейкам памяти) при обращении процессора к памяти:

- процессор и память располагаются в одной ячейке (задержка минимальна;
- процессор и память располагаются в разных ячейках, но обе эти ячейки подсоединены к одному и тому же коммутатору;
- процессор и память располагаются в разных ячейках, причем обе эти ячейки подсоединены к разным коммутаторам; в этом случае запрос должен пройти через два коммутатора и задержки будут максимальными.

5.4. Вычислительные системы с распределенной памятью (Cray T3D/T3E)

Сгау T3D/T3E – это компьютеры с массовым параллелизмом или массивно-параллельные компьютеры с распределенной памятью. Максимальная конфигурация предполагает использование более 2000 процессоров. Основные компоненты Сгау T3D/T3E содержат два основных компонента: узлы и коммуникационную среду.

В подобных системах некоторое количество вычислительных узлов, которые объединяются друг с другом некоторой коммуникационной средой. Каждый вычислительный узел имеет один или несколько процессоров и локальную память, которую эти процессоры используют. Каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Обращения к памяти других узлов организованы другим образом. Узлы являются полнофункциональными компьютерами. Количество узлов определяется конфигурации системы.

Различают следующие типы узлов:

1. Управляющие узлы. Эти узлы работают в многопользовательском режиме и на них выполняются однопроцессорные программы. Также на этих узлах работают командные файлы.
2. Узлы операционной системы. Такие узлы поддерживают выполнение многих системных сервисных функций ОС.
3. Вычислительные узлы. Эти узлы отвечают за выполнение программы пользователя в монопольном режиме.

В состав узла входят процессорный элемент и сетевой интерфейс. Процессорный элемент состоит из:

- процессора Alpha,
- локальной памяти, которая является частью логически разделяемой распределенной памяти компьютера,
- вспомогательных подсистем.

Сетевой интерфейс взаимодействует с определенным сетевым маршрутизатором в составе коммуникационной сети. На рис. 21 представлена трехмерная целочисленная прямоугольная решетка. В узлах решетки расположены маршрутизаторы. Для соединения маршрутизаторов используется топологи трехмерного тора (рис. 21), что позволяет:

- выбрать альтернативный маршрут для обхода поврежденных связей;

- обеспечить быструю связь граничных узлов и небольшое среднее число перемещений по тору при взаимодействии разных ПЭ.

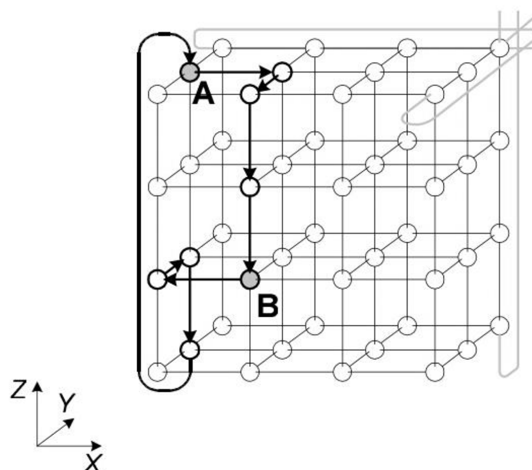


Рис. 21. Коммуникационная решетка компьютера Cray T3E

Архитектура данного компьютера поддерживает барьерную синхронизацию. Барьером является точка в программе, в которой процесс ожидает подхода к барьеру остальных процессов.

Достоинства рассматриваемой архитектуры: возможность подобрать конфигурацию в зависимости от имеющегося бюджета и своих потребностей в вычислительной мощности; схема позволяет неограниченно наращивать число процессоров в системе и увеличивать ее производительность.

Вычислительные системы с распределенной памятью могут быть использованы в вычислительных кластерах.

5.5. Концепция GRID и метакомпьютинг

Метакомпьютером будем систему, состоящую из компьютеров, объединенных для выполнения параллельных вычислений. Часто в связи с метакомпьютером используется термин «grid» или «грид», который можно считать синонимом рассматриваемого понятия и под которым понимается вычислительная сеть.

Особенности метакомпьютера:

- большая ресурсная база;
- распределенная природа метакомпьютера вызывает появление латентности;
- возможность динамической смены конфигурации за счет системы поддержки, которая осуществляет поиск подходящих для вычислений ресурсов, проверяет их работоспособность и распределяет поступающие задачи;
- неоднородность метакомпьютера, вызванная различными системами команд и форматами представления данных в устройствах, входящих в состав метакомпьютера; различной пропускной способностью каналов связи; различной архитектурой входящих в состав метакомпьютера систем.
- необходимость взаимодействия с множеством организаций и как результат – трудность стандартизации и усложнение политики безопасности., что усложняет политику доступа, использование конкретных ресурсов, стандартизацию служб и сервисов метакомпьютера.

Процесс организации вычислений в системе, состоящей из метакомпьютеров, подключенных к сети Internet или к любой другой технологии называется метакомпьютингом.

Количество нерешенных проблем в настоящее время перевешивает эффект от использования метакомпьютинга.

Производительность параллельных компьютеров.

При оценке пиковой производительности параллельных компьютеров следует учитывать следующие показатели:

- число команд, выполняемых компьютером в единицу времени. Единицей измерения, как правило, является MIPS (Million Instructions Per Second). Этот показатель может меняться для различных групп команд.
- формат используемых данных, который влияет на производительность процессора.

- число вещественных операций, выполняемых компьютером в единицу времени. Данная единица измерения получила название - Flops (Floating point operations per second).

При оценке производительности следует учесть, что исходя из знания вычислительной сложности программы, можно получить только нижнею оценку времени выполнения этой программы.

Для адекватной оценки производительности необходимо комплексное тестирование всей программно-аппаратной среды [2].

В зависимости от целей тестирования выделяют и используют следующие уровни:

- базовый уровень программного обеспечения: тестирование эффективности работы операционной системы, компиляторов и систем программирования;
- базовый уровень аппаратуры: определение скорости выполнения элементарных операций, скорости обмена между различными уровнями иерархии памяти и объемы доступной памяти на каждом уровне;
- уровень операций ввода/вывода: анализ эффективности различных режимов чтения и записи данных при работе с внешними устройствами, определение скорости выполнения основных операций с файлами и целесообразности выполнения асинхронных операций ввода/вывода;
- базовый коммуникационный уровень: определение параметров среды взаимодействия параллельных процессов, эффективности выполнения основных коммуникационных процедур и примитивов синхронизации;
- коммуникационный уровень приложений: исследование эффективности отображения различных логических топологий процессов на коммуникационную среду, получение и анализ коммуникационных профилей характерных параллельных программ;
- уровень модельных приложений: определение характеристик компьютера при выполнении простых программ различной структуры;

- уровень приложений: комплексная проверка характеристик компьютера при выполнении реальных программ.

Задания

1. Может ли быть полезной на практике классификация компьютеров по их стоимости (весу, размеру, отказоустойчивости)? Если да, то для какого класса пользователей?
2. Может ли быть полезной на практике классификация компьютеров по их пиковой производительности? Если да, то для какого класса пользователей?
3. К какому классу по классификации Флинна можно отнести современные компьютеры с общей памятью, например, HP Superdome?
4. К каким классам по классификации Флинна можно отнести суперскалярные VLIW- процессоры?
5. Приведите примеры современных компьютеров класса SIMD.
6. Опишите с помощью метрики Фенга первые 10 компьютеров из списка Top500.Проделайте то же самое с помощью подходов Хендлера, Шнайдера, Скилликорна.
7. Как соотносятся друг с другом классификации Хендлера и Флинна?
8. Попробуйте найти взаимосвязь между изложенными в данном параграфе различными классификациями компьютеров.
9. *Какие параметры архитектуры параллельного компьютера нужно знать пользователю для создания эффективных программ? Предложите классификацию компьютеров, опираясь на выделенные параметры.
10. *В данном параграфе мы говорили о подходах к классификации компьютеров. Предложите классификацию технологий параллельного программирования, разработанных к настоящему времени.

Глава 6. Большие задачи и параллельные вычисления

6.1. Решение больших задач

Задачи, решения которых предполагает использование больших вычислительных мощностей. Усложнение математических моделей, используемых в процессе решения таких задач, приводит к усложнению методов реализации численных экспериментов, для поддержки которых используются параллельные вычисления.

Рассмотрим основные этапы численного эксперимента (рис. 22). Снижение эффективности любого из этапов приведет к снижению эффективности всего эксперимента.

Рис. 22. Этапы численного эксперимента

6.2. Граф алгоритма и параллельные вычисления

В процессе разработки программы присутствует частичный порядок выполнения операций, который предполагает установку очередности выполнения операции во времени или независимое их выполнение. Для программы частичный порядок сохраняется в независимости от совокупного времени выполнения множества операций, которое может меняться. То есть предполагается возможность выбора некоторой реализации в пределах неизменного частичного порядка.

Пусть некоторая программа описывает алгоритм с фиксированными входными данными. *П* построении ориентированного графа в качестве вершин выберем множество точек арифметического пространства, которое взаимно однозначно отображается на множество всех операций алгоритма. Для любой пары вершин u, v будем считать, что операция, которая соответствует вершине u, отвечает за доставку соответствующей вершине v аргумента операции. В случае, если операции зависимы друг от друга, проведем дугу из вершины u в вершину v. Если соответствующие операции могут выполняться независимо друг от друга, дуга не проводится. Случаи, в которых аргументы операции являются начальными данными или результат операции нигде не используется, вершины графа не будут иметь соответственно входящие и выходящие дуги. Такие вершины будем называть соответственно входными или выходными вершинами графа. Граф алгоритма почти всегда зависит от входных данных, поэтому граф - граф параметризованный. От значений параметров зависит число вершин, и совокупность дуг. Граф определяется, как граф информационной зависимости и соответствует реализации алгоритма при фиксированных входных данных. Такой граф еще принято называть **графом алгоритм**.

В случае наличия в программе условных операторов, соответствующий алгоритм называется **детерминированным**, иначе - **недетерминированным**. Каждой вершине графа детерминированного алгоритма однозначно

соответствует операция реализующей этот алгоритм программы. Для недетерминированного алгоритма такое соответствие построить нельзя.

Введенный в рассмотрение граф алгоритма есть ориентированный ациклический мультиграф, что для любых программ означает выполнение только явных вычислений и невозможность определения величины через саму себя. В общем случае граф алгоритма есть мультиграф, т. е. две вершины могут быть связаны несколькими дугами. Это будет тогда, когда в качестве разных аргументов одной и той же операции используется одна и та же величина. Обозначим граф, как ***C= (V, E),*** где ***V*** - множество вершин, ***E*** - множество дуг графа ***C***.

На основе графа алгоритма можно построит по определенным правилам строгую параллельную форму графа. Совокупность вершин с одинаковыми индексами определяется, как ярус параллельной формы. Ширина яруса - число вершин в группе. Высота параллельной формы определяется числом ярусов параллельной формы, а ширину составляет максимальная ширина ярусов. Максимальной называется параллельная форма минимальной высоты.

Между различными реализациями одного и того же алгоритма на различных компьютерах и различными параллельными формами графа алгоритма существует взаимное соответствие. По результатам исследования графа алгоритма и его параллельных форм, можно оценить запас параллелизма в алгоритме и выбрать его лучшую реализацию на конкретном компьютере параллельной архитектуры.

6.3. Концепция неограниченного параллелизма

Необходимость реализации решения задач на параллельных вычислительных системах способствовала развитию математической концепции построения параллельных алгоритмов – концепции неограниченного параллелизма. Концепция основана на предположении о том, что реализация алгоритма на параллельной вычислительной системе, не

накладывает на алгоритм никаких ограничений. Предполагается использование неограниченного числа процессоров, их универсальности и работа в синхронном режиме, наличие общей памяти, передача информации осуществляются мгновенно и без конфликтов. В рамках концепции предполагается реализация алгоритмов минимальной высоты.

Пример. Вычисление произведения n чисел $\boldsymbol{a_1, a_2, ..., a_n}$, в котором отчетливо видна идея, играющая большую роль в построении алгоритмов малой высоты [1].

Пусть $\boldsymbol{n = 8}$. Обычная схема, реализующая процесс последовательного умножения, выглядит следующим образом:

Данные a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

Ярус 1 a_1 a_2

Ярус 2 $(a_1\ a_2)$ a_3

Ярус 3 $(a_1\ a_2\ a_3)$ a_4

Ярус 4 $(a_1\ a_2\ a_3\ a_4)$ a_5

Ярус 5 $(a_1\ a_2\ a_3\ a_4\ a_5)$ a_6

Ярус 6 $(a_1\ a_2\ a_3\ a_4\ a_5\ a_6)$ a_7

Ярус 7 $(a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7)$ a_8

Высота параллельной формы равна 7, ширина равна 1. При данной схеме вычисления и наличии более одного процессора, на каждом шаге вычисления все процессоры кроме одного будут простаивать. Рассмотрим параллельную форму другого алгоритма решения данной задачи:

Данные a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

Ярус 1 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8

Ярус 2 $(a_1\ a_2)$ $(a_3\ a_4)$ $(a_5\ a_6)$ $(a_7\ a_8)$

Ярус 3 $(a_1\ a_2\ a_3\ a_4)$ $(a_5\ a_6\ a_7\ a_8)$

Высота параллельной формы равна 3, ширина равна 4. Повышение загруженности процессоров работой привело к снижению высоты. Процесс построения чисел каждого яруса по описанной схеме называется процессом

сдваивания. На каждом ярусе осуществляется максимально возможное число произведений непересекающихся пар чисел, взятых на предыдущем ярусе. В общем случае высота параллельной формы равна $\boldsymbol{log_2 n}$. Эта параллельная форма реализуется на $\boldsymbol{n/2}$ процессорах, и загруженность процессоров уменьшается от яруса к ярусу. На рис. 23 для $\boldsymbol{n = 8}$ представлен граф последовательного применения операции, внизу – граф для принципа сдваивания. Начальные вершины символизируют ввод данных.

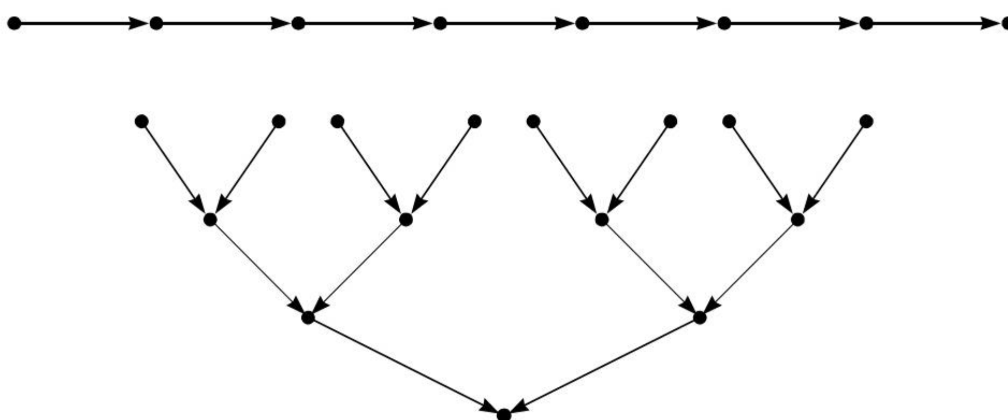


Рис. 23. Последовательный граф и граф сдваивания

Таким образом, если какая-нибудь задача определяется n входными данными, то в общем случае не существует алгоритма ее решения с высотой менее $\boldsymbol{log_2 n}$. Если получен алгоритм высоты порядка $\boldsymbol{log^a\ n}$, где $\boldsymbol{a > 0}$, то такой алгоритм считается эффективным с точки зрения времени его реализации на параллельной вычислительной системе при исключении иных аспектов реализации.

Идея абстракции относительно реальной работы параллельных систем, допускаемая концепцией, была использована в математических исследованиях и обеспечила появление ряда изобретений в численных методах.

Концепция получила развитие в 1950-1960 годы. Алгоритмы, построенные на основе концепции, оказались несостоятельными, в связи с действительной сложностью информационных связей между операциями,

численной неустойчивостью, наличием большого количества конфликтов в памяти. Однако, несмотря на недостатки концепции, предельная степень абстракции от реальной ситуации в вычислительных системах позволила проводить математические исследования, что привело к появлению отдельных изобретений в области численных методов.

6.4. Внутренний параллелизм

Если для алгоритма, предполагающего реализацию в непараллельной вычислительной системе, удастся построить граф, найти для этого графа параллельную форму с достаточной шириной ярусов, то данный алгоритм, теоретически можно реализовать на параллельном компьютере. При этом вычислительные свойства алгоритма сохранятся. Численная устойчивость алгоритма сохраняется в любой форме. Такой параллелизм в алгоритмах называется внутренним.

Пример. Вычисление произведения $\boldsymbol{A = BC}$ двух квадратных матриц $\boldsymbol{B}$, C порядка n. Будем считать элементы матриц $\boldsymbol{A, B, C}$ числами и обозначим их a_{ij}, b_{ik}, c_{kj}. Согласно определению операции умножения матриц имеем

$$a_{ij} = b_{ik}\, c_{kj},\ \text{где } i, j = 1, 2,.., n.$$

На рис. 24 представлен граф для $n = 2$.

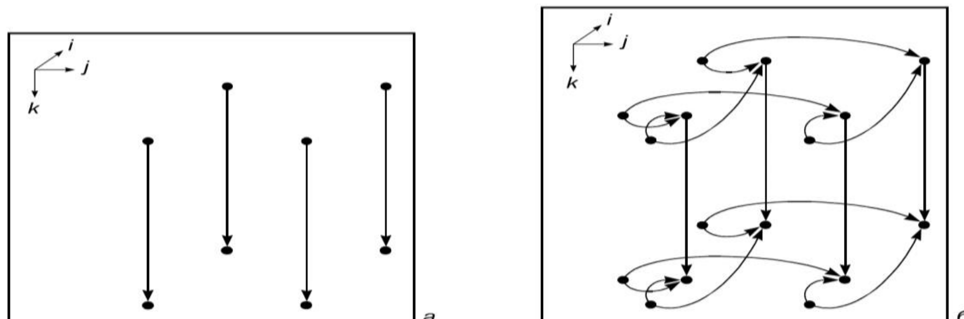


Рис. 24. Графы перемножения матриц: а) граф алгоритма для n = 2,

б) граф с рассылками элементов b_{ik} и c_{kj} для n = 2

Граф алгоритма распадается на $\boldsymbol{n^2}$ не связанных между собой подграфов. Каждый подграф содержит n вершин и представляет один путь, расположенный параллельно оси $\boldsymbol{k}$ граф алгоритма представлен на рис 24 а. В полном графе имеется множественная рассылка данных. Элемент b_{ik} рассылается по всем вершинам, имеющим те же самые значения координат $\boldsymbol{i}$, $\boldsymbol{k}$. Элемент c_{kj} рассылается по всем вершинам, имеющим те же самые координаты $\boldsymbol{k, j}$. Для $\boldsymbol{n = 2}$ эти рассылки показаны на рис. 24 б. Все вершины соответствуют операциям вида $\boldsymbol{a + bc}$.

Глава 7. Параллельные алгоритмы

Параллельный алгоритм – алгоритм, предназначенный для реализации на параллельной вычислительной системе. Рассмотрим задачу вычисления чисел Фибоначчи, решение которой реализовано в виде последовательного и параллельного алгоритмов.

Математическая модель

Пример [2]. Пусть последовательность Фибоначчи задана рекурентным соотношением $\boldsymbol{F(n) = F(n-1) + F(n-2)}$, ***при*** $\boldsymbol{F(0) = 1}$ ***и*** $\boldsymbol{F(1) = 1}$. Определить значение элемента n (для n>1).

Последовательный рекурсивный алгоритм

```
F(n)
{
 if(n<2) return 1;
 return F(n-2)+F(n-1);
}
```

Обозначим как $\boldsymbol{G(n)}$ число операций для вычисления числа $\boldsymbol{F(n)}$.

$$G(n) = G(n-1) + G(n-2) + 1,$$

где

G(n – 1) – количество операций, необходимое для вычисления чисел *F(n – 1),* G(n – 2) - количество операций, необходимое для вычисления чисел F(n – 2), 1 – одна операция для нахождения суммы.

Определяем: ***G(0) = 0, G(1) = 0.*** $\boldsymbol{G_{A1}(n) = F(n) - 1}$

И получаем оценку:

$$n \to \infty,\ F_n \sim \frac{\varphi^n}{\sqrt{5}},\ \text{где}\ \varphi = \frac{1+\sqrt{5}}{2}\ —$$

$$G_{A1}(n) \sim 0{,}45 \cdot 10^{n/4{,}78}.$$

Параллельный рекурсивный алгоритм

Предположим, что каждая из функций ***F(n – 2)*** и *F(n – 1)* обрабатывается на отдельном процессоре [2].

```
F(n)
{ if(n<2) return 1;
Процессор 1 Процессор 2
a= F(n-2) b= F(n-1)
 return a+b;
}
```

Для выполнения параллельного алгоритма около $\boldsymbol{10^{n/4{,}75}}$ процессоров. Оценка числа операций этого алгоритма составляет ***(n – 2).*** Глубина самой длинной ветви рекурсии составит *(n-2).* Выполнение ***(n-2)*** операций осуществляется последовательно.

Параллельный алгоритм должен выполняться быстрее, чем самый оптимальный из последовательных алгоритмов, реализующих решение задачи.

7.1. Классификация алгоритмов по типу параллелизма

1. Алгоритмы, использующие параллелизм данных (Data Parallelism). Этот тип параллелизма характерен для численных алгоритмов обработки, имеющих дело с большими массивами, представляемыми, например, в виде векторов и матриц. Простейшим примером такой задачи является, например, процедура перемножения двух матриц.

2. Алгоритмы с распределением данных (Data Partitioning). Это разновидность параллелизма данных, при котором пространство данных может быть разделено на непересекающиеся области, с каждой из которых связаны независимые процессы, оперирующие каждый со своими данными. Требуется лишь редкий обмен между этими процессами.

3. Релаксационные алгоритмы (Relaxed Algorithm). Алгоритм может быть представлен в виде независимых процессов без синхронизации связи между ними, но процессоры должны иметь доступ к общим данным.

4. Алгоритмы с синхронизацией итераций (Synchronous Iteration). Многие из стандартных численных итерационных параллельных алгоритмов процессоры завершили предыдущую итерацию.

Самовоспроизводящиеся задачи (Replicated Workers). Для задач этого класса создается и поддерживается центральный пул (хранилище) похожих вычислительных задач. Параллельно реализуемые процессы осуществляют выбор задач из пула, выполнение требуемых вычислений и добавление новых задач к пулу. Вычисления заканчиваются, когда пул пуст. Эта технология характерна для исследований графа или дерева.

Конвейерные вычисления (Pipelined Computation). Этот тип вычислений характерен для процессов, которые могут быть представлены в виде некоторой регулярной структуры, например, в виде кольца или двумерной сети. Каждый процесс, находящийся в узле этой структуры, реализует определенную фазу вычислений.

7.2. Общая схема разработки параллельных алгоритмов

Разработка параллельных алгоритмов состоит из следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.

2. Определение для сформированного набора подзадач информационных взаимодействий.

3. Масштабирование подзадач, определение количества процессоров.

4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

Этапы 1-4 могут повторяться, в случае необходимости, например для улучшения эффективности алгоритма. Если желаемые показатели не достигаются, то следует изменить математическую модель задачи.

Приведенная схема является общей, в этой связи в каждом конкретном случае последовательность этапов может меняться. Например, если заранее не известно точное число процессоров, но известны границы решающего поля, разработку алгоритма можно начать с масштабирования базового набора задач и только потом перейти к декомпозиции и выявление связей по информации. Схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений приведена на рис. 25.

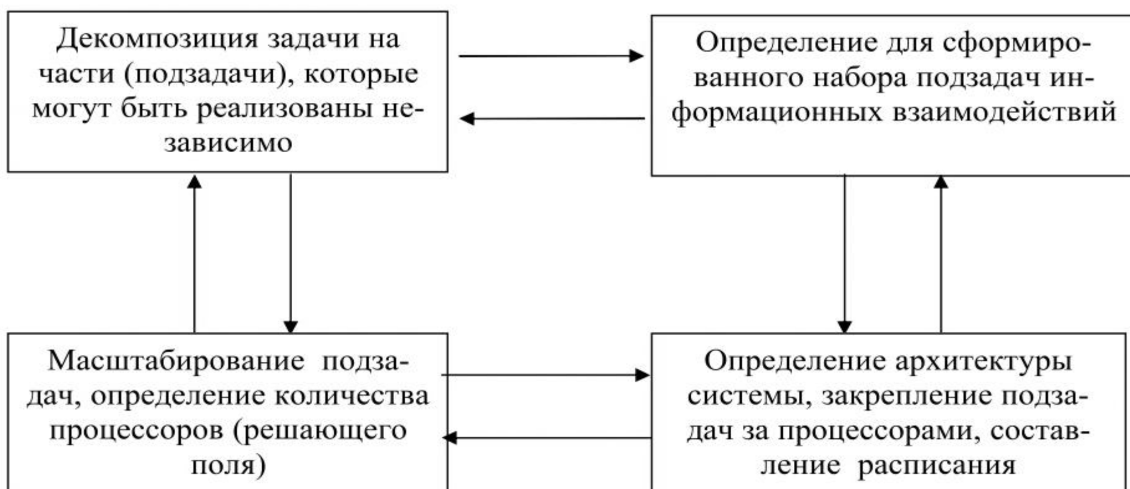


Рис. 25. Схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений

На этапе декомпозиции осуществляется выделение базовых подзадач.

В процессе декомпозиции предъявляются минимальные требования обеспечить:

- примерно равный объем вычислений в выделяемых подзадачах;
- минимальный информационный обмен данными между процессорами.

На этапе анализа информационных зависимостей между подзадачами различают следующие типы этих зависимостей:

- локальные (на соседних процессорах) и глобальные (в которых принимают участие все процессоры) схемы передачи данных;
- структурные (соответствующие типовым топологиям коммуникаций) и произвольные способы взаимодействия;
- статические (задаваемые на этапе проектирования) или динамические (определяемые в ходе выполняемых вычислений);
- синхронные (следующая операция выполняется после выполнения предыдущей операции всеми процессорами) и асинхронные способы взаимодействия (процессы могут не дожидаться полного завершения действий по передаче данных) подзадачи обладают высокой степенью информационной взаимозависимости.

Этап масштабирования параллельного алгоритма выполняется в случае, если количество подзадач (областей данных) отличается от числа процессоров. Тогда осуществляется переход на этап декомпозиции. При этом, число подзадач уменьшают за счет укрупнения области исходных данных. В первую очередь следует объединить области, для которых подзадачи обладают высокой степенью информационной взаимозависимости.

На этапе закрепления задач за процессорами следует учитывать информационные взаимодействия между областями данных этих задач. Такие задачи целесообразно размещать на процессорах, связанных прямыми линиями передачи данных.

Приведенная схема может использоваться для построения параллельного алгоритма, который характеризуется параллелизмом данных и параллелизмом задач.

Если имеет место параллелизм данных, то задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Должно соблюдаться требование равномерная загрузка процессоров. При этом следует учитывать возможность различной производительности процессоров.

Эффективность параллельной программы зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных.

Вычислительная задача разбивается на несколько самостоятельных подзадач в том случае если в ней отсутствует параллелизм по данным. Каждый процессор занимается решением отдельной подзадачи. В данном случае имеет место параллелизм задач. Количество задач влияет на количество процессоров. При обеспечении равномерной загрузки процессоров и минимизации обмена данными между ними можно ожидать значительного ускорения. Эффективность кода предполагает анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсоемких частей.

7.3. Простые параллельные операции

Рассмотрим две элементарные операции:

- распределение входных данных по процессорам
- поиск минимального или максимального элемента списка.

Пусть P_i **-** ***i-й*** процессор; ***p*** **-** число процессоров*,* ***N*** - число элементов во входном списке, M_j - ***j-ая*** ячейка памяти через. Параллельно операции записываются в скобках ***Parallel Start*** и ***Parallel End.*** Если сначала параллельно выполняется один блок операций, а затем другой, то они заключаются в отдельные пары скобок.

Распределение данных в модели **CREW PRAM**

В модели CREW одну и ту же ячейку памяти могут читать одновременно несколько процессоров. Таким образом, запись значения в процессоры происходит достаточно быстро. Значение записывается в память, затем все процессоры читают его. Такая модель называется – конкурентное чтение.

```
P[1] записывает значение в M[1]
Parallel Start
for k = 2 to p do
P[k] читает значение из M[1]
end for
Parallel End
```

Распределение данных в модели EREW PRAM

Значение, записанное процессором P_i, может прочитать только один процессор. Если использовать цикл чтение / обработка / запись на втором процессоре для записи значения в другую ячейку памяти, то на следующем шаге уже два процессора смогут прочитать это значение. Далее записав значение еще в две ячейки, можно использовать четыре процессора. Это - модель с исключительным чтением.

```
P[1] записывает значение в ячейку M[1]
procLoc = 1
for j = 1 to log p do
Parallel Start
for k = procLoc + 1 to 2 * procLoc do
P[k] читает M[k - procLoc]
P[k] записывает в M[k]
end for k
Parallel End
procLoc = procLoc * 2
end for j
```

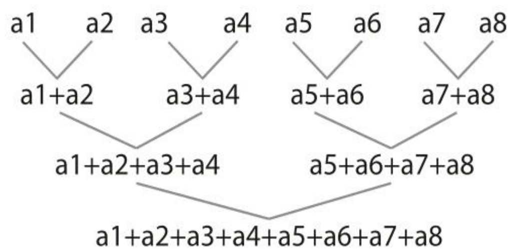
Глава 8. Базовые методы построения параллельных алгоритмов: метод сдваивания, геометрический параллелизм

8.1. Базовые методы построения параллельных алгоритмов

Параллельные алгоритмы не всегда показывают более высокую, чем последовательные алгоритмы, степень эффективности. Несмотря на это имеется достаточно большой набор задач, решение которых возможно улучшить за счет применения параллельных вычислений.

Правильность этих методов подтверждена многолетней практикой и дает возможность обеспечивать их применимость для решения самых разных задач. Рассмотрим последовательно следующие методы:

- метод сдваивания;
- метод геометрического параллелизма;



Номер шага	Процессор 1	Процессор 2	Процессор 3	Процессор 4
1	s12 = a1+a2	s34 = a3+a4	s56 = a5+a6	s78 = a7+a8
2	s1234 = s12+s34		s5678 = = s56+s78	
3	s1234 = s1234+s5678			

Рис. 26. Нахождения суммы 8-ми элементов массива чисел на четырех процессорах

Пример. Определение суммы элементов массива а, содержащего n элементов [2]. На рис. 26 приведен пример нахождения суммы 8-ми элементов массива, используя 4 процессора.

Последовательный алгоритм:

```
s=a[1];
for(i=2;i<=n;i++)
s=s+a[i];
```

Будем учитывать только операции сложения чисел, пренебрегая операциями, связанными с организацией цикла суммирования, чтением элементов массива из памяти и им подобными. Данные операции не важны с точки зрения оценки свойств рассматриваемого метода.

Пусть $\boldsymbol{\tau_A}$ - время выполнения операции сложения двух чисел; $\boldsymbol{T_1}$ - время работы последовательного алгоритма, тогда:

$$T_1 = (n-1)\tau_A. \tag{1}$$

Степень параллелизма алгоритма равна 1.

8.2. Построение параллельного алгоритма методом сдваивания

Предположим, что размер массива составляет n = 2^q. В этом случае количество этапов выполнения алгоритма сдваивания составит q = $\log_2 n$:

1-й этап - n/2 действий,

2-й этап - n/4,

…

q-й – 1.

Пусть k – некоторый этап выполнения алгоритма. Тогда степень параллелизма на этом этапе составит $n/2^k$. Ускорение S_p параллельного алгоритма по сравнению с наилучшим последовательным, достигаемое в системе из p процессоров, при p = n/2:

$$T_p\Big|_{p=n/2} = \tau_A \cdot \log_2 n, S_p = \frac{n-1}{\log_2 n}, E_p = \frac{n-1}{\log_2 n}\frac{2}{n} \approx \frac{2}{\log_2 n}. \tag{2}$$

где α – доля операций алгоритма, выполнение которых невозможно одновременно с другими операциями (операции со степенью параллелизма 1); E_p – сверхлинейное ускорение, которое превышает вычислительные возможности системы (число процессов для решения вычислительной

задачи) S_p > p. Причины появления сверхлинейного ускорения – особенности, используемых вычислительных систем, неудачный выбор последовательного алгоритма

Метод обеспечивает высокое ускорение и низкую эффективность. При ***n*** = 1024 ускорение равно 102, эффективность – 20%. Причина низкой эффективности в простаивании большей части процессоров на всех стадиях. Так как вычисления выполняются на всех стадиях только на первом процессоре. Алгоритм предполагает использование ***n/2*** процессоров для обработки n элементов. Разобьем массив , состоящий из n на p одинаковых частей. При условии уменьшения количества процессоров, вычисления в каждой части будут выполняться двумя этапами на отдельном процессоре:

1 этап: находится сумма ***n/p*** элементов соответствующей процессору части элементов массива. Степень параллелизма равна p, так как взаимодействие между процессорами не требуется.

2 этап: с помощью метода сдваивания определяется окончательная сумма.

Время $\tilde{T}$, необходимое для вычисления суммы при использовании p процессоров – формула (3). Пусть ***p << n***. Эффективность, обеспечиваемая применением рассматриваемого метода, выражена формулой (4); ускорение – формулой (5).

$$\tilde{T}_p = \tau_A \cdot \frac{n}{p} + \tau_A \cdot \log_2 p. \quad (3)$$

$$\tilde{S}_p = \frac{n\tau_A}{\tau_A \cdot \frac{n}{p} + \tau_A \cdot \log_2 p} = p\frac{1}{1 + \frac{p}{n}\log_2 p}. \quad (4)$$

$$\tilde{E}_p = \frac{1}{1 + \frac{p}{n}\log_2 p}. \quad (5)$$

При ***p=n/log₂n и n/p*** - целое число, имеем:

$$\tilde{S}_p = \frac{p}{2}\tilde{E}_p = 50\%. \quad (6)$$

При оценке общего случая необходимо учитывать операции, которые обеспечивают взаимодействие процессоров. Массив, в данном случае целесообразно разделить на части. Размеры частей могут отличаться друг от друга не более чем на единицу. При большом ***n/p*** можно пренебречь дисбалансом вычислительной нагрузки процессоров.

Пусть $\boldsymbol{\tau_s}$ - время для передачи числа от одного процессора другому.

Для систем с распределенной памятью следует учесть, что на это время оказывают влияние две характеристики канала передачи данных: пропускная способность и латентность.

Для систем с общей памятью $\boldsymbol{\tau_s}$ зависит от времени обращения к синхронизирующему примитиву. Таким образом, время работы алгоритма составит:

$$\bar{T}_p = \tau_{\mathrm{A}} \cdot \frac{n}{p} + \left(\tau_{\mathrm{A}} + \tau_s\right) \cdot \log_2 n, \qquad (7)$$

$$\bar{S}_p = p \frac{1}{1 + \left(1 + \frac{\tau_s}{\tau_c}\right) \frac{p}{n} \log_2 n}. \qquad (8)$$

где τ_c - время, затрачиваемое на сложение двух чисел.

Пусть $\boldsymbol{\tau_s < \tau_c}$ и р по своей величине близко к ***n.*** Тогда показатели ускорения и эффективности будут снижаться за счет операций, выполняемых для обеспечения взаимодействия процессоров.

Рассматриваемый метод может применяться для ряда редукционных операций, таких, как: определение суммы, произведение множества чисел, поиск минимума (максимума) для набора данных, для ускорения рассылки равных данных множеству процессоров.

Пример. Пусть в системе с распределенной памятью требуется передать значение некоторой, известной на первом процессоре величины на все остальные процессоры [2].

Алгоритм последовательной передачи

```
// p -- общее число процессов.
// rank -- номер выполняемого процесса,
// каждому процессу присвоен уникальный номер
// из диапазона [Θ, p-1]
if(rank = = Θ)
{
 for(next = 1; next < p; next++)
Send(next);
}
else
 Recv(Θ);
```

Затраченное время:

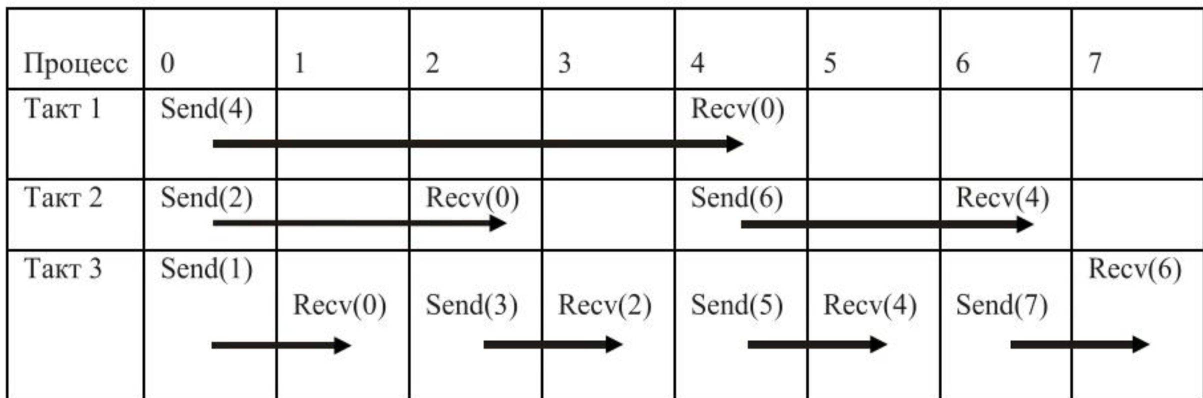
$$T_p = \overset{0}{\tau}_s (p-1). \quad (9)$$

Использование следующего алгоритма, основанного на методе сдваивания, позволяет получить следующее время рассылки:

$$T_p = \tau_s \log_2 p. \quad (10)$$

```
SendOneToAll ( )
{
 for(p1=1; p1 < p; p1*=2);
 if(rank > Θ)
  {
// определим номер младшего в rank разряда,
// отличного от Θ
```

```
for( delta =1; ( delta & rank ) = = Θ ; delta *= 2 );
 from = rank - delta; // определим номер передающего процессора
 Recv(from);
 }
 else
 delta = p1;
 for(delta /= 2; delta > 0 ; delta /= 2)
 {
 next = rank + delta;
 if (next < p)
Send(next);
 }
}
```



Процесс	0	1	2	3	4	5	6	7
Такт 1	Send(4)				Recv(0)			
Такт 2	Send(2)		Recv(0)		Send(6)		Recv(4)	
Такт 3	Send(1)	Recv(0)	Send(3)	Recv(2)	Send(5)	Recv(4)	Send(7)	Recv(6)

Рис. 27. Схема рассылки данных

На рисунке 27 представлена схема рассылки данных.

8.3. Метод геометрического параллелизма

Метод геометрического параллелизма подходит для обработки изображений, вычислений, связанных с математической физикой и т.д.

Метод обеспечивает выполнение множества однотипных действий над однородными взаимосвязанными данными и эффективен для алгоритмов, обладающих свойством локальности:

При распределении данных по процессорам равными частями следует обеспечить расположение вспомогательных данных, используемых конкретным процессором на минимальном количестве других процессоров и объем этих данных должен быть минимальным.

Рассмотрим применение метода на примере.

Пример («Стена Фокса») [2].

Группе каменщиков необходимо согласовать построить стену.

Каждый каменщик отвечает за построение участка стены. Все участки - одинаковы. Каменщики начинают работу одновременно с укладывания нижнего слоя (рис. 28). Переход к укладке следующего слоя на конкретном участке осуществляется при условии окончания работы по укладыванию этого слоя на соседних участках. В противном случае возникают простои, связанные с синхронизацией работ на соседних участках.

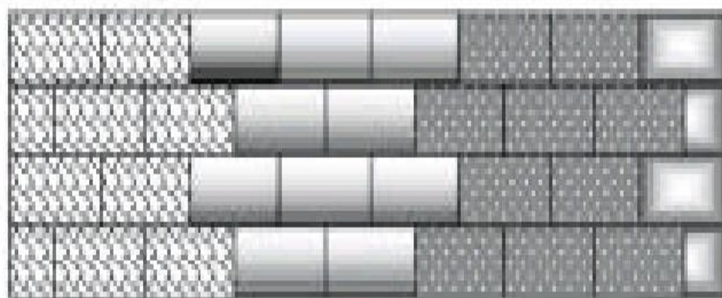


Рис. 28. Геометрическое решение

Вычисления реализуются в системе с распределенной памятью. Каждое вычисление выполняется отдельным процессором.

Пусть ***n*** – число кирпичей в ряду, ***k*** – высота стены, τ_c – время, необходимое для укладки одного кирпича, T_p – время решения задачи с помощью p параллельно работающих процессов. В случае одного процесса, потребуется последовательно уложить ***n*k*** кирпичей:

$$T_1(kn) = \tau_c kn. \qquad (11)$$

В случае нескольких процессов, каждому из которых выделяется вертикальный участок стены, длина которого составляет ***n/p***, объем полезной

работы составит ***τ_c * n/p.*** Укладку кирпича на левой границе 0стены предваряет вычисление в процессе с номером rank функции, которая обеспечивает получение сообщения, поступающего от процесса с номером ***(rank – 1).*** Процесс с номером ***(rank – 1)*** отвечает за выполнение функции передачи необходимых данных. Двусторонний обмен данными осуществляется между процессом rank и процессом ***rank – 1*** в общем случае. Таким образом каждые два соседних процесса обмениваются сообщениями, число которых равно двум. Все процессы, не являющиеся первыми и последними в цепочке, обмениваются четырьмя сообщениями с соседями. Оптимальное время работы составит:

$$T_p = \tau_c \frac{kn}{p} + 4k\,\tau_s. \qquad (12)$$

Алгоритм, обеспечивающий это время:

```
for(шаг = Θ; шаг <k ; шаг++)
{
for(кирпич = rank*n/p; кирпич <(rank+1) * n/p; кирпич ++)
 Уложить(кирпич)
if(rank %2 )
{if(rank > Θ) Send (rank-1, кирпич уложен )
if(rank > Θ) Recv (rank-1, место готово? )
if(rank < p-1) Recv (rank+1, место готово? )
if(rank < p-1) Send (rank+1, кирпич уложен )
}
else
{
if(rank < p-1) Recv(rank+1, место готово? )
if(rank < p-1) Send(rank+1, кирпич уложен )
if(rank > Θ) Send(rank-1, кирпич уложен )
```

```
if(rank > Θ) Recv(rank-1, место готово? )
}
}
```

Такт	0	1	2	3	4	5
1	=<	⇐	=<	⇐	=<	⇐
2	⇒	>=	⇒	>=	⇒	>=
3		=<	⇐	=<	⇐	
4		⇒	>=	⇒	>=	

Рис. 28 а. Распределения данных аналогично методу геометрического параллелизма

Порядок операций обмена данными в алгоритме представлен на рис. 28 а.

Эффективность и ускорение алгоритма

$$S_p = p\frac{1}{1+4\dfrac{p}{n}\dfrac{\tau_s}{\tau_c}}, \quad E_p(kn) = \frac{1}{1+4\dfrac{p}{n}\dfrac{\tau_s}{\tau_c}}. \tag{13}$$

Приведенные выше оценки имеют смысл при выполнении следующих условий:

- количество кирпичей для процессов – одинаково;
- от положения кирпича в стене не зависит количество операций, выполняемых при его укладке;
- производительности процессов;
- латентности и пропускные способности каналов передачи данных равны.

На практике эти условия выполняются только приближенно в связи с наличием процессов, которые при укладке ряда кирпичей выполняют шаги за разное время. Примером может служить работа каменщиков на крайних участках стены. Если известны трудоемкость укладки каждого кирпича и производительность каждого процессора, то при статической балансировке можно улучшить показатели следующим образом: предположим, что заранее

известен процесс, выполняющий вычисления медленнее других и время на обработку каждого кирпича одинаково. В этом случае, для этого процесса, можно уменьшить объем количество кирпичей. Таким образом, удастся избежать простоев, и каждый процесс будет затрачивать равное время на вычисления. Иначе нагрузка процессоров должна балансироваться динамически.

Глава 9. Базовые методы построения параллельных алгоритмов конвейерный параллелизм; диффузная балансировка загрузки

9.1. Конвейерный параллелизм

При конвейерном выполнении работ ***n-й*** каменщик укладывает ***n-й*** слой кирпичей. На рисунке 29 слои одного цвета укладываются одним каменщиком. После того, как первый полностью уложит первый слой, он продолжает укладку слоя ***p + 1.*** Таким образом, первым начать работу может только первый каменщик, а остальные - ожидать укладки нижних слоев. Время ожидания будет самым большим у последнего каменщика. В конце работы первый каменщик будет ждать, пока работу закончат все остальные.

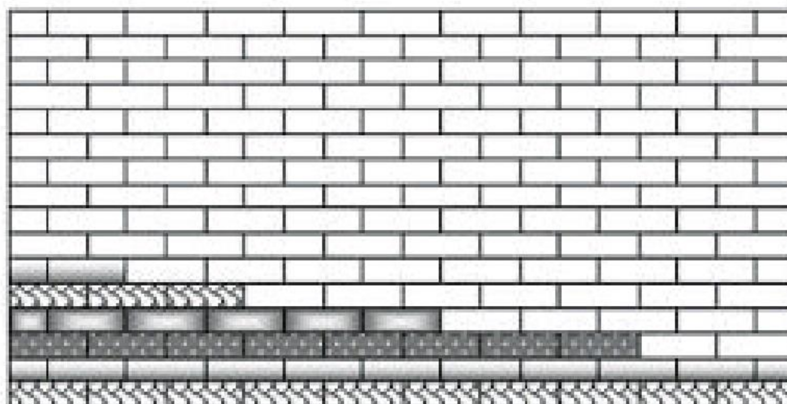


Рис. 29. Пример укладки слоев каменщиками

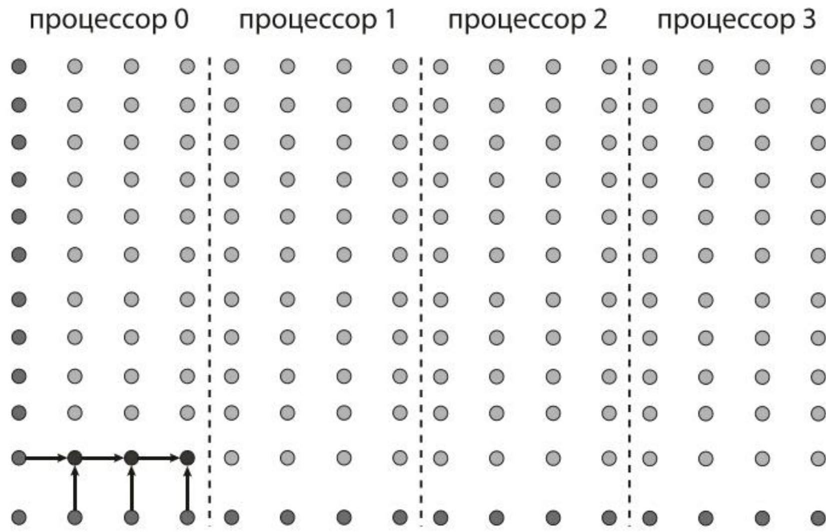
При высокой и длинной стене большинство каменщиков будет равномерно загружена (при условии неизменности объема работ, одинаковой скорости работы каменщиков, одинаковом времени укладки одного кирпича) . Предложенная стратегия имеет низкую эффективность в связи с тем, что при укладке нового кирпича в верхнем слое появляется потребность в сообщении о наличии соответствующего кирпича в нижнем слое. На рис. 30 представлен случай распределения данных аналогично методу геометрического параллелизма.

До тех пор, пока нулевой процессор не закончит обработку n/p точек с номерами ***[0, n/p – 1]*** другие процессоры будут находиться в состоянии ожидания. Данная ситуация проиллюстрирована на рис. (рис. 30 а).

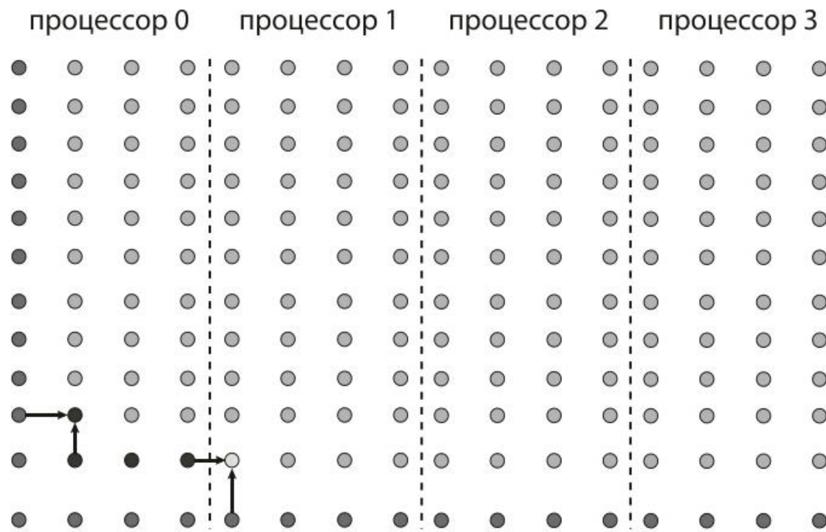
Процессор один получит информацию о точке с номером ***(n/p – 1)*** только после окончания обработки ***n/p*** точек будут. Таким образом, в работе будут участвовать два процессора:

- процессор 0 – вычисление значений третьего слоя в узлах *[0, n/p – 1];*

- процессор 1 – вычисление значений второго слоя в узлах ***[n/p, 2n/p – 1].*** Данная ситуация проиллюстрирована на (рис. 30 б).



а



б)

Рис. 30. Эффективное конвейерное решение

Передача и прием данных выполняются каждым процессором один раз (рис. 31 а). Основные характеристики оцениваются следующим образом [2]:

$$T_p = \tau_A \cdot \frac{kn}{p} + 2k\,\tau_s,\ S_p = p\frac{1}{1+2\dfrac{p}{n}\dfrac{\tau_s}{\tau_c}},\ E_p = \frac{1}{1+2\dfrac{p}{n}\dfrac{\tau_s}{\tau_c}}. \tag{14}$$

Оценки не учитывают потерь, обусловленных начальным простоем практически всех процессоров (при загрузке конвейера) и финальными простоями, из-за того, что процессоры с меньшими номерами закончат

работу раньше остальных. В отличие от метода геометрического параллелизма, в котором при равномерном распределении работы все процессоры в каждый из моментов времени вычисляют значения для одного и того же слоя (укладывают один и тот же слой кирпичей), в методе конвейерного параллелизма это принципиально не так. Слои обрабатываемые, разными процессорами, имеют разные номера. На каждом процессоре хранится информация только о двух соседних слоях: слое, рассчитываемом на процессоре в данный момент, и о предыдущем слое (рис. 31 б).

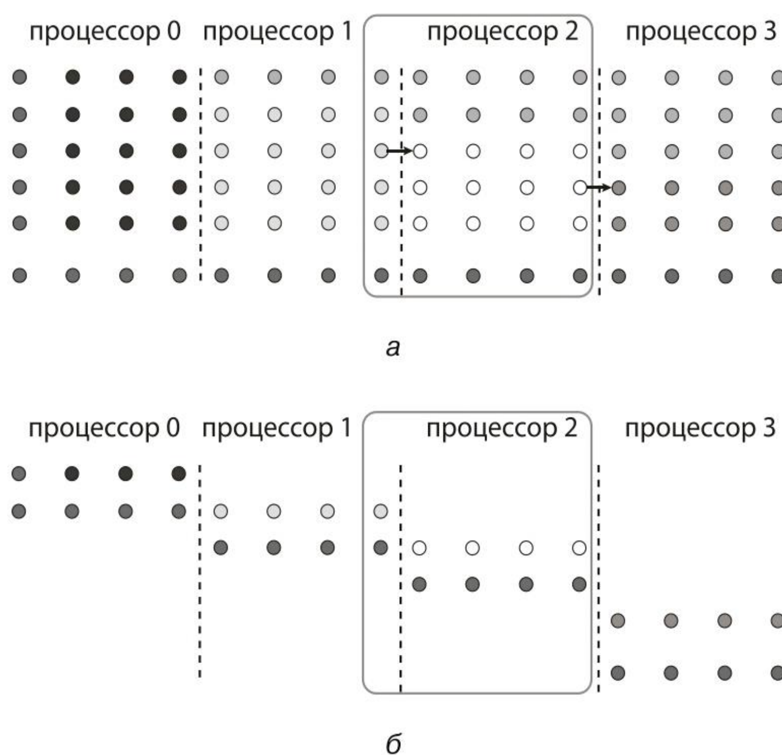


Рис. 31. Эффективное конвейерное решение

Учитывая влияния начальных и конечных простоев на ускорение и эффективность, получим:

$$S_p^{all} = p\frac{1}{\left(1+\frac{p}{k}\right)\left(1+2\frac{p}{n}\frac{\tau_s}{\tau_c}\right)}. \quad (15)$$

$$E_p^{all} = \frac{1}{\left(1+\frac{p}{k}\right)\left(1+2\frac{p}{n}\frac{\tau_s}{\tau_c}\right)}. \quad (16)$$

$$T^{all} = (p + k - 2)\left(\frac{n}{p}\tau_c + 2\tau_s\right). \tag{17}$$

где T^{all} – общее время работ, равное сумме начального, промежуточного и конечного времени работы.

Рассмотренный метод обладает высокой эффективностью только при соблюдении условия ***p << k.*** Стена должна быть достаточно высокой, в противном случае накладные расходы на запуск и остановку конвейера обработки могут почти вдвое (при ***p = k)*** снизить ускорение и эффективность.

Отличия от метода геометрического параллелизма:

величина максимальной степени внутреннего параллелизма при геометрическом параллелизме - ***n***, а при конвейерном — ***min (n, k).***

Максимально достижимое ускорение, при условии, что затраты на передачу данных – нулевые, в случае геометрического параллелизма равно ***n***, а в случае конвейерного ***– min(n, k).***

9.2. Диффузная балансировка загрузки

В отличие от предыдущих методов, использующих метод статической балансировки загрузки процессоров, метод диффузной балансировки загрузки является методом динамической балансировки загрузки процессоров.

В данном случае перераспределение вычислительной нагрузки выполняется между логически соседними процессорами. Данный метод применяется при решении задач с квазистационарным (т. е. медленно меняющимся) распределением вычислительной нагрузки по узлам сетки и обеспечивает сохранение свойства «локальности» алгоритма, не требуя непосредственного взаимодействия каждого из процессоров со всеми остальными.

Пусть известно распределение узлов вычислительной сетки по процессорам и известно, что это распределение не является оптимальным. Это может быть обусловлено:

- неудачным начальным распределением работ;
- изменением эффективной производительности процессоров или изменением трудоемкости обработки узлов уже в ходе выполнения работы.

Изменение трудоемкости может быть вызвано следующими причинами:

1. Не всегда возможно указать трудоемкость обработки узлов сетки. Граничные и внутренние узлы сетки требуют разных времен обработки, точное соотношение которого неизвестно.
2. Используемые процессоры могут обладать разной, неизвестной заранее, производительностью.
3. Трудоемкость обработки каждого из узлов расчетной сетки может отличаться на разных моментах модельного времени.
4. Эффективная производительность процессоров может меняться с течением времени.

Приведем пример вычисления нового распределения узлов сетки по процессорам при условии, что известно текущее распределение узлов и время, затраченное каждым из процессоров, на обработку этих узлов.

Пусть расчетная сетка содержит n узлов, и на шаге $\boldsymbol{j}$ процессор $\boldsymbol{i}$ обработал $\boldsymbol{n_i}$ точек за время $\boldsymbol{t_i}$, при этом время $\boldsymbol{t_i}$ не включает в себя затраты на ожидание других процессоров. Тогда, в соответствии с основной идеей метода диффузной балансировки загрузки, можно определить новое распределение узлов сетки по процессорам:

$$n_i' = n \frac{\frac{n_i}{t_i}}{\sum_{j=1}^{p} \frac{n_j}{t_j}}. \tag{18}$$

В этом случае равномерное распределение обеспечивается при условии равенства временной характеристики трудоемкости обработки всех узлов сетки на каждом из шагов и изменение трудоемкости вызвано причинами 2 и 4. Равномерное распределение одинаковых по трудоемкости заданий пропорционально производительностям процессоров. Если процессоры обладают одинаковой производительностью, а трудоемкость обработки узлов сетки различна по причинам 1 и 3, требуется другой подход к перераспределению узлов. Пусть все узлы сетки имеют единую нумерацию, при этом каждый процессор обрабатывает непрерывный диапазон номеров узлов; $proc_j$ - номер процессора, на котором обрабатывалась точка с номером $\boldsymbol{j}$, $\boldsymbol{\tau_{(i)} = t_i/n_i}$ - трудоемкость расчета каждой из точек, обработанных на процессоре $\boldsymbol{i}$, $\boldsymbol{\delta_j = \tau_{proc(j)}}$ - время обработки узла с номером j. Тогда целью балансировки при этих условиях является минимизация $\boldsymbol{\theta}$.

$$\theta = \min_i \max \sum_{j \in proc_i''} \sigma_j . \qquad (19)$$

еление нового распределения $\boldsymbol{n_i''}$ можно

итма

```
for( j = θ; j < n; j++)
{ t = t + δj
 if (( t ≥ Tmid) { ni'' = j - m; m = j; t = θ; }}
```

Для сравнения двух выше приведенных распределений используем таблицу (см. ниже):

Номер процессора *i*	**1**	**2**	**3**	**4**
n_i	5	10	14	7
t_i	1	2	1	3
n_i'	7	7	19	3
n_i''	10	10	11	5

В первой и второй строках указано исходное распределение узлов сетки по процессорам и время, затраченное каждым из четырех процессоров на обработку. Третья строка описывает величины соответствующие новому распределению сетки. В четвертой строке указаны величины,

соответствующие новому распределению сетки согласно алгоритму. Полученные решения подчеркивают необходимость выяснения на этапе расчета причин возникновения дисбаланса. Что позволит выбрать стратегию перераспределения вычислений между процессорами, соответствующую либо первому распределению, либо второму, либо их комбинации.

Диффузная балансировка загрузки способствует сокращению длительности расчета, благодаря выравниванию вычислительной нагрузки приходящейся на каждый из процессоров. Но также требует дополнительных затрат времени на сбор статистики о выполненном расчете, на определение нового плана распределения узлов сетки и на само перераспределение узлов.

Диффузная балансировка должна проводиться при снижении эффективности расчета ниже допустимого уровня по причине дисбаланса, накопившегося за множество шагов.

Решение о количестве узлов сетки, которое следует оставить на каждом из процессоров должно приниматься централизовано, несмотря на локальный характер перераспределения узлов расчетной сетки между процессорами.

Глава 10. Типы параллельных алгоритмов

Рассмотрим такие типы параллельных алгоритмов, как поиск и сортировка [4].

10.1. Параллельный поиск

Рассмотрим задачу поиска элемента в списке. Пусть число процессоров равно числу элементов списка.

В следующем алгоритме предполагается, что список расположен в ячейках с M_1 по M_N, искомое значение - в ячейке M_{N+1}, а номер ячейки, в которой оно обнаружено, должен быть записан в M_{N+2}.

for j = 1 to N do

P[j] читает M[j] в X и M[N + 1] в target

```
if X = target then
записать j в M[ N + 2 ]
end if
end for
```

Все пустые ячейки памяти инициированы нулевым значением, следовательно по окончании работы алгоритма в ячейке M_{N+2} будет нуль, если искомое значение в списке не обнаружено. Если значение в списке найдено, то обнаруживший его процессор запишет в $\mathbf{M_{N+2}}$ номер содержащей его ячейки. На каждом из N процессоров этот алгоритм выполняет один цикл чтение / обработка / запись, поэтому общее время работы равно $O(1)$, а стоимость (время работы, умноженное на число используемых процессоров) равна **O(N).** Оптимальный последовательный алгоритм поиска, двоичный поиск, имеет стоимость ***O(log N).***

Если уменьшить стоимость за счет уменьшения числа процессоров, получим следующий алгоритм

```
При наличии p <= N процессоров
for j = l to p do
P[ j ] выполняет последовательный двоичный поиск
в ячейках с M[ (j - l) * (N / p) + l ] по M[ j * (N / p) ]
и записывает номер ячейки, содержащей X, в M[ N + 2 ]
end for
```

При наличии одного процессора он будет вести поиск в ячейках с M_1 до M_N. Поэтому на одном процессоре этот алгоритм сводится к последовательному двоичному поиску. Значит его стоимость равна **O(log N),** а время выполнения тоже ***O(log N).***

При использовании ***N*** процессоров мы возвращаемся к первому параллельному варианту со стоимостью ***O(N)*** и временем выполнения *O(1).* В промежуточных вариантах, для p списков, содержащих **N / p** элементов каждый, время выполнения равно ***O(log(N /p))***, а его стоимость ***O(p log(N /***

p)). Если ***p = log N, то log (N / log N) = log N - log (log N) ≈ log N.*** Время выполнения равно ***O(log N)*** при стоимости ***O((log N) * (log N)) = O(log2 N).***

Порядок времени выполнения параллельного алгоритма совпадает с порядком последовательного двоичного поиска, однако константа в оценке оказывается меньше, поэтому параллельный алгоритм будет выполняться быстрее. Стоимость ***O(log2 N)*** превышает стоимость **O(log N)** оптимального последовательного поиска, однако не настолько, чтобы сделать параллельный алгоритм бессмысленным.

10.2. Сортировка на линейных сетях

Начнем с метода сортировки, основанного на линейной конфигурации сети. Если число процессоров равно числу сортируемых значений, то сортировку можно осуществить, передавая сети в каждом цикле одно значение.

Первый процессор читает поданное значение, сравнивает его с текущим и передает большее значение своему соседу. Остальные процессоры делают то же самое: сохраняют меньшее из двух значений и пересылают большее следующему звену в цепочке. Получаем следующий алгоритм:

```
for i = l to N - 1 do
занести следующее значение в M[1]
Parallel Start
P[j] читает M[j] в Current
for k = l to j - 1 do
P[k] читает M[k] в New
if Current > New then
P[k] пишет Current в M[k + l]
Current = New
else
P[k] пишет New в M[k + l]
end if
end for k
Parallel End
```

```
занести следующее значение в M[1]
Parallel Start
P[j] читает M[j] в Current
for k = l to j - 1 do
P[k] читает M[k] в New
if Current > New then
P[k] пишет Current в M[k + l]
Current = New
else
P[k] пишет New в M[k + l]
end if
end for k
Parallel End
Parallel Start
for j = l to N - 1 do
P[j] пишет Current в M[j]
end for j
Parallel End
```

На первых этапах выполнения вычислений первый процессор считывает значение элемента списка в переменную ***New***. Далее, в процессе работы алгоритма, очередному параллельному проходу предшествует запись значения текущего элемента списка, при его наличии, в начальную ячейку памяти *Current*. Процессу соответствует цикл ***for***, который содержит параллельные блоки. Первый из этих блоков отвечает за чтение первого значения списка, а второй блок – за сравнение элементов. Элементы записываются в память по окончании вычислений.

На рис. 32 а и 32 б [2] представлена работа этого алгоритма на входном списке 15, 18, 13, 12, 17, 11, 19, 16, 14.

Шаг А – происходит запись в память 1-го элемента списка, который считывается процессором 1.

Шаг В – происходит запись в память 2-го элемента, который сравнивается со значением в процессоре $\boldsymbol{P_1}$. Значение, которое оказывается большим записывается в $\boldsymbol{M_2}$.

Шаг С – происходит запись в M_1 третьего элемента списка. Происходит сравнение этого элемента с содержимым процессора P_1. Происходит считывание значения из M_2, которое осуществляет процессор $\boldsymbol{P_2}$.

Шаг D - выполнение операции сравнения процессорами $\boldsymbol{P_1}$ и P_2.

1-й, 3-й, 5-й и т.д. шаги – чтение

2-й, 4-й, 6-й и т.д. шаги – сравнение, в котором участвуют все процессоры.

Алгоритм сортировки выполняет вычисления за 16 шагов + 1 (запись результата.

Для ***N*** процессоров - ***2 * (N - 1) + 1*** действий. Вычислительная сложность - ***O(N).*** Стоимость алгоритма - $\boldsymbol{O(N^2)}$.

Рис. 32 а. Алгоритм сортировки на линейных сетях

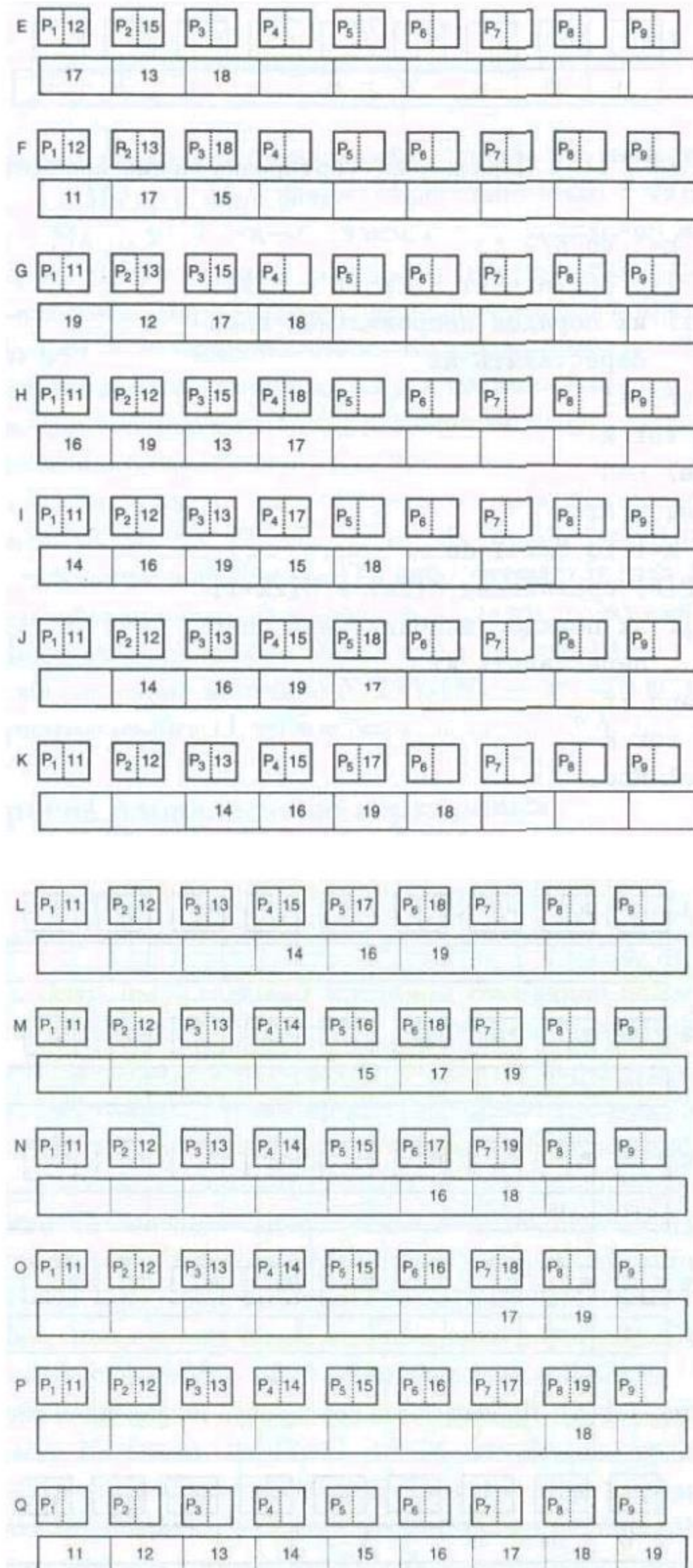


Рис. 32 б. Алгоритм сортировки на линейных сетях

Четно-нечетная сортировка перестановками

Выше нам удавалось снизить стоимость алгоритмов за счет сокращения числа процессоров. Необходимое число процессоров можно уменьшить вдвое, воспользовавшись следующим способом сортировки, который сравнивает соседние значения и при необходимости переставляет их [2].

```
for j = 1 to N / 2 do
Parallel Start
for k = l to N / 2 do
P[k] сравнивает M[2k - l] и M[2k]
if их порядок неправильный then
переставить их
end if
end for k
Parallel End
Parallel Start
for k = l to N / 2 - 1 do
P[k] сравнивает M[2k] и M[2k + 1]
if их порядок неправильный then
переставить их
end if
end for k
Parallel End
end for j
```

Сравнение пар M_1 ***и*** M_2***;*** M_3 ***и*** M_4***; …,*** M_{N-1} ***и*** M_N выполняется при каждом проходе алгоритма. Далее выполняется сравнение пар M_2 ***и*** M_3***;*** M_4 ***и*** M_5***; …,*** M_{N-2} ***и*** M_{N-2}***.*** В случае необходимости выполняется перестановка. На рис.33 представлен случай расположения наименьшего элемента в конце списка. За каждый проход осуществляется сдвиг элемента на две позиции в сторону правильного положения. Элемент ставится в нужное место путем

сдвига на *N-1* позицию ***N/2*** раз при повторении цикла. За один проход цикла выполняются две операции сравнения. Рисунок иллюстрирует применение процедуры для списка 15, 18, 13, 12, 17, 11, 19, 16, 14. Символом «О» помечены нечетные шаги. Символом «Е» - четные.

	15	18	13	12	17	11	19	16	14
O1	15	18	12	13	11	17	16	19	14
E1	15	12	18	11	13	16	17	14	19
O2	12	15	11	18	13	16	14	17	19
E2	12	11	15	13	18	14	16	17	19
O3	11	12	13	15	14	18	16	17	19
E3	11	12	13	14	15	16	18	17	19
O4	11	12	13	14	15	16	17	18	19
E4	11	12	13	14	15	16	17	18	19

Рис. 33. Выполнение процедуры сортировки на списке 15, 18, 13, 12, 17, 11, 19, 16, 14

Стоимость алгоритма составляет $N/2 * O(N)$. Вычислительная сложность - $\boldsymbol{O(N^2)}$. Общее время работы составляет $\boldsymbol{O(N)}$.

Глава 11. Типы параллельных алгоритмов

В процессе исследовании параллельных алгоритмов на графах для представления графов используются матрицы смежности.

11.1. Параллельный алгоритм поиска кратчайшего пути

Определим матрицу смежности взвешенного графа:

$$AdjMat[i,j] = \begin{cases} w_{ij} \text{ при } v_i v_j \in E, \\ 0 \quad \text{при } i = j, \\ \infty \text{ при } v_i v_j \notin E \end{cases} \qquad \text{для } i,j \in [1,N]. \tag{19}$$

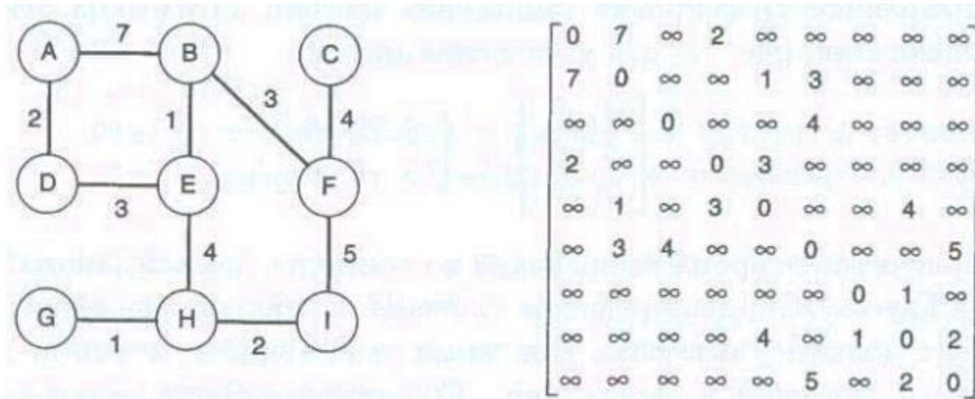


Рис. 34. Взвешенный граф и его матрица смежности

На рис. 34 изображен взвешенный граф и его матрица смежности.

В матрице смежности представлены прямые пути между вершинами графа, т. е. пути, состоящие из одного ребра. Восстановим кратчайшие пути по графу, состоящие из произвольного числа ребер, по матрице смежности. В исходной матрице $\boldsymbol{A = A^1}$ содержатся длины кратчайших путей из нуля или одного ребра. В матрице $\boldsymbol{A^j}$ будут записаны длины кратчайших путей из не более чем $\boldsymbol{j}$ ребер. Ясно, что в матрице $\boldsymbol{A^{N-1}}$ будут записаны длины всех кратчайших путей с произвольным числом ребер, потому что всякий путь из $\boldsymbol{N}$ и более ребер содержит цикл, а значит, кратчайший путь должен содержать не более $\boldsymbol{N - 1}$ ребер. Построим матрицу $\boldsymbol{A^2}$ по матрице $\boldsymbol{A^1}$. Кратчайший путь из двух ребер между вершинами $\boldsymbol{x}$ и $\boldsymbol{y}$ проходит еще в точности через одну вершину. Например, всякий путь длины два между вершинами $\boldsymbol{A}$ и $\boldsymbol{E}$ проходит либо через вершину $\boldsymbol{B}$, либо через вершину $\boldsymbol{D}$**.** Сравнив сумму весов ребер $\boldsymbol{AB}$ и $\boldsymbol{BE}$ с суммой весов ребер $\boldsymbol{AD}$ и $\boldsymbol{DE}$, видим, что путь через вершину $\boldsymbol{D}$ короче пути через вершину $\boldsymbol{B}$. В общем случае, если посмотреть на сумму весов ребер $\boldsymbol{A*}$ и $\boldsymbol{E*}$**,** где * пробегает все вершины от $\boldsymbol{A}$ до $\boldsymbol{I}$ (за исключением самих вершин $\boldsymbol{A}$ и $\boldsymbol{E}$), то минимальное значение суммы весов будет давать длину кратчайшего пути из двух или менее ребер. Отсюда получаем:

$$A_{ij}^2 = \min_{k \in V}(A_{ik}^1 + A_{kj}^1)\,. \tag{20}$$

$$
\begin{bmatrix}
0 & 7 & \infty & 2 & 5 & 10 & \infty & \infty & \infty \\
7 & 0 & 7 & 4 & 1 & 3 & \infty & 5 & 8 \\
\infty & 7 & 0 & \infty & \infty & 4 & \infty & \infty & 9 \\
2 & 4 & \infty & 0 & 3 & \infty & \infty & 7 & \infty \\
5 & 1 & \infty & 3 & 0 & 4 & 5 & 4 & 6 \\
10 & 3 & 4 & \infty & 4 & 0 & \infty & 7 & 5 \\
\infty & \infty & \infty & \infty & 5 & \infty & 0 & 1 & 3 \\
\infty & 5 & \infty & 7 & 4 & 7 & 1 & 0 & 2 \\
\infty & 8 & 9 & \infty & 6 & 5 & 3 & 2 & 0
\end{bmatrix}
$$

Рис. 35. Новая матрица смежности

Применив эту процедуру к матрице смежности на рис. 34, получим матрицу, показанную на рис. 35.

Матрицу $\boldsymbol{A^3}$ можно построить по матрицам $\boldsymbol{A^1}$ и $\mathbf{A^2}$, заметив, что кратчайший путь из трех или менее ребер должен состоять из кратчайшего пути из двух или менее ребер, за которым следует кратчайший путь из одного или менее ребер и наоборот. Матрицу $\boldsymbol{A^4}$ можно построить либо по матрицам $\boldsymbol{A^1}$ и $\boldsymbol{A^3}$, либо только по матрице $\boldsymbol{A^2}$. Поэтому до конца добраться легче, вычисляя матрицы $\boldsymbol{A^2, A^4, A^8, \ldots, A^{N'}}$, где $\boldsymbol{N'}$ - первая степень двойки которая превышает число вершин, уменьшенное на 1. Все кратчайшие расстояния в графе с рис. получим, подсчитав матрицу A^8.

Параллельный подсчет кратчайших расстояний в графе можно реализовать на основе этих матричных операций. Заменив в алгоритме умножения матриц сложение операцией взятия минимума, а умножение сложением, получим алгоритм, вычисляющий нужные нам матрицы. Применение модифицированного алгоритма матричного умножения к матрицам A^1 и A^1 даст матрицу A^2, а к матрицам A^2 и A^2 - матрицу A^4. Теперь параллельный алгоритм подсчета кратчайших расстояний превращается просто в параллельный алгоритм умножения матриц, поэтому анализ последнего применим и в данном случае.

11.2. Параллельный алгоритм поиска минимального остовного дерева

Алгоритм Прима порождения минимального остовного дерева строит дерево постепенно, добавляя на каждом шаге вершину, соединенную с уже построенной частью дерева ребром минимальной длины.

При этом вершины, расположенные рядом с уже построенной частью дерева, образуют «кайму». Располагая большим числом процессоров, можно вместо этого рассматривать одновременно все вершины и при каждом проходе выбирать ближайшую из них.

Спроектируем алгоритм для *p* процессоров, предполагая, что $p < N$. Это означает, что каждый процессор будет отвечать за N / p вершин. Начнем построение минимального остовного дерева с одной из вершин. В начальный момент это ближайшая ко всем остальным вершина дерева, поскольку она единственная. При каждом проходе каждый процессор исследует каждую из вершин и выбирает из них ближайшую к вершинам дерева. Затем процессор передает информацию о выбранной вершине центральному процессору, который выбирает среди них вершину с абсолютно минимальным расстоянием до построенной части дерева. Эта вершина добавляется к дереву, а информация о ней передается остальным процессорам, и они обновляют свои данные о дереве. Процесс повторяется $N - 1$ раз, пока к дереву не будут добавлены все вершины.

В следующем алгоритме множество вершин, за которые отвечает процессор P_j, обозначается через V_j, а через v_k обозначается вершина графа. В каждом из процессоров локально используется два массива. Элемент **closest(v)** первого из них содержит метку вершины построенной части минимального остовного дерева, ближайшей к вершине *v*, а элемент ***distance*** (***v[i], v[j])*** - кратчайшее расстояние между вершинами v_i и v_j.

Алгоритм в модели CREW PRAM:

v[0] помечается как первая вершина дерева

Parallel Start

for j = l to p do

for каждой вершины в V[j] do

занести v[0] в closest

end for

end for j

Parallel End

for j = 1 to N - 1 do

Parallel Start

for k = 1 to p do

P[k] вычисляет наименьшие расстояния от своих вершин до минимального остовного дерева

P[k] возвращает distance(v, closest(v)), v и closest(v)

end for k

Parallel End

Центральный процессор P находит наименьшее из найденных расстояний и добавляет соответствующую вершину v в минимальное остовное дерево. Центральный процессор ***P*** сообщает вновь добавленную вершину v всем остальным процессорам

Parallel Start

for k = 1 to p do

if v лежит в V[k] then

P[k] помечает, что v занесена в дерево

end if

Pk меняет значения closest и distance для всех вершин еще не занесенных в дерево, учитывая вновь добавленную в дерево вершину

end for k

Parallel End

end for j

Анализ

Первый цикл алгоритма устанавливает начальные значения. Его выполнение требует ***N / p*** шагов, поскольку количество начальных значений, устанавливаемых каждым процессором, пропорционально числу вершин, за которые он отвечает. Первый параллельный блок в главном цикле for всякий раз выполняет ***N / p - 1*** сравнений, ровно столько сравнений требует последовательный алгоритм поиска максимума или минимума. Следующий шаг состоит в том, чтобы выбрать минимальное расстояние из всех сообщенных p процессорами, на что уходит еще p - 1 сравнений. Шаг распределения значений в модели CREW занимает, как было показано, два цикла. В последнем параллельном блоке одно сравнение уходит на то, чтобы проверить, отвечает ли данный процессор за вновь добавленный узел, а исправление массивов closest и distance требует ***N / p*** операций. Итак, при одном проходе главного цикла обработки выполняется ***2(N / p) + p + 1*** операций, а число проходов равно ***N - 1***. Поэтому общее число операций в главном цикле ***(N - 1) * (2 * (N/p) + p + 1)***, а общая сложность алгоритма равна $O(N^2/p)$ при стоимости $p * O(N^2/p)$, или $O(N^2)$. Как мы уже видели в других случаях, оптимальность достигается при числе процессоров около ***N / log N***.

11.3. Генетические параллельные алгоритмы

Механизм наследственности в биологических популяциях впервые был использован Джоном Холландом для решения задач оптимизации. Он был назван репродуктивным планом Холланда. Свое название - генетические алгоритмы, получили в работах Гольдберга и Де Йонга.

Цель генетического алгоритма при решении задачи оптимизации состоит в том, чтобы найти лучшее возможное, но не гарантированно оптимальное решение. Для реализации генетического алгоритма необходимо выбрать подходящую структуру данных для представления решений. В постановке задачи поиска оптимума, экземпляр этой структуры должен содержать информацию о некоторой точке в пространстве решений. Структура данных генетического алгоритма состоит из набора хромосом. Хромосома, как правило, представляет собой битовую строку. Хромосомы также могут быть реализованы как векторы вещественных чисел. В настоящее время используются структуры фиксированной длины - битовые строки. Каждая хромосома (строка) представляет собой последовательное объединение ряда подкомпонентов, которые называются генами. Гены находятся в локусах хромосомы. Их значения называются алелями. В представлении хромосомы бинарной строкой, ген является битом этой строки, локус - есть позиция бита в строке, а аллель - это значение гена, 0 или 1. Биологический термин «генотип» ставится в соответствие полной генетической модели особи. Генотип соответствует структуре в генетическом алгоритме.

Термин «фенотип» соответствует вектору в пространстве параметров задачи. Фенотип относится к внешним наблюдаемым признакам и В генетике под мутацией понимается преобразование хромосомы, случайно изменяющее один или несколько генов. Часто встречается следующий вид мутаций - случайное изменение только одного из генов хромосомы. Термин кроссинговер обозначает порождение из двух хромосом двух новых путем обмена генами. В литературе по генетическим алгоритмам также употребляется термин кроссовер, скрещивание или рекомбинация. В простейшем случае кроссинговер в генетическом алгоритме реализуется так же, как и в биологии. При скрещивании происходит разрезание хромосом в случайной точке и их обмен частями между собой. В частности, если хромосомы (11, 12, 13, 14) и (0, 0, 0, 0) разрезать между вторым и третьим

генами и обменять их части, то получатся следующие потомки (11, 12, 0, 0) и (0, 0, 13, 14).

Основные структуры и фазы генетического алгоритма

Задача максимизации некоторой функции двух переменных ***f(x1, x2)***, при ограничениях: $0 < x1 < 1$ и $0 < x2 < 1$. Обычно методика кодирования реальных переменных ***x1*** и ***x2*** состоит в преобразовании их в двоичные целочисленные строки определенной длины, достаточной для того, чтобы обеспечить желаемую точность.

Предположим, что 10-ти разрядное кодирование достаточно для ***x1*** и ***x2***. Установить соответствие между генотипом и фенотипом можно, разделив соответствующее двоичное целое число на 210 - 1. Например, 0000000000 соответствует 0 / 1023 или 0, тогда как 1111111111 соответствует 1023 / 1023 или 1.

Оптимизируемая структура данных есть 20-битовая строка, представляющая собой конкатенацию (объединение) кодировок ***x1*** и ***x2***. Пусть переменная ***x1*** размещается в крайних левых 10 битах строки, тогда как ***x2*** размещается в правой части генотипа особи. Таким образом, генотип представляет собой точку в 20-мерном целочисленном пространстве (вершину единичного гиперкуба), которое исследуется генетическим алгоритмом. Для этой задачи фенотип будет представлять собой точку в двумерном пространстве параметров ***(x1, x2)***. Для решения задачи оптимизации необходимо определить для каждой структуры в пространстве поиска меру качества. Для этой цели используется функция приспособленности. При максимизации целевая функция часто сама выступает в качестве функции приспособленности, для задач минимизации целевая функция инвертируется и смещается в область положительных значений.

Рассмотрим фазы работы простого генетического алгоритма. Вначале случайным образом генерируется начальная популяция (набор хромосом).

Работа алгоритма продолжается, пока не будет смоделировано заданное число поколений или выполнен некоторый критерий останова. В каждом поколении реализуется пропорциональный отбор приспособленности, одноточечная рекомбинация и вероятностная мутация. Пропорциональный отбор реализуется путем назначения каждой особи (хромосоме) *i* вероятности ***P(i)***, равной отношению ее приспособленности к суммарной приспособленности популяции по целевой функции):

Затем происходит отбор (с замещением) всех п особей для дальнейшей генетической обработки, согласно убыванию величины ***P(i).*** Пропорциональный отбор можно реализовать с помощью рулетки – механизма, предложенного Гольдбергом в 1989 г. Каждый сектор «колеса» рулетки соответствует одному сектору для каждого члена популяции. Размер ***i***-го сектора таким образом пропорционален соответствующей величине ***P(i).*** Такой отбор позволяет более приспособленным членам популяции выбираться чаще по вероятности. После отбора выбранные n особей подвергаются рекомбинации с заданной вероятностью ***P(i)***, при этом п хромосом случайным образом разбиваются на ***n / 2*** пар. Для каждой пары с вероятность ***P(i)*** может быть выполнена рекомбинация. Если рекомбинация происходит, то полученные потомки заменяют собой родителей. Одноточечная рекомбинация работает следующим образом. Случайным образом выбирается одна из точек разрыва, т. е. участок между соседними битами в строке. Обе родительские структуры разрываются на два сегмента по этому участку. Далее сегменты различных родителей склеиваются, и в результате образуется два генотипа потомков. После стадии рекомбинации выполняется фаза мутации. В каждой строке, которая подвергается мутации, каждый бит инвертируется с вероятностью Рт. Популяция, полученная после мутации, записывается поверх старой, и на этом завершается цикл одного поколения в генетическом алгоритме.

Полученное новое поколение обладает (по вероятности) более высокой приспособленностью, наследованной от «хороших» представителей

предыдущего поколения. Таким образом, хорошие характеристики передаются по всей популяции из поколения в поколение. Скрещивание наиболее приспособленных особей приводит к исследованиям наиболее перспективных участков пространства поиска. Таким образом, сходимость популяции направлена к локально оптимальному решению задачи, а иногда, может быть, благодаря мутации, и к глобальному оптимуму.

Теперь можно сформулировать основные шаги генетического алгоритма.

Генетический алгоритм

1. Создать начальную популяцию

2. Цикл по поколениям пока не выполнено условие останова // цикл жизни одного поколения

3. Оценить приспособленность каждой особи

4. Выполнить отбор по приспособленности

5. Случайным образом разбить популяцию на две группы пар

6. Выполнить фазу вероятностной рекомбинации для пар популяции и заменить родителей

7. Выполнить фазу вероятностной мутации

8. Оценить приспособленность новой популяции и вычислить условие останова

9. Объявить потомков новым поколением

10. Конец цикла по поколениям

Модификации генетического алгоритма

Очевидно, что тонкая настройка базового генетического алгоритма может быть выполнена путем изменения значений вероятностей рекомбинации и мутации, существует много исследований и предложений в данной области. В настоящее время предлагаются разнообразные модификации генетических алгоритмов в части методов отбора по

приспособленности, рекомбинации и мутации. Приведем несколько примеров.

Метод турнирного отбора (Бриндел, 1981 г.; Гольдберг и Деб, 1991 г.) реализуется в виде *n* турниров для выборки п особей. Каждый турнир состоит в выборе *k* элементов из популяции и отбора лучшей особи среди них.

Элитные методы отбора (Де Ионг, 1975 г.) гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции. Сохраняется только одна лучшая особь, которая не прошла отбор, рекомбинацию и мутацию. Этот метод может быть внедрен практически в любой стандартный метод отбора.

Двухточечная рекомбинация (Гольдберг 1989 г.) и равномерная рекомбинация (Сисверда, 1989 г.) являются вполне достойными альтернативами одноточечному оператору. При двухточечной рекомбинации происходит выбор двух точек разрыва. Далее родительские хромосомы обмениваются сегментом. Сегмент располагается между двумя точками разрыва. При равномерной рекомбинации каждый бит первого родителя наследуется первым потомком с известной вероятностью. Иначе бит передается второму потомку.

Механизмы мутаций могут быть также заимствованы из молекулярной биологии, например, обмен концевых участков хромосомы (механизм транслокации), обмен смежных сегментов (транспозиция). По мнению М.В. Ульянова, интерес представляет механизм инверсии, т. е. перестановки генов в хромосоме, управляющим параметром при этом может выступать инверсионное расстояние - минимальное количество единичных инверсий генов, преобразующих исходную хромосому в мутированную.

Применение генетических алгоритмов

Генетические алгоритмы носят эвристический характер. Можно предположить, что генетический алгоритм способен осуществлять эффективный поиск оптимального решения, при условии, что:

- пространство поиска достаточно велико, и предполагается, что целевая функция не является гладкой и унимодальной в области поиска, т. е. не содержит один гладкий экстремум;
- нет необходимости в поиске универсального оптимального решения, достаточно быстро найти подходящее решение.

Если целевая функция обладает свойствами гладкости и унимодальности, то любой градиентный метод, такой как метод наискорейшего спуска, будет более эффективен. Генетический алгоритм является в определенном смысле универсальным методом, в связи с тем, что не учитывает специфику задачи. Если существует возможность получить сведения о целевой функции и сведения о пространстве поиска, то становится возможным использование эвристического поиска, который превосходит по эффективности универсальный алгоритм.

Сегодня генетические алгоритмы успешно применяются для решения классических NP-полных задач, задач оптимизации в пространствах с большим количеством измерений, ряда экономических задач оптимального характера, например, задач распределения инвестиций.

Использование эволюционных алгоритмов вместе с параллельными вычислениями позволяет эффективно решать задачи оптимизации, размер которых превышает 2^{100}. Такие эволюционные алгоритмы называются распределенными генетическими алгоритмами.

Можно выделить два способа применения параллельных технологий к генетическим алгоритмам [3]:

1. Изменение структуру алгоритма (технология master-slave). При использовании этого способа параллельной обработке подвергаются такие независимые элементарные операции, как:

- построение особи, в операторе создания начальной популяции;
- построение потомка и его мутация, в операторе воспроизводства;
- оценка приспособленности особи.

Технология master-slave предполагает, что работа над популяцией осуществляется на главном процессоре, а приспособленность особей вычисляется на подчиненных процессорах. Недостатком данного метода является возможное снижение эффективности вычислений при увеличении количества процессоров. Это происходит за счет потери времени на обмен данными между главным процессором и подчиненным и потери времени на ожидание завершения работы всех подчиненных процессоров для продолжения вычислений.

2. Изменение структуры алгоритма в результате применения ограничений возможности скрещивания (мелкоструктурированный параллельный генетический алгоритм и крупноструктурированный параллельный генетический алгоритм. Крупноструктурированный параллельный генетический алгоритм имеет отношение к мултипопуляционным моделям, таким как метод случайных иммигрантов в исследованиях Grefenstette, метод ниш и островным моделям. При таком способе вычислений популяция делится на локальные популяции, развитие которых происходит параллельно и независимо, но при этом сохраняется возможность обмена особями между популяциями. Особи разных локальных популяций не могут образовывать брачные пары. Естественный отбор действует только внутри локальной популяции. Каждая локальная популяция обрабатывается на отдельном процессоре. Основой алгоритма является определенный метод разбиения популяции. Для минимизации затрат на обмен данными, топология связи между локальными популяциями отражает топологию связи между процессорами или компьютерами вычислительной системы. Большое количество связей между популяциями снижает эффективность вычислений. А уменьшение количества связей приводит к преждевременной сходимости в ряде локальных популяций и потери

эффективности вычислений при поиске. При этом способе действует механизм обмена лучшими решениями. Лучшее решение распространяется в локальных популяциях для повышения общей приспособленности особей и предотвращения преждевременной сходимости, при условии, что отслеживается разнообразие в каждой локальной популяции. В случае с мелкоструктурированным параллельным генетическим алгоритмом имеется всего одна популяция с ограничениями на скрещивание и конкуренцию. Внутри популяции действует регулярная структура, предусматривающая для каждой особи окрестность. Выбор конкурента или пары осуществляется только внутри окрестности. Такая структура удобно реализуется на вычислительной системе с распределенной архитектурой. Особь соответствует одному процессору. Стратегии, применяемые в таких алгоритмах, называются поколенческими стратегиями естественного отбора с полной заменой родителей. Перекрытие окрестностей вызывает распространение лучшей особи по всей популяции.

Благодаря возможности окрестности меняться в процессе работы, мелкоструктурированный алгоритм имеет большую гибкость, чем крупноструктурированный.

Глава 12. Расширенные методы построения параллельных алгоритмов

12.1. Генерация псевдослучайных чисел

Последовательности случайных чисел (ПСЧ) используются при решении задач многомерной многоэкстремальной оптимизации, при вычислении кратных интегралов, при изучении поведения сложных систем методами имитационного моделирования и молекулярной динамики, при разработке игр и анализе стратегий поведения, при создании реалистичных сцен виртуальной реальности и во многих других приложениях. Использование случайных чисел, предполагает проведение множества

вычислительных экспериментов с уникальным набором случайных чисел для каждого эксперимента. Ответ формируется путем обработки результатов экспериментов и точность его тем выше, чем больше экспериментов проведено. Данная модель вычислений характеризуется наличием внутреннего параллелизма. Независимые эксперименты могут быть выполнены одновременно, каждый на своем процессоре. При этом необходимо указать метод формирования на каждом из процессоров уникального набора случайных чисел, с тем, что бы эксперименты, выполненные на разных процессорах, не дублировали друг друга

Требования к генераторам псевдослучайных чисел для МВС

При решении задач стохастическими (недетерминированными, использующими случайные числа или их имитацию) методами используется два вида генераторов последовательностей чисел:

- генераторы случайных чисел, основанные на некоторых физических принципах;
- генераторы псевдослучайных чисел, основанные на детерминированных алгоритмах.

Генераторы случайных чисел, основанные на физических принципах:

- обеспечивают формирование произвольно длинных последовательностей случайных чисел;
- обеспечивают формирование на различных процессорах различных последовательностей;
- не обеспечивают воспроизводимость результатов, что является преимуществом с точки зрения криптографических приложений и является недостатком с точки зрения удобства отладки вычислительных приложений. Отсутствие воспроизводимости затрудняет отладку приложений;
- свойства формируемых последовательностей могут меняться в зависимости от таких параметров окружающей среды. Например, класс физических генераторов, основанных на комбинации простых осцилляторов,

чувствителен к наличию внешнего высокочастотного электромагнитного поля, что вызывает резонанс между осцилляторами и приводит к получению последовательности, неудовлетворяющей тестам, и непригодной для проведения вычислительных экспериментов;

- не поддаются априорному тестированию, так как свойства генератора, могут быть разными в момент тестирования и в момент решения задачи;
- являются аппаратно-зависимыми, что снижает переносимость разрабатываемых программных кодов.

Генераторы псевдослучайных чисел, основанные на алгоритмических принципах:

- обеспечивают формирование последовательностей, длина которых ограничена некоторой конечной величиной – периодом, но в связи с наличием методов формирования последовательностей, обладающих произвольно большим периодом, этот недостаток можно обойти;
- обеспечивают формирование на различных процессорах различных последовательностей;
- методы алгоритмической генерации обеспечивают возможность согласованного формирования на множестве процессоров фрагментов единой последовательности, что при создании масштабируемых параллельных алгоритмов;
- могут быть предварительно протестированы на стандартных тестах и на ряде задач требуемого класса.

Использование физических генераторов предпочтительнее, если не требуется воспроизводимость результатов и качество последовательностей удовлетворительно.

Методы формированиия последовательностей псевдослучайных чисел на произвольном числе процессоров:

Создание генератора для каждого из процессоров. Метод не пригоден при разработке масштабируемых алгоритмов и подходит только для небольшого числа процессоров

Использование на процессоре с номером rank элементов последовательности ***u[(rank+ip) mod T],*** где **p** – число процессоров, а ***T*** – период исходной последовательности ***u*** (leapfrog-метод). Метод может быть использован при формировании последовательностей для векторных вычислительных систем.

Метод имеет следующие недостатки:

1. Не обеспечивает идентичности результатов. Пусть параллельный алгоритм формирует множества псевдослучайных точек, распределенных в некоторой области трехмерного пространства. Для каждого числа процессов метод будет формироваться свое множество точек. Например, при использовании одного процесса будут сформированы точки, координаты которых определяются псевдослучайными числами с номерами: (0,1,2),(3,4,5),(6,7,8),(9,10,11),(12,13,14),(15,16,17). При использовании двух процессов, последовательность первого - числа с четными номерами, второго - с нечетными, поэтому будут сформированы другие точки:

первый процесс: (0,2,4),(6,8,10),(12,14,16).

второй процесс: (1,3,5),(7,9,11),(13,15,17).

Таким образом, нельзя будет сравнить результаты, полученные при решении задачи одним и двумя процессами, так как сформированные множества точек не совпадают.

2. **Высокое качество последовательности *u [i],* не гарантирует при произвольном числе процессов p качества последовательностей *u[(rank+ip) mod T].*** На рис. 36 приведен пример исходной последовательность ***uk=(5uk–1+1) mod 64***, которая имеет случайный вид, но её подпоследовательность vk, включающая каждый четвертый элемент исходной последовательности ***(vk=u4k),*** является линейной функцией (в пределах периода, рис. 36), что не отвечает требованию случайности.

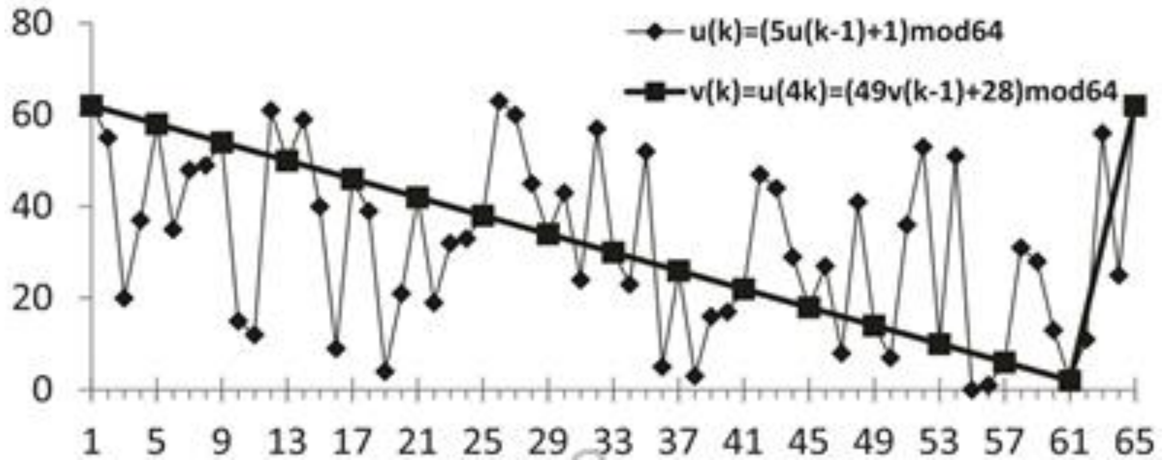


Рис. 36. Исходная последовательность uk=(5u(k-1)+1) mod 64 и подпоследовательность vk

Использование на процессоре с номером *rank* элементов последовательности ***u [num (rank)]… u[num (rank+1) – 1***]. Каждый процессор использует большой непрерывный фрагмент одной и той же последовательности ***u*** (skip-ahead метод). Если рассмотреть применение метода в пределах приведенного выше примера с точками, то метод позволяет использовать в первом процессе числа с номерами (0,…,8), во втором – (9,…,17), формируя тот же набор пространственных точек, что и при последовательной обработке. Эффективные параллельные методы требуют возможности формирования фрагментов последовательности, начинающихся в требуемых местах, без вычисления на каждом процессоре всех предшествующих фрагменту элементов исходной последовательности.

Требования, предъявляемые к генераторам псевдослучайных чисел, предназначенных для использования на многопроцессорных системах:

1) возможность формирования последовательности достаточно большой длины;

2) возможность определения любого элемента последовательности за короткое время без вычисления всех ее предыдущих членов.

Линейно-конгруэнтные генераторы

Линейные конгруэнтные генераторы (22) позволяют с помощью соотношений (23) - (24) одновременно вычислять на векторном компьютере, оснащенном k процессорами, очередные k элементов последовательности, используя для вычисления каждого элемента порядка $O(1)$ операций [2].

$$U_{n+1} = (aU_n + c) \bmod m. \tag{22}$$

Для этого достаточно один раз предварительно вычислить величины $a^k \bmod m$ и

$$\left(\frac{a^k - 1}{a-1}c\right) \bmod m.$$

$$U_n = \left(a^n U_0 + \frac{a^n - 1}{a-1}c\right) \bmod m, \tag{23}$$

$$U_{k(n-1)} = \left(a^k U_{kn} + \frac{a^k - 1}{a-1}c\right) \bmod m. \tag{24}$$

Выражение (23) позволяет вычислять элементы с произвольным номером n, используя порядка $O(\log n)$ операций. Например, понизим вдвое степени стоящих в скобках слагаемых. Используя (25), можно преобразовать (23) к виду (26) где $n=k+t$, $k=n/2$, x – наименьшее целое, не меньшее x.

Рекурсивно повторяя процесс понижения степени, получим требуемый результат.

$$a^{k+t} - 1 = a^t\left(a^k - 1\right) + a^t - 1, \tag{25}$$

$$U_n = \left\{ \left(a^k U_0\right) \bmod m \cdot \left(a^t U_0\right) \bmod m + a^t \bmod m \cdot \left(\frac{a^k - 1}{a-1}c\right) \bmod m + \right.$$
$$\left. + \left(\frac{a^t - 1}{a-1}c\right) \bmod m \right\} \bmod m. \tag{26}$$

Алгоритм, основанный на использовании бинарного представления номера вычисляемого элемента ***n*,** потребует еще меньшего числа операций:

$$n = \sum_{i=0}^{[\log(n)]} \beta_i 2^i ,\ \beta_i = \frac{n}{2^i} \bmod 2. \tag{27}$$

Рассмотрим функцию вычисления значения элемента последовательности, имеющего номер на 2^i больше, чем номер элемента, заданного в качестве аргумента (28).

$$F_i\left(U_j\right) = \beta_i\left[\left(a_i U_j + c_i\right) \bmod m\right] + \left(1 - \beta_i\right) U_j , \tag{28}$$

$$a_0 = a,\ a_{i+1} = a_i^2 \bmod m, \tag{29}$$

$$c_0 = c,\ c_{i+1} = \left[\left(a_i + 1\right) c_i\right] \bmod m. \tag{30}$$

Пусть в качестве аргумента функции F_i задан элемент последовательности с номером j. Тогда при $b_i = 1$ значением функции является элемент последовательности с номером $j + 2^i$, а при элемент с номером j. Коэффициенты a_i и c_i определяются рекуррентно, согласно (29), (30), и могут быть вычислены однократно в начале работы. Их количество равно $M = \log N$, где N — максимальный номер требуемого в ходе выполнения всех вычислений элемента последовательности. Таким образом, зная элемент последовательности с нулевым номером, можно, используя (31), вычислить произвольный элемент последовательности за M обращений к функции F_i.

$$U_n = F_0\left(F_1\left(F_2\left(\ldots F_b\left(U_0\right)\ldots\right)\right)\right). \tag{31}$$

К недостаткам линейных конгруэнтных генераторов относится:

- малая длина периода ПСЧ, обусловленная тем, что для эффективной реализации операций умножения и сложения по модулю m, разрядность модуля ограничена разрядностью операндов, непосредственно обрабатываемых процессором;

- d-мерные точки, координаты которых сформированы с помощью (25), располагаются не более чем в d!m гиперплоскостях размерности (d – 1). Этот факт значительно ограничивает применимость линейных конгруэнтных генераторов, например, в геометрических приложениях. В частности, при a = 216 + 3, c = 0 и m = 231 (параметры, использованные в библиотеке IBM System 360/370) в трехмерном случае (d = 3) таких плоскостей не более 16.

12.2. Декомпозиция сеточных графов

При численном решении задач механики сплошной среды с помощью методов конечных разностей или конечных элементов широко используется геометрический параллелизм, при котором сетка, покрывающая расчетную область, разбивается на множество доменов, каждый из которых обрабатывается отдельным процессором. Основная проблема при использовании метода геометрического параллелизма связана с необходимостью решения задачи статической балансировки загрузки — выбора такой декомпозиции расчетной сетки, при которой вычислительная нагрузка распределена равномерно между процессорами, а накладные расходы, вызванные дублированием вычислений и необходимостью передачи данных между процессорами, малы.

На рисунке 37 приведен пример двумерной расчетной сетки - часть неструктурированной треугольной сетки, описывающей пространство вокруг крыла и закрылка самолета. Узлы сетки сгущаются к мелким деталям моделируемой конструкции и к зонам, в которых газодинамические параметры подвергаются существенные изменения. Каждая из них содержит примерно одинаковое число узлов и при выполнении численного моделирования, обрабатывается отдельным процессором. Процессоры взаимодействуют между собой и объемы передаваемых между процессорами данных определяются используемым численным методом. Для многих методов считается, что они пропорциональны длине границ доменов — числу ребер соединяющих вершины из разных доменов.

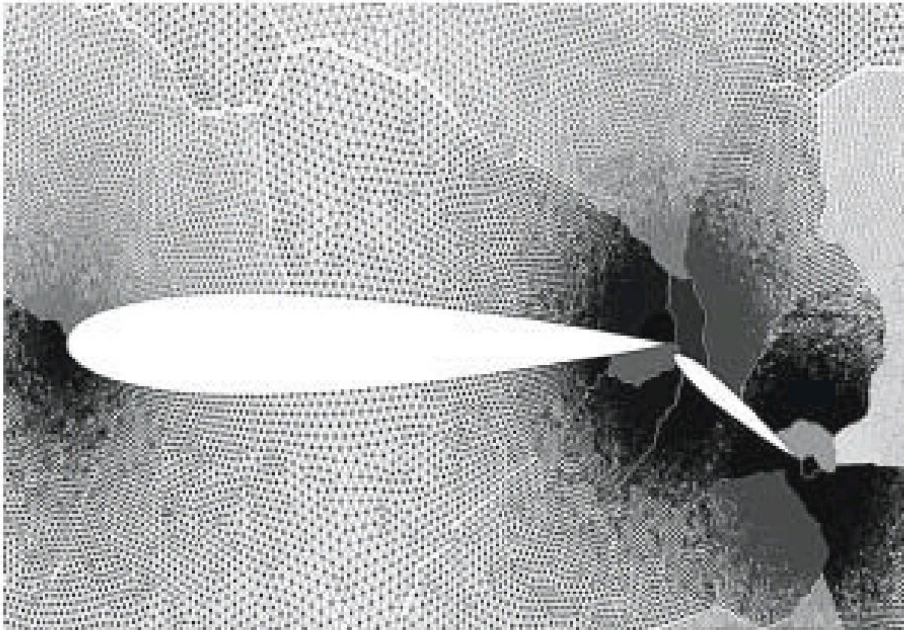


Рис. 37. Двумерная расчетная сетка

Для обеспечения эффективности метода геометрического параллелизма в размере 80% при двух тысячах процессоров, доля операций, не поддающихся распараллеливанию на все процессоры (последовательных операций) не должна превышать 10^{-4} процента.

Критерии декомпозиции — требования, предъявляемые к виду доменов: равномерное распределение по доменам суммарного веса узлов/ребер и минимизация максимального веса исходящих из домена ребер.

Пусть задан граф $\boldsymbol{G^0 = (V, E), V\{v_i\}, |V| = n}$. Пусть вершины v_i и ребра $\boldsymbol{e_{ij}}$ этого графа имеют веса $w(v_i)$ и $\boldsymbol{w(v_i, v_j)}$ соответственно. Найдем такое разбиение $\boldsymbol{R(V) =(V_1, ..., V_p)}$ вершин на заданное число доменов p, при котором $\boldsymbol{J}$ - взвешенная сумма весов вершин и разрезанных ребер - принимает минимальное значение (32):

$$\min_{R(V)}\left\{J = \max_{k=1,\dots,p}\sum_{v_i\in V_k}\left(w(v_i)+\alpha\sum_{v_j\notin V_k}w(v_i, v_j)\right)\right\}. \qquad (32)$$

множитель $\boldsymbol{\alpha}$ обеспечивает согласование единиц измерения весов вершин и ребер, что соответствует критерию. Так как выравнивается вычислительная нагрузка, приходящаяся на каждый домен (она ассоциируется с весами

вершин) и снижаются коммуникационные затраты, ассоциированные с числом разрезанных ребер. Задача декомпозиции сводится к минимизации $\boldsymbol{J}$.

Выделение обособленных доменов

На рис. 38 представлена декомпозиция сетки на домены:

- с нулевого по пятый - домены первого вида, среди которых нет ни одной пары доменов, соединенных ребром;
- домен 6, на узлы которого замыкаются все выходящие из доменов первого вида ребра.

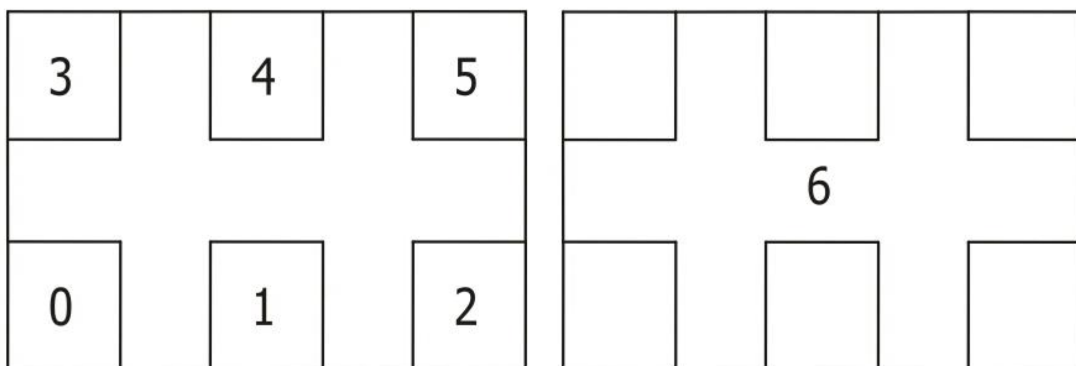


Рис. 38. Декомпозиция сетки на домены

В случае обработки домена единственным процессором теряется эффективность, которая компенсируются снижением числа конфликтов кэш-памяти **в системах с общей памятью**. В случае обработки доменов разными процессорами можно уменьшить взаимное влияние, вызванное обращением разных процессоров к данным, отображаемым в одни и те же строки в кэш-памяти. Если речь идет о системах с распределенной памятью, то данный подход позволяет использовать нетрадиционную стратегию организации работ, предполагающую конечность скорости распространения сигналов в природе. При таком подходе обособленные домены (0—5) рассматриваются как невзаимодействующие на некотором небольшом интервале времени. Таким образом, задача распадается на множество несвязанных. Каждая задача решается независимо на отдельном процессоре. Решения для доменов первого вида позволяют сформировать граничные условия для доменов

второго вида, обработка которых выполняется на втором этапе. Декомпозиция, рассмотренная в данном примере, называется двудольной.

В общем случае, r-дольная декомпозиция графа позволит выполнить расщепление задачи на r последовательно обрабатываемых групп независимых подзадач. Каждому процессору назначается не один домен, а r доменов, по одному из каждой группы, что позволяет равномерно задействовать все процессоры на каждом из r этапов.

Данная технология используется при решении задач неявными методами, предполагающими решение на каждом из шагов систем СЛАУ с помощью прямых или итерационных методов. Разбиение большой системы СЛАУ на множество систем меньшего размера способствует сокращению общего времени решения задачи.

3. Минимизация максимальной степени домена

Определим понятие макрографа декомпозиции. Макрограф состоит из p вершин, каждая из которых соответствует одному домену V_i разбиения исходного графа:

$$\widehat{G} = \left(\widehat{V},\ \widehat{E}\right),\ \widehat{V} = \{\hat{v}_i\},\ \widehat{E} = \{\hat{e}_{ij}\},\ \left|\widehat{V}\right| = p. \qquad (33)$$

Вес вершины макрографа равен суммарному весу вершин, входящих в домен i:

$$\widehat{w}_i = \sum_k w_k,\ v_k \in V_i. \qquad (34)$$

Вес ребра соединяющего вершины i, j макрографа, равен суммарному весу ребер, соединяющих вершины доменов i и j:

$$\widehat{w}_{ij} = \sum_{k,l} w_{kl},\ v_k \in V_i,\ v_l \in V_j. \qquad (35)$$

Максимальная степень вершины макрографа – число граничащих с ним доменов, служит критерием минимизации, который обеспечивает снижение

числа актов обмена данными между процессорами на каждом шаге по времени:

$$\min_i w_i \text{ при } r(v_i) < K, \qquad (36)$$

где K — максимально допустимая степень вершины макрографа, определяемая особенностями решаемой задачи. Минимизация позволяет снизить коммуникационные расходы за счет уменьшения потерь, вызванных латентностью каналов передачи данных.

При использовании адаптивных сеток отсутствие треугольников сетки, вершины которых принадлежат трем разным доменам, позволяет упростить логику параллельного вычислительного алгоритма.

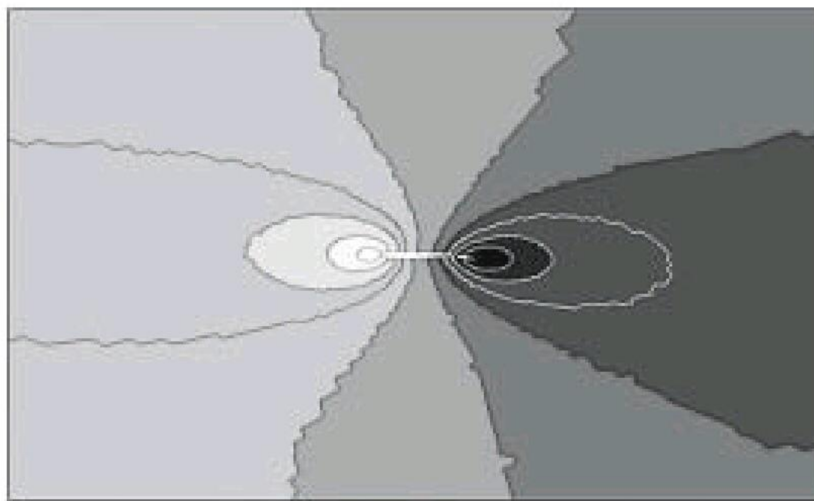


Рис. 39. Декомпозиция

Например, время расчета на 63 процессорах при использовании декомпозиции, показанной на рисунке 39 , составило 4,16 секунды, что гораздо меньше 10,56 секунд при расчете, использующем классическую декомпозицию, представленную на рис. 40. При расчетах использовалась одна и та же программа, но узлы вычислительной сетки были распределены по доменам разными способами. Выигрыш более чем в два раза получен исключительно за счет удачной декомпозиции сетки.

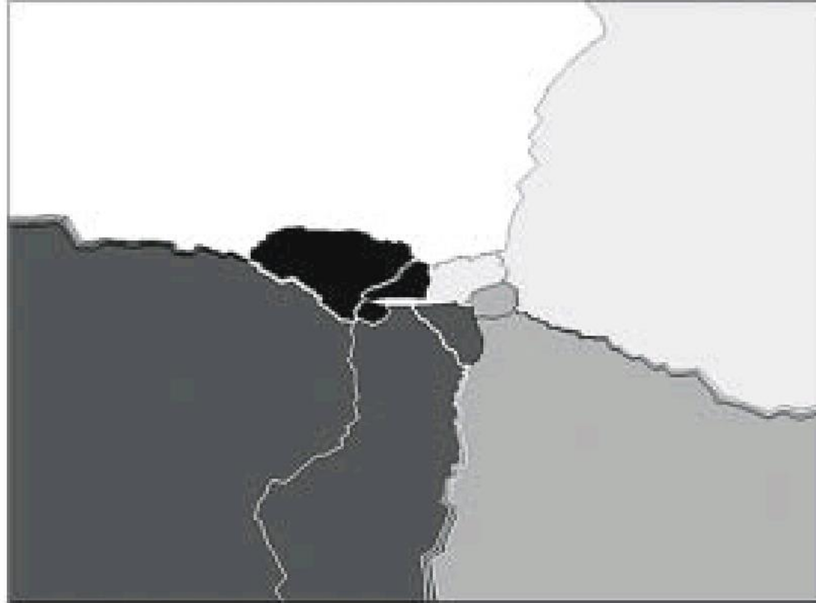


Рис. 40. Классическая декомпозиция

4. **Обеспечение связности множества внутренних узлов доменов**

На рисунке 41 показаны две части домена, состоящего из разрозненных фрагментов сетки. Формирование таких доменов является одним из существенных недостатков, присущих иерархическим методам декомпозиции графов.

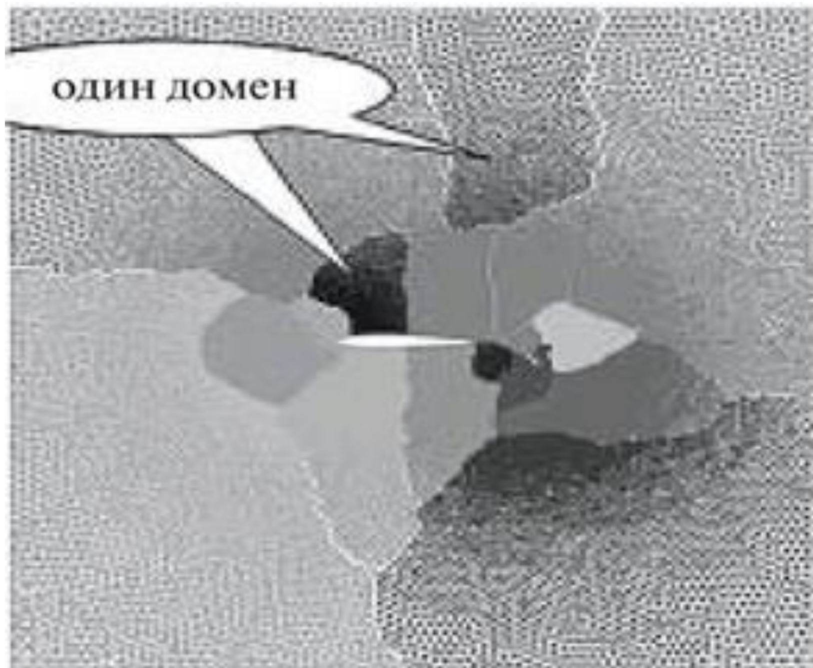


Рис. 41. Две части домена, состоящего из разрозненных фрагментов сетки

Граф такого домена – фрагментирован. Формирование подобных разбиений характерно для пакетов, подобных ParMetis (или его последовательной версии Metis). Это бесплатный пакет и библиотека функций, широко используемые на практике для решения задач декомпозиции сеток. Но при большом числе доменов и процессоров, полученные с помощью стандартных пакетов решения содержат артефакты, в том числе пустые домены и несвязные домены. Фрагментированные домены снижают эффективность проводимых на их основе расчетов. Так как вершины макрографа, соответствующие несвязным доменам имеют большую, чем остальные, степень. Следовательно, процессоры, обрабатывающие фрагментированные домены, затрачивают большее время на обмен данными в связи ростом объема передаваемых данных, и числа актов приема-передачи данных. Фрагментированность домена может приводить к снижению эффективности компрессии сеточных функций (уменьшения объема данных, описывающих исходную функцию).

Декомпозиция на основе исходной нумерации узлов

Данный метод позволяет получать декомпозицию, удовлетворяющую соответствующим критериям. Метод используется при первоначальном распределении по процессорам больших сеток для последующего решения самой задачи декомпозиции.

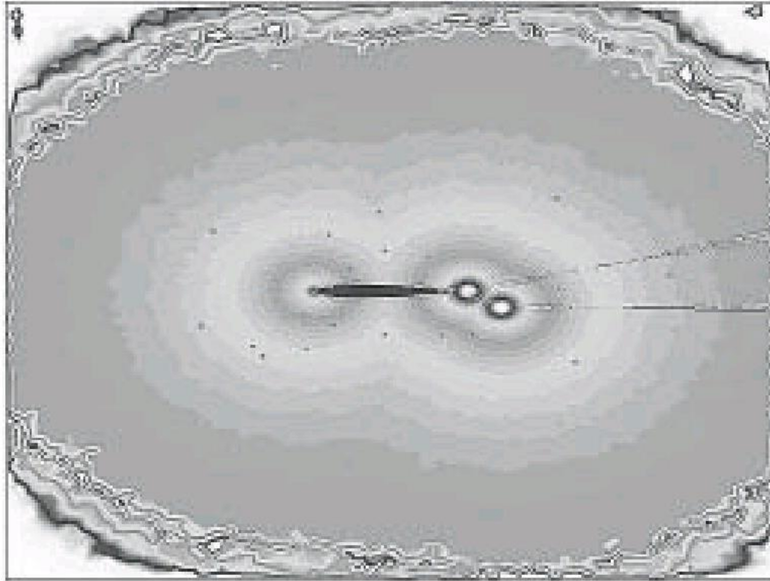


Рис. 42. Границы между доменами, проходящие по большому числу ребер

Нерегулярная сетка представляет – множество последовательно пронумерованных узлов, между которыми определены некоторые связи – ребра. Рассмотрим распределения узлов по процессорам – распределение по исходной нумерации. При разбиении сетки, содержащей n узлов на p доменов, в домен с номером k можно отнести узлы с номерами от kn/p до (k+1) n/p – 1. Это гарантирует равномерность распределения узлов, но приводит к тому, что границы между доменами проходят по большому числу ребер (рис. 42).

Отдельные точки, на рис. 42, имеющие соседние номера, будут назначены одному процессору. В результате, при расчете каждого шага по времени потребуется передача информации обо всех узлах домена между процессором обрабатывающим этот домен и практически всеми остальными процессорами. В этом случае оценка времени выполнения алгоритма:

$$T_p = k\frac{n}{p}\left(\tau_A + 4\tau_s\right), \tag{37}$$

Этот метод, полезен для предварительного распределения узлов.

Глава 13. Расширенные методы построения параллельных алгоритмов

13.1. Динамическая балансировка загрузки процессоров

Реализация декомпозиции приводит к изменению вычислительных затрат по обработке узлов и следовательно к росту дисбаланса вычислительной нагрузки на процессоры, что влечет за собой необходимость ее перераспределения.

Стратегии планирования использования процессорной мощности.

Использование алгоритмов статической балансировки загрузки процессоров допускается в случае, если веса вершин графа фиксированы и не меняются с течением модельного времени (при изменении номера шага по времени) wi(t) = const.

Использование диффузионных методов балансировки загрузки или полное перераспределение вершин с помощью алгоритмов декомпозиции сеток, в случае медленного изменения весов вершин от шага к шагу по времени:

Использование методы, в которых отсутствует априорное планирование распределения вычислений по процессорам, например - серверный параллелизм. Методы применяются в случае, если веса вершин значительно изменяются от шага к шагу по времени и не могут быть определены заранее.

В главе 9 подробно рассматривается метод диффузной балансировки загрузки, являющийся методом динамической балансировки загрузки процессоров.

13.2. Визуализация сеточных данных

Визуализация – наглядное отображение данных. Увеличение количества процессоров, используемых при параллельных вычислениях, необходимость преобразования большого числа сеточных данных для

вычислительных процессов к удобному для анализа виду, приводит к потребности в визуализации.

Для анализа больших объемов данных необходимо привлекать специальные, ориентированные на удаленную визуализацию решения. Это обусловлено следующим причинами:

ограниченная пропускная способность каналов связи между суперкомпьютером и рабочим местом пользователя не обеспечивает возможность быстрой передачи больших объемов данных от суперкомпьютерного центра на рабочее место пользователя;

ограниченный объем оперативной памяти компьютера пользователя затрудняет обработку больших объемов данных, значительно снижая эффективную производительность компьютера рабочего места;

ограниченность размера файловой системы компьютера пользователя не позволяет в полном объеме хранить результаты вычислительных экспериментов и не обеспечивает приемлемую скорость доступа к хранимым данным.

Существенная часть операций, обеспечивающих визуализацию больших объемов данных должна выполняться на параллельных вычислительных мощностях многопроцессорного сервера визуализации.

Клиент-серверная технология

Данная технология использует технологию клиент-сервер (рис. 43) в соответствии с которой:

- сервер визуализации, обеспечивает обработку большого объема данных и реализуется на многопроцессорной системе

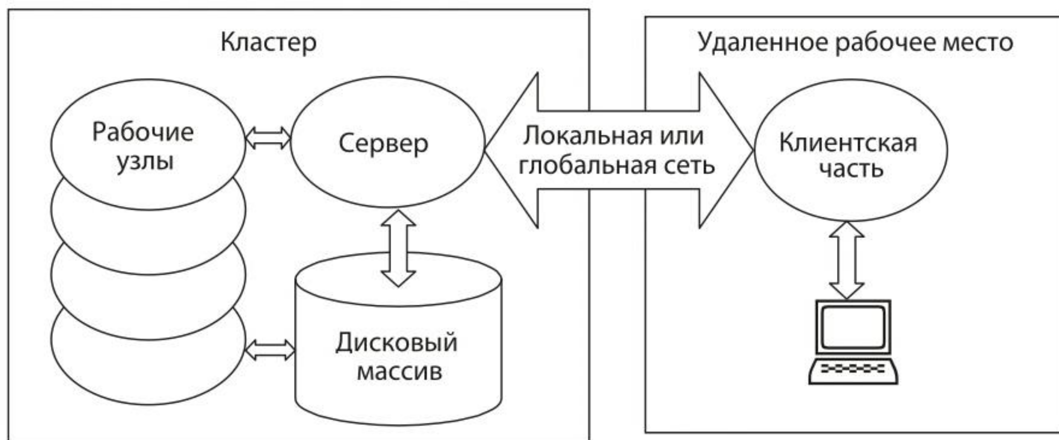


Рис. 43. Технология клиент-сервер

- клиент визуализации реализуется на вычислительной системе рабочего места пользователя. Клиент обеспечивает интерфейс взаимодействия с пользователем и выполняет непосредственное отображение данных, предварительно подготовленных сервером.

Визуализация результатов вычислительных экспериментов подразумевает взаимодействие двух компонент:

- модуля численного моделирования;
- модуля визуализации.

Существуют следующие подходы к организации такого взаимодействия:

- **online-визуализация** – визуализация данных непосредственно при проведении расчета, использующая данные, расположенные в оперативной памяти.
- **offline-визуализация** – визуализация данных, периодически записываемых на диск с помощью отдельной программы.

Рассмотри каждый из этих подходов:

Online-визуализация

Преимущества:

1. Экономия дискового пространства и экономия времени на чтение и запись результатов.

2. Потенциальная возможность активного влияния на ход выполнения вычислений.

Проблемы:

Предъявляются большие требования к оперативной памяти: кроме вычислительного модуля ее потребляет модуль визуализации.

Затруднены возможности повторного изучения единожды визуализированных данных.

Ограничены возможности визуализации динамики процесса

Для запоминания промежуточных результатов потребуется дополнительная оперативная память, объем которой ограничен.

Ограничены возможности визуализации заданных моментов времени, поскольку темп изучения пользователем визуальных образов не согласован с темпом проведения расчета.

Неконтролируемо ухудшается балансировка загрузки процессоров.

Задачу балансировки загрузки для вычислительного ядра можно решить, но основная цель интерактивной системы визуализации — показать объект за заданное короткое время. В этом контексте проблема эффективного использования многопроцессорной системы отходит на второй план. В одну задачу объединяются не просто две разные задачи, но две задачи, требующие разработки алгоритмов разных классов, к которым предъявляются взаимно

противоположные требования. Оптимизация согласованного использования системы этими двумя ядрами (вычислительным и визуализации) – более сложная задача, чем раздельная оптимизация под каждую из этих задач.

Возникают организационные трудности, обусловленные режимом коллективного доступа к мощностям суперкомпьютерных центров.

Offline-визуализация

Второй подход предполагает использование пакетов визуализации: как ParaView, Visit, EnSight, OpenDX, RemoteViewer. В их рамках осуществляется раздельная реализация модулей численного моделирования и визуализации — выделение их в отдельные программы, взаимодействующие через файловую систему.

Недостаток: необходимость периодической записи данных на диск промежуточных результатов, что требует повышенных ресурсов дисковой памяти и приводит к дополнительным потерям времени.

Этапы детализации:

Моделирование	Сервер визуализации	Клиент визуализации
а) декомпозиция сетки **б)** запись структуры сетки **в)** запись фрагментов сетки **г)** запись фрагментов сеточной функции	**д)** чтение структуры сетки **е)** декомпозиция сетки, назначение фрагментов сетки на процессоры **ж)** чтение фрагментов сетки **з)** чтение фрагментов сеточной функции **и)** формирование данных, описывающих виртуальную сцену и их передача на клиент визуализации	**к)** прием данных от сервера визуализации и преобразование их к виду, пригодному для отображения **л)** отображение **м)** манипулирование образом объекта

Рассмотрим наиболее затратные с точки зрения времени выполнения этапы.

Декомпозиция сетки - пункты а и е. Декомпозиция необходима для эффективной организации параллельной обработки данных на нескольких процессорах сервера визуализации.

Ввод-вывод фрагментов сетки и сеточных функций (пункты в, г, ж, з). При визуальном изучении результатов может потребоваться детальная визуализация любого фрагмента сетки, что вызывает необходимость хранить сетку без каких-либо потерь топологической информации. Объем информации о топологии сетки является доминирующим относительно объема геометрической информации. В некоторых частных случаях, таких как регулярные сетки или двумерные планарные сетки, объем топологической информации может быть значительно сокращен без потери

точности описания сетки. Однако, в общем случае, для неструктурированных сеток такие алгоритмы неизвестны, что делает этапы записи и чтения сетки затратными и с точки зрения дисковой памяти, и с точки зрения времени выполнения. В отличие от сетки, сеточные функции, могут быть записаны с частичной потерей точности.

Формирование данных, описывающих виртуальную сцену и их передача на клиент визуализации (пункт и).

Здесь решаются следующие задачи:

1) требуется сократить объем данных, передаваемых между серверной и клиентской частями системы визуализации , причем сократить этот объём до заданного уровня, определяемого отношением времени реакции системы визуализации и пропускной способности канала связи

между сервером и клиентом визуализации;

2) требуется выполнить пункт 1) за время, не превышающее заданного времени реакции системы визуализации;

3) требуется обеспечить достаточную точность представления исходного объекта содержимым сформированной виртуальной сцены.

Требуется совместное решение всех трех задач. Основным методом, позволяющим решить за заданное время все три задачи, является огрубление изучаемого объекта с контролируемой точностью

Основная задача при этом является такое сокращение числа описывающих объект примитивов (узлов, ребер, граней и т. п.), при котором формируемый на их основе образ визуально близок исходному неогрубленному объекту. Требуется сократить общий объем данных, описывающих огрубленный объект за заданное время, до заданной величины.

Визуализация изоповерхностей - наглядный метод визуализации трехмерных скалярных данных.

Изоповерхность – это множество точек пространства, в которых скалярная функция принимает заданное значение. Изоповерхность можно описать как набор треугольников, опирающихся вершинами на множество точек трехмерного пространства. В этом случае наглядность визуализации обеспечивается видеоускорителями, которые позволяют отображать массивов треугольников за счет быстрого вывода на экран визуализации и возможности формирования стереоизображений автоматически. Ядро системы визуализации образуют алгоритмы фильтрации и изоповерхности, так как число описывающих изоповерхность треугольников может превышать число треугольников, отображаемых графическим ускорителем. Алгоритмы обеспечивают аппроксимацию исходной триангуляции новой

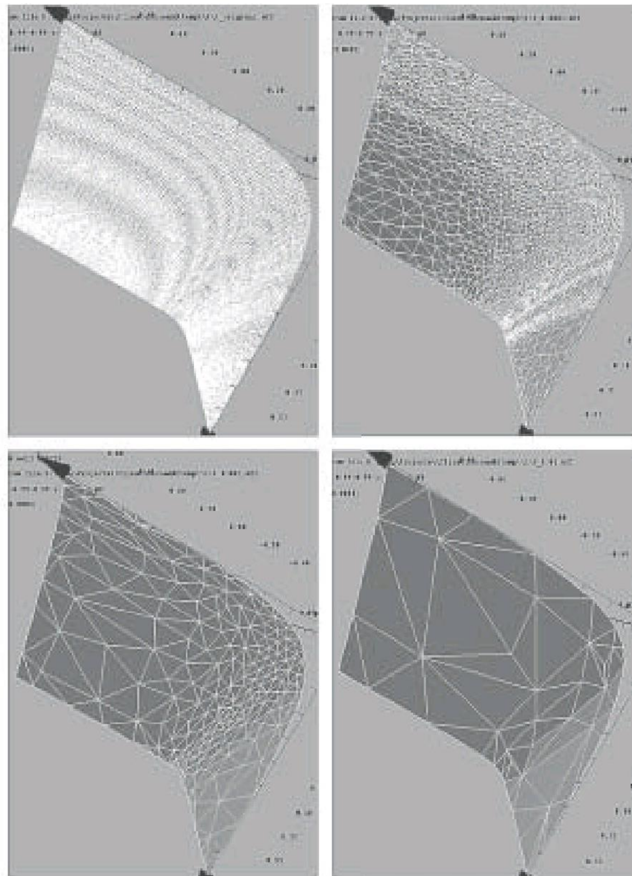


Рис. 44. Пример огрубления поверхности

триангуляцией, описываемой ограниченным объемом данных, что позволяет восстановить образ с высоким уровнем качества (рис. 44).

Алгоритм огрубления первичных данных должен обладать быстротой выполнения и способностью проводить огрубление данных до заданного объема за время передачи данных на компьютер пользователя.

Необходимость этапа огрубления сетки до размеров, при которых возможна передача данных за короткое время через медленные каналы связи, связана с совпадением числа описывающих изоповерхность узлов по порядку величины с числом узлов исходной трехмерной сетки. Требуемые коэффициенты «сжатия» — отношения объемов данных, описывающих исходную изоповерхность и ее образ, — достигают сотен тысяч и более, поэтому следует использовать методы сжатия с потерей точности. Стандартные методы сжатия без потерь не могут сжать информацию, описывающую топологию изоповерхности. Для этого используются алгоритмы, которые строят триангулированную поверхность, аппроксимирующую исходную изоповерхность. Триангулированная поверхность содержит меньшее количество узлов. Задача огрубления произвольной триангулированной поверхности имеет отношение не только к визуализации объектов, описываемых неструктурированными сетками, но и к визуализации объектов, изначально заданных на регулярных решетках, топологически эквивалентных индексным параллелепипедам. На рис. 45 представлен пример сечения куба изоповерхностью.

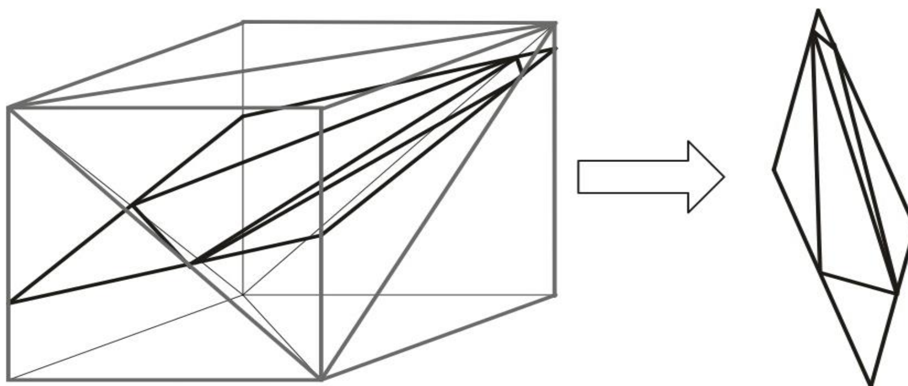


Рис. 45. Пример сечения куба изоповерхностью

Сечение является четырехугольником. В связи с применением триангуляции для описания изоповерхности следует разделить этот четырехугольник на треугольники. На рис. 46 приведены два способа разбиения на треугольники.

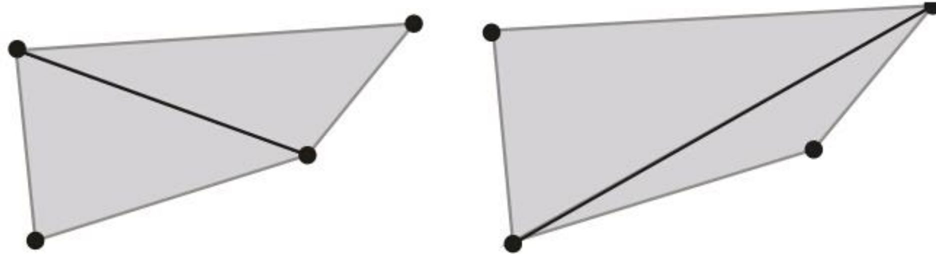


Рис. 46. Два способа разбиения на треугольники

Для исключения этой ситуации куб предварительно разбивается на пирамиды и уже по точкам пересечения с ребрами пирамид проводят сечение.

Примеры визуализации

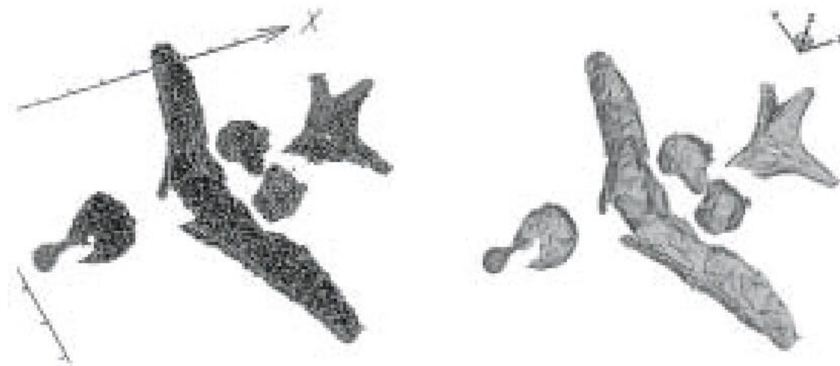


Рис. 47. Результаты визуализации изоповерхностей плотности с помощью методов с потерей точности

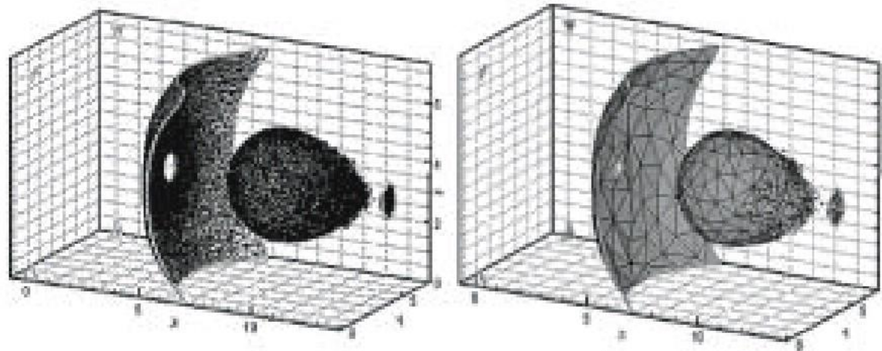


Рис. 48. Результаты визуализации изоповерхностей плотности
с помощью системы Tecplot

На рисунках 47, 48 представлены результаты визуализации изоповерхностей плотности с помощью методов с потерей точности и с помощью системы Tecplot, выполняющей визуализацию без огрубления.

Визуализация обеспечивает возможность анализа произвольно большого объема данных – позволяет оценить вид изучаемого объекта в целом и его произвольные фрагменты независимо от того, насколько велик общий объём описывающих объект данных.

13.3. Сети сортировки

Подход к сортировке, основанный на понятии «сеть сортировки». Данный подход позволяет создавать параллельные алгоритмы сортировки больших объемов данных.

Сеть сортировки – это вид алгоритмов сортировки, в которых порядок выполнения операций сравнения и их количество не зависит от значения элементов сортируемого массива. Каждый элемент массива последовательно обрабатывается компараторами сравнения/перестановки. Изображение сортируемых элементов массива соответствует горизонтальными линиями данных. Изображение компараторов соответствует вертикальными отрезками. Каждый компаратор соединяет две линии данных; у каждого компаратора есть два входа и два выхода – верхние и нижние. Компаратор принимает на вход два элемента массива. В компараторе элементы массива, расположенные на двух линиях данных, сравниваются между собой и при

необходимости переставляются местами таким образом, чтобы на верхнем выходе всегда был меньший из двух элементов, а на нижнем – больший.

На рисунке 49 показана сеть сортировки трех элементов и пример упорядочивания массива {9, 4, 2}. Для сортировки используется метод вставки. Каждому элементу массива соответствует линия данных, значение на которой меняется по мере срабатывания компараторов.

Первый компаратор сортирует массив из двух элементов, а оставшиеся два компаратора обеспечивают вставку третьего элемента в нужное место. Сеть сортировки характеризуется временем работы – числом шагов, требуемых для выполнения сортировки. Фактически, сеть сортировки задает расписание, согласно которому некоторые компараторы могут срабатывать одновременно.

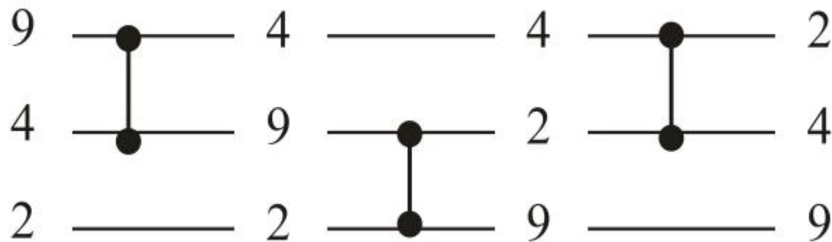


Рис. 49. Сеть сортировки трех элементов и пример упорядочивания массива {9, 4, 2}

В сети, показанной на рис. 50, на шаге 5 одновременно могут выполняться компараторы (1, 2), (3, 4) и (5, 6). Благодаря такой параллельной обработке сортировка будет выполнена за 9 шагов.

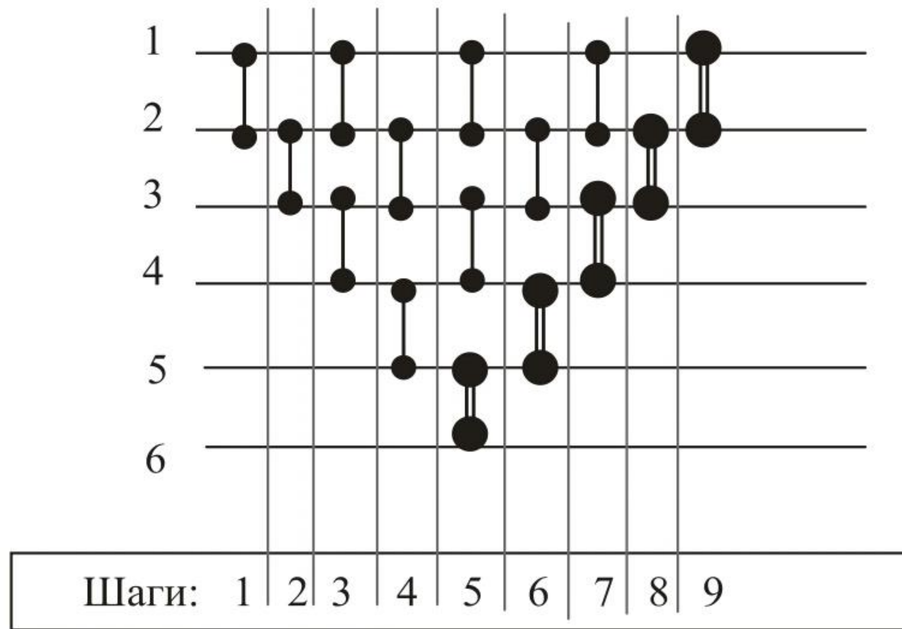


Рис. 50. Одновременное выполнение на шаге 5 компараторов (1, 2), (3, 4) и (5, 6)

В сетях сортировки актуальна задача минимизации времени работы. Для значений n <= 10 известны сети, обладающие минимально возможным временем. Для шести процессоров известна сеть сортировки, выполняющаяся всего за 5 шагов (рис. 51).

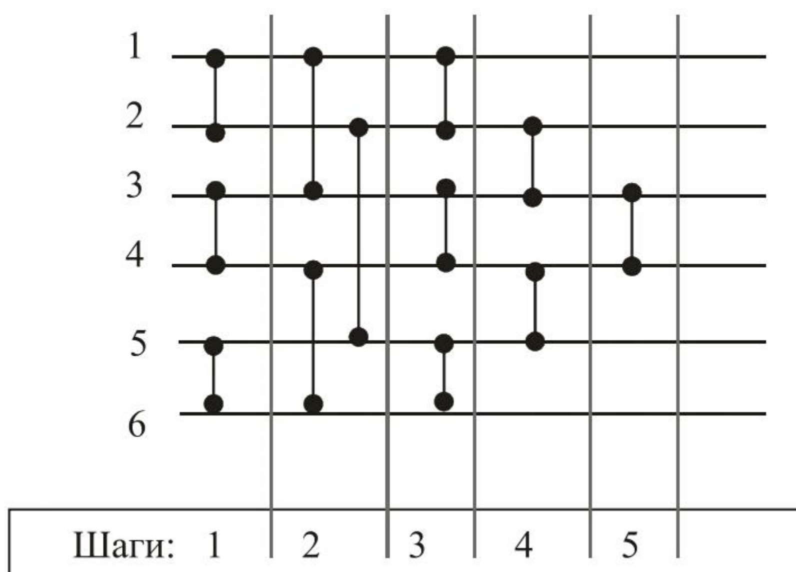


Рис. 51. Сеть сортировки для шести процессоров

Глава 14. Технологии параллельного программирования: использование традиционных последовательных языков

14.1. Принципы анализа параллельных алгоритмов

При проведении анализа параллельных алгоритмов рассматриваются две характеристики – коэффициент ускорения и стоимость. **Коэффициент ускорения** характеризует скорость работы параллельного алгоритма относительно наилучшего последовательного алгоритма. Для последовательного алгоритма сортировки необходимо O(N log N) операций. Для параллельного алгоритма сортировки (сложность O(N)) составит O(log N). **Стоимость** вычисляется как произведение сложности алгоритма и количества необходимых для вычисления процессоров. Так для параллельного алгоритма сортировки O(N), при количестве процессоров равном числу входных записей, стоимость составит $O(N^2)$. То есть по сравнению с алгоритмом последовательной сортировки, стоимость которого составляет O(N log N), параллельный алгоритм обладает большей стоимостью. Для того чтобы параллельный алгоритм сортировки имел смысл, число используемых процессоров не должно меняется при росте длины входных данных, и быть в значительной степени меньше предполагаемого объема входных данных.

14.2. Использование традиционных последовательных языков (системы программирования OpenMP, DVM, mpC)

Использования в традиционных языках программирования специальных комментариев, добавляющих "параллельную" специфику в изначально последовательные программы. Это спецкомментарии для компилятора. Использование спецкомментариев добавляет возможность параллельного исполнения и полностью сохраняет исходный вариант

программы. Это – один из подходов к разработке эффективного параллельного программного обеспечения. В рамках подхода рассмотрим три системы программирования OpenMP, DVM, mpC

14.2.1. OpenMP

В рассматриваемом стандарте соответствующая алгоритму решения последовательная программа служит основой для применения параллельных вычислений. Для этого используются директивы, процедуры и переменные окружения. Стандарт OpenMP разработан для языков Fortran (77, 90, и 95), C и C++. Технология OpenMP опирается на понятие общей памяти, и поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов. Термин «нить» следует понимать как легковесный процесс, поток. На рис. 52 представлен процесс исполнения программы. Текст программы делится на последовательные и параллельные секции. Затем инициируется нить-мастер, которая обеспечивает выполнение программы.

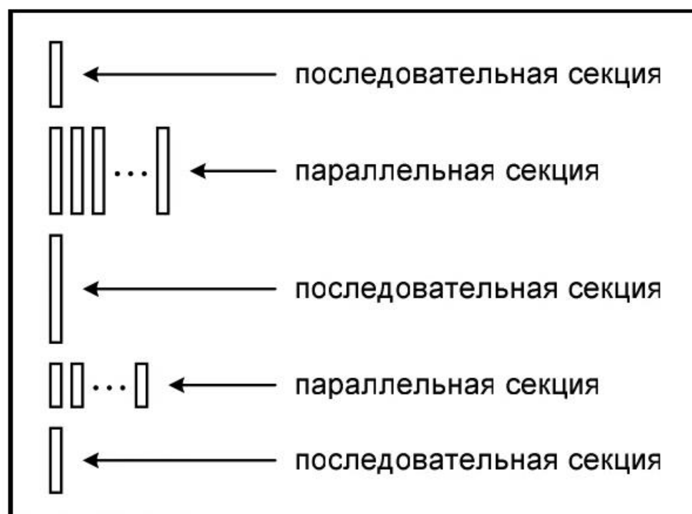


Рис. 52. Разбиение текста программы на последовательные и параллельные области

Все последовательные секции выполняет нить-мастер. Параллелизм обеспечивается схемой FORK/JOIN. Нить-мастер порождает дополнительные нити при входе в параллельную область (выполняется операция FORK). При этом каждая порожденная нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все порожденные нити исполняют одинаковый код, который соответствует параллельной области. Основная нить дожидается завершения остальных нитей при выходе из параллельной области и она же обеспечивает последующее выполнение программы (выполняется операция JOIN).

В параллельной секции происходит разделение переменных программы на общий (SHARED) и локальный (PRIVATE) классы. Глобальные переменные могут использоваться всеми нитями. Для локальных переменных создается экземпляр на каждой нити.

Таким образом, за выполнение параллельных секций отвечает набор нитей, у которых есть доступ к глобальным и локальным переменным.

Полные тексты спецификаций OpenMP для языков Fortran, C и C++ на сайте http://www.openmp.org.

Преимущества технологии OpenMP:

1. Технология предоставляет пользователю возможность работать как с параллельной так и с последовательной секциями программы. Директивы OpenMP не обрабатываются компилятором последовательного вычислительного устройства. Переменные окружения и специальные функции защищены механизмами, предусмотренными стандартом.
2. Технология предоставляет возможность постепенного перехода к параллельной версии.

14.2.2. Система программирования DVM

Данная модель объединяет элементы моделей параллелизма по данным и управлению. На этой модели основаны языки параллельного программирования Fortran-DVM и C-DVM. DVM-система разработки параллельных программ разработана в Институте прикладной математики им. М. В. Келдыша РАН.

Основные компоненты DVM-системы:

- компиляторы с языков F-DVM и C-DVM;
- система поддержки выполнения параллельных программ;
- отладчик параллельных программ;
- анализатор производительности;
- предсказатель производительности.

Принципы DVM-системы:

Модель выполнения параллельной программы должна иметь высокий уровень реализации. На этом требовании основана рассматриваемая система.

Необходимо, чтобы спецификации системы были прозрачными для рядовых компиляторов. Программа на языках Fortran-DVM и C-DVM помимо описания алгоритма средствами традиционных языков Fortran 77 или C, содержит спецификации параллелизма – правила параллельного выполнения этого алгоритма. Эти спецификации, которые по-другому называют директивами, должны быть "невидимы" для стандартных компиляторов.

Языки параллельного программирования должны представлять собой традиционные языки последовательного программирования, расширенные спецификациями параллелизма. Эти языки должны предлагать программисту модель программирования, близкую к модели выполнения. Знание программистом модели выполнения его программы и ее близость к модели программирования существенно упрощает для него анализ

производительности программы и проведение ее модификаций, направленных на достижение приемлемой эффективности.

Динамическая реализация модели выполнения параллельной программы осуществляется системой поддержки выполнения DVM-программ. Это позволяет обеспечить динамическую настройку DVM-программ при их запуске на параметры приложения и конфигурацию параллельного компьютера.

DVM-программ исполняется на виртуальной многопроцессорной системе, которую предоставляет базовое системное программное обеспечение и аппаратное обеспечение программе пользователя. Для распределенной вычислительной системы примером является MPI-машина. Виртуальная многопроцессорная система всегда представляется в виде многомерной решетки процессоров. Число процессоров виртуальной многопроцессорной системы и конкретный способ ее представления задаются при запуске DVM-программы.

В момент запуска DVM- программа выполняется одновременно на всех процессорах виртуальной многопроцессорной системы. В это время в DVM-программе существует единственный поток управления (единственная ветвь).

В DVM-программе используются два уровня параллелизма. На верхнем уровне в программе описывается какое-то число независимых ветвей (задач), которые могут выполняться параллельно. Задачи DVM - это независимые по данным крупные блоки программы. В конце ветвей допускается выполнение глобальной редукционной операции. В рамках каждой ветви могут дополнительно выделяться параллельные циклы. Никакой другой иерархии параллелизма DVM не допускает, и описать в теле параллельного цикла еще несколько независимых ветвей нельзя.

Поток управления разбивается на несколько независимых потоков. Это происходит при входе в параллельную конструкцию. Потоки определяют процесс вычислений на процессорах. При выходе из параллельной

конструкции потоки управления на всех процессорах вновь становятся одинаковыми.

Все переменные DVM-программы размножаются по всем процессорам. В процессе вычислений для каждого процессора создается локальная копия переменной. При работе с распределенными массивами их расположение определяется специальной директивой.

Любой оператор присваивания DVM-программы выполняется в соответствии с правилом собственных вычислений, то есть оператор обрабатывается процессором, на котором происходит обработка переменной из левой части оператора.

Любая DVM -программа работает в соответствии с моделью SPMD на всех выделенных ей процессорах.

Отладка и оптимизация DVM-программ

Для отладки DVM-программ авторы системы разработали как набор инструментальных средств, так и специальную технологию отладки. Функциональная отладка DVM-программ осуществляется поэтапно:

- программа отлаживается на рабочей станции;
- программа пропускается в специальном режиме для проверки DVM-указаний;
- программа тестируется на параллельной машине в режиме сравнения промежуточных результатов ее параллельного выполнения с лучшими результатами, которые были полученными при последовательном выполнении.

Для отладки программы на реальной параллельной машине используются средства трассировки, позволяющие зафиксировать последовательность обращений к переменным и их значения, а также последовательность вызовов функций системы поддержки выполнения DVM-программ.

Повысить эффективность DVM-программ помогает анализатор производительности, который позволяет пользователю получить информацию об основных показателях эффективности выполнения его программы или ее частей на параллельной системе. Для повышения эффективности программ можно использовать специальный инструмент — предсказатель производительности. Он позволяет на рабочей станции смоделировать выполнение DVM-программы на параллельной ЭВМ с заданными заранее параметрами (топология коммуникационной сети, ее латентность и пропускная способность, а также производительность процессоров) и спрогнозировать для такой системы характеристики выполнения программы.

Исходные тексты системы, библиотеки выполняемых программ, документация, примеры и другие материалы доступны через Интернет по адресу: http://www.keldysh.ru/dvm

14.2.3. Система программирования mpC

Язык mpC – первый язык высокого уровня, разработанный специально для программирования неоднородных сетей. Он имеет все средства для определения основных свойств параллельного алгоритма, влияющие на скорость его выполнения в неоднородной вычислительной среде: необходимое число параллельных процессов; объем вычислений и передаваемых данных в рамках каждого процесса; сценарий взаимодействия процессов и т.д. mpC позволяет менять эти характеристики динамически во время выполнения программы. Информация, извлеченная из описания параллельного алгоритма, вместе с данными о реальной производительности процессоров и коммуникационных каналов, помогает системе программирования mpC найти эффективный способ отображения процессов mpC-программы на компьютеры сети.

Язык и система программирования mpC разработана в Институте системного программирования РАН. Детали описания и текущее состояние проекта можно найти на сайте http://www.ispras.ru/~mpc.

Основы программирования на mpC

mpC-программа – множество параллельных процессов, взаимодействующих посредством неявной передачи сообщений. Количество процессов, составляющих программу и характеристики компьютеров, на которых эти процессы выполняются, указываются пользователем программы в момент ее запуска. Реализацию этого механизма обеспечивают внешние средства. Код на mpC, составленный в соответствии с моделью SPMD, управляет тем, какие именно вычисления выполняются каждым из процессов программы.

Группа процессов, совместно выполняющих некоторый параллельный алгоритм, в языке mpC называется сетью. Сеть в mpC – механизм, позволяющий программисту абстрагироваться от реальных физических процессов параллельной программы. В простом случае сеть - множество виртуальных процессоров.

Определение сети в программе вызывает создание группы реальных процессов, представляющей эту сеть: каждому виртуальному процессору соответствует отдельный процесс параллельной программы. В разные моменты времени выполнения параллельной программы один и тот же реальный параллельный процесс может представлять различные виртуальные процессоры различных сетей.

Вопросы и задания

1. Какие конструкции языка C препятствуют автоматическому распараллеливанию программ?

2. Отвечая на предыдущий вопрос, вы сформировали список конструкций, препятствующих автоматическому распараллеливанию программ. Внимательно проанализируйте полученный список и подумайте,

верно ли такое утверждение: "Любую программу, в которой нет конструкций из списка, можно автоматически распараллелить"?

3. В цикле содержится вызов функции. Какие конструкции языков необходимо учитывать, чтобы определить, являются ли все итерации данного цикла независимыми или нет? Рассмотрите языки C и Fortran.

4. Какой вычислительной системе лучше соответствует технология OpenMP: вычислительному кластеру из рабочих станций или SMP-компьютеру?

5. Чем отличаются понятия "процесс" и "нить"?

6. Верно ли утверждение: "OpenMP-программы работают согласно модели SingleProgram Multiple Data (SPMD)?"

7. Чем различаются общие и локальные переменные в последовательной секции OpenMP-программы?

8. Допускает ли OpenMP изменение числа параллельных нитей по ходу работы программы?

9. Как будет обрабатываться такая конструкция: в параллельном цикле стоит вызов функции, в теле которой так же есть параллельный цикл?

10. Почему в OpenMP нет конструкций, аналогичных директивам DVM ALIGN и

DISTRIBUTE?

11. Предположим, что в OpenMP не было бы директивы BARRIER. Смоделируйте барьерную синхронизацию нитей с помощью других средств OpenMP, в частности, с помощью механизма критических секций.

12. Можно ли автоматически конвертировать DVM-программу в программу на OpenMP?

13. Напишите с помощью OpenMP, DVM и mpC программы перемножения двух матриц, решения системы линейных уравнений методом Гаусса, нахождения обратной матрицы. Проанализируйте выполненную работу с точки зрения эффективности полученных программ, простоты написания программ, переносимости программ.

14. Проанализируйте, что общего и чем различаются классы переменных вOpenMP, DVM и трС.

15. Для чего в DVM введена директива ALIGN?

16. Сравните модели выполнения параллельных циклов в OpenMP и DVM.

17. Для нахождения суммы элементов вектора в рамках OpenMP при сложении частичных сумм можно воспользоваться критической секцией. Как выполнить эту же операцию в DVM?

Глава 15. Технологии параллельного программирования: системы программирования на основе передачи сообщений и другие системы

15.1. Системы программирования на основе передачи сообщений (системы параллельного программирования Linda, Message Passing Interface (MPI), MPI-2, DVM

Появление систем программирования на основе передачи сообщений вызвано распространением компьютеров с распределенной памятью. В таких системах отсутствует единое адресное пространство. Обмен данными между параллельными процессами обеспечивается за счет механизма явной передачи сообщений через коммуникационную среду. Отдельные процессы описываются с помощью традиционных языков программирования, а для организации их взаимодействия вводятся дополнительные функции. В этой связи системы программирования, основанные на явной передаче сообщений, реализуются в виде интерфейсов и библиотек.

Известны следующие системы программирования на основе передачи сообщений - Shmem, Linda, PVM, MPI и др.

Система Linda

Система создана в Йельском университете США в середине 80-х годов 20 века. В пределах системы параллельная программа – множество

параллельных процессов. Каждый процесс работает как последовательная программа. Все процессы имеют доступ к общей памяти – пространству кортежей, единицей хранения в которой является кортеж. Каждый кортеж – взятая в скобки упорядоченная последовательность значений, разделенных запятыми. Например,

("Hello", 42, 3.14), (5, FALSE, 97, 1024, 2), ("worker", 5)

Так, первый кортеж этого примера состоит из строки "Hello", элемента целого типа 42 и вещественного числа 3.14. Во втором кортеже есть элемент целого типа 5, элемент логического типа false и три целых числа. Последний кортеж состоит из двух элементов: строки "worker" и целого числа 5. Количество элементов в кортеже, вообще говоря, может быть любым, однако конкретные реализации могут накладывать ограничения. Если первым элементом кортежа является строка, то эта строка называется именем кортежа.

Все процессы работают с пространством кортежей по принципу: поместить кортеж, забрать, скопировать.

В отличие от традиционной памяти:

1.Процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам.

2. Если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной – в пространстве кортежей окажется два кортежа с одним и тем же именем.

3. Изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов, и вновь поместить измененный кортеж в память. В отличие от других систем программирования, процессы в системе Linda никогда не взаимодействуют друг с другом явно, и их общение всегда идет через пространство кортежей.

Linda позволяет добавить в любой последовательный язык четыре новые функции, в результате этот язык становится средством параллельного программирования. Эти функции и составляют систему Linda: три для операций над кортежами и пространством кортежей и одна функция для порождения параллельных процессов.

Преимущество системы – простота концепции.

Проблема – вопрос эффективной поддержки пространства кортежей. Если вычислительная система обладает распределенной памятью, то общение процессов через пространство кортежей заведомо будет сопровождаться большими накладными расходами. Все предлагаемые решения либо усложняют систему, либо являются эффективными только для узкого класса программ.

Удачным примером можно считать реализацию популярного пакета для квантово-химических расчетов Gaussian.

Современное состояние и дополнительную информацию по системе Linda можно найти на сайте http://www.cs.yale.edu/Liiida/liiida.html.

Система MPI(Message Passing Interface)

MPI – наиболее распространенная технология программирования параллельных компьютеров с реализацией модели распределенной памяти. Основным способом взаимодействия параллельных процессов является передача сообщений. Интерфейс стандарта MPI соблюдается системой программирования MPI на каждой вычислительной системе и пользователем при создании своих программ.

Система обеспечивает создание параллельных программ в стиле MIMD, а именно - обеспечивает объединение процессов с различными исходными текстами. На практике программистами используется SPMD-модель, которая использует один и тот же код для всех параллельных процессов. На сегодняшний день большинство версий MPI обеспечивают работу с нитями.

Под MPI-программой понимают множество параллельных взаимодействующих процессов, каждый из которых порождается один раз. Совокупность процессов образуют параллельную часть программы. Не допускается порождение дополнительных процессов или уничтожение существующих в ходе выполнения MPI-программы.

Каждый процесс работает в своем адресном пространстве. Организация взаимодействия между процессами осуществляется при помощи обмена сообщениями. В MPI отсутствуют общие переменные или данные. Для локализации взаимодействия параллельных процессов программы создаются группы процессов, с отдельной средой для общения – коммуникатором. При старте программы все порожденные процессы работают в рамках коммуникатора, имеющего предопределенное имя mpi_comm_world. Этот коммуникатор служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет два уникальных атрибута - коммуникатор и номер в коммуникаторе.

Способом общения процессов является посылка сообщений (набора данных некоторого типа). Основные атрибуты сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения.

Имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс mpi_. Интерфейс MPI описан в файле mpi.h, который подключается директивой #inciude <mpi. h>, стоящей в начале программы.

MPI поддерживает работу с языками C, Fortran, C++.

О технологии MPI можно узнать на сайте http://www.mpiforum.org.

Задания

1. Что общего и в чем различия между традиционной общей памятью в SMP-компьютерах и пространством кортежей в системе Linda?

2. В программе необходимо использовать критическую секцию. Как реализовать критическую секцию в рамках системы Linda?
3. Продумайте детали реализации системы Linda в Linux-кластере. Укажите возможные узкие места, влияющие на эффективность выполнения программ.
4. Необходимо написать программу для компьютера с общей памятью. Чему отдать предпочтение: OpenMP или Linda? Сравните данные технологии с различных точек зрения.
5. С использованием системы Linda напишите программу, реализующую сложение элементов вектора по схеме сдваивания.
6. Можно ли написать автоматический конвертор, преобразующий программы из OpenMP в систему Linda и наоборот?
7. Можно ли написать автоматический конвертор, преобразующий программы из MPI в систему Linda и наоборот?
8. В распоряжении программистов есть, с одной стороны, MPI и OpenMP, а с другой стороны, компьютеры с общей и распределенной памятью. Какая технология программирования какой архитектуре лучше соответствует?
9. Чем отличаются процессы в MPI и в системе Linda?
10. Что означает в MPI посылка и прием сообщения с блокировкой?
11. Всем MPI-процессам надо обменяться сообщениями "по кольцу". Найдите такой способ организации программы, который бы гарантировал отсутствие тупиковой ситуации. Как сделать то же самое, но с использованием только блокирующих операций MPl_Send и MPl_Recv?
12. Верно ли, что в коллективных операциях участвуют все процессы приложения?
13. Могут ли каким-то образом общаться процессы, принадлежащие разным коммуникаторам?

14. Верно ли, что никакие два процесса программы не могут иметь одинаковые номера?
15. Сравните технологии программирования MPI и DVM с разных точек зрения.
16. Какая взаимосвязь между понятиями виртуальной топологии процессов в MPI и сетью в трС? Каково назначение этих конструкций?
17. Можно ли в MPI-программе учесть возможную неоднородность вычислительной системы?
18. Почему в MPI нет механизма критических секций?
19. Почему использование асинхронных операций обмена сообщениями, выполняющихся на фоне вычислений, вместо операций с блокировкой необязательно приведет к ускорению программы?
20. Можно ли в MPI принять сообщение, не зная точно номера посылающего процесса?
21. Известно, что MPI не гарантирует справедливости в распределении приходящих сообщений. Как в схеме мастер/рабочие гарантировать регулярную загрузку и регулярное обслуживание каждого подчиненного процесса?
22. Как в MPI определить размер буфера, необходимого для приема сообщения?
23. Какие особенности программно-аппаратной среды компьютера могут быть использованы системой поддержки MPI для эффективного выполнения функции MPI_Startall?
24. Какое выражение нужно поместить на место параметра color функции MPl_Comm_split, чтобы группу из 100 процессов разделить на две подгруппы с номерами от 0 до 40 и от 41 до 99?
25. Реализуйте с использованием MPI метод Якоби по схеме DVM-программы, приведенной в предыдущем параграфе. Сравните

полученную MPI-программу с аналогичной DVM-программой. Какие достоинства есть у каждой из этих программ?

26. Объясните, в каких случаях вы отдали бы предпочтение именно системе Linda,MPI, OpenMP, DVM, mpC?

15.2. Другие языки и системы программирования (Т-система, НОРМА)

Т-система

Т-система – технология автоматического динамического распараллеливания программ. Т-системы была разработана в конце 80-х годов 20 века в Институте программных систем РАН (г. Переславль-Залесский).

Т-система использует парадигму функционального программирования для обеспечения динамического распараллеливания программ, что позволило реализовать в Т-системе формы для синхронизации или распределения нагрузки. Функциональный стиль Т-системы совмещается с традиционными языками программирования с помощью расширений языков C, C++ или языка Fortran. Явные параллельные конструкции в языке отсутствуют, и программист в тексте явно не указывает, какие части программы следует выполнять параллельно. Базовые принципы Т-системы опираются на результаты общей теории функционального программирования.

Пусть имеется сложное арифметическое выражение, включающее много подвыражений, заключенных в скобки. Эти подвыражения можно вычислять в любом порядке. В теории функционального программирования этот закон арифметики обобщается на произвольные рекурсивные функции, что дает прямой метод для распараллеливания функциональных программ, построенных из "чистых" функций. Чистые функции – это одно из базовых понятий Т-системы, обозначающее функции без побочных эффектов. В каждый момент времени необходимо выделять готовые к вычислению "подвыражения" и распределять их по имеющимся процессорам. За основу

берется граф, узлы которого представляют вызванные функции, а дуги соответствуют отношению "подвыражение – выражение".

Для добавления функциональной семантики в традиционный язык программирования вводится понятие неготового значения. В языке С это достигается введением дополнительного атрибута у описания переменных. Новое ключевое слово tval в описании "tval int i" определяет переменную, значение которой может быть целым числом или неготовым значением (пока не посчитанным). Для обозначения функций без побочных эффектов дополнительно используется слово tfun, выход Т-функции обозначается словом tout и т. д.

Последовательность шагов в процессе разработки программ на языке ТС

1. Разработка дизайна кода. На этом этапе решается вопрос о том, какие фрагменты алгоритма будут реализованы на языке ТС в виде Т-функций, а какие реализованы в виде привычного последовательно исполняемого кода на стандартных языках последовательного программирования С,С++ или Fortran.
2. Реализация и первичная отладка на однопроцессорном компьютере. Разработанная ТС-программа отлаживается на обычном однопроцессорном компьютере в последовательном режиме без использования Т-системы. Для этого все новые ключевые слова, добавленные в язык программирования, автоматически переопределяются с помощью соответствующих макросов.
3. Отладка на многопроцессорных установках – полнофункциональная отладка в параллельной вычислительной среде.
4. Оптимизация программы с помощью трассировки, профилировки и других средств.

Преимущества Т-системы заметны в следующих ситуациях:

- до выполнения программы отсутствует информации о механизмах балансировки работы параллельных процессов;

- вычислительная схема программы может быть реализована совокупностью рекурсивно вызывающих друг друга функций. Т-система реализует организацию параллельных фрагментов программы, их распределение по узлам кластера, синхронизация работы фрагментов, явные операции обмена данными между ними.

Недостатки Т-системы:

- вызов Т-функции может вызвать пересылку данных из одного узла кластера в другой, что повлечет дополнительные накладные расходы. Этот фактор может влиять на эффективность, если его не учесть на этапе проектирования программы;
- балансировка вычислительной нагрузки лежит на системе, но гарантии оптимальности она не дает;
- при реализации программ для Т-системы программист обязан изложить алгоритм в функциональном стиле и описать программу в виде набора чистых Т-функций;
- программист отвечает за выбор оптимального размера потенциально параллельных фрагментов. Т-функция имеет невысокую вычислительную сложность, что повышает финансовые расходы. Большая вычислительная сложность - к малому количеству порождаемых в процессе параллельных фрагментов и, как следствие, к неравномерной загрузке вычислительных узлов системы.

Система программирования НОРМА

Первоначально термин НОРМА расшифровывался как Непроцедурное Описание Разностных Моделей Алгоритмов. Затем появилась и другая трактовка: НОРМАльный уровень общения прикладного математика с компьютером.

Язык НОРМА является специализированным непроцедурным языком, предназначенным для спецификации задач вычислительного характера, в частности, задач математической физики. Идеи, позволяющие автоматически

строить программу по спецификации задачи, были сформулированы И. Б. Задыхайло в работе в 1963 году.

Разработчик прикладных программ абстрагируется от особенностей конкретных компьютеров и мыслит в привычных терминах своей предметной области. Отталкиваясь от конкретных потребностей Института прикладной математики им. М. В. Келдыша РАН, авторы языка старались максимально упростить решение класса задач математической физики. Специфика предметной области – это ориентация на сеточные методы. Именно этот факт наложил значительный отпечаток как на концепцию языка, так и на все его основные конструкции.

Конструкции языка носят декларативный характер и описывают правила вычисления значений. Язык предназначен для автоматизированной разработки программ. Транслятор выполняет синтез выходной программы помимо традиционных задач, в частности, синтаксического и семантического анализа. Порядок предложений языка может быть произвольным. Организация вычислений не принимается в расчет при формировании запросов на вычисления. Рассматриваемый язык позволяет формулировать запрос на вычисления, не уточняя, каким именно образом вычисления следует организовать. Все информационные связи выявляются и учитываются транслятором-синтезатором на этапах анализа исходной программы и синтеза выходного текста. На трансляторе лежит и выбор конкретного способа организации вычислений. В частности, на этапе синтеза результирующей программы он может сгенерировать как последовательный, так и параллельный код.

НОРМА – язык, использующий однократное присваивание.

Каждая переменная может принимать значение только один раз. Такие понятия, как память, побочный эффект, оператор присваивания и управляющие операторы в языке НОРМА отсутствуют просто "по определению". Запись на языке НОРМА, по существу, является записью численного метода решения конкретной задачи.

НОРМА позволяет опустить этап последовательного программирования, а для генерации параллельной программы предлагает сразу отталкиваться от записи в терминах математических формул.

Выходные программы могут быть записаны на языках Fortran MPI, Fortran PVM, Fortran 77 и других диалектах Fortran.

Преимущества: соотношения как язык описания заданий наиболее близки разработчикам численных методов и прикладным программистам. Сохраняется весь внутренний параллелизм реализуемого алгоритма. Математические соотношения не используют понятие памяти, в них отсутствует пересчет значений переменных. Для таких языков существующая технология определения параллелизма в программах реализуется значительно проще.

Описание языка на сайте http://www.keldysh.ru/nonna.

Глава 16. Методы исследования и эквивалентные преобразования параллельных алгоритмов и программ

16.1. Информационная структура алгоритмов и программ

Развитие вычислительной техники вызывает модернизацию прикладного программного обеспечения. Как правило, ускорение вычислений достигается за счет отображения характерных особенностей программ и алгоритмов в аппаратуре. Основных элементы программ — циклы, ветвления, процедуры и т. п. находят отражение в буферах команд и регистрах, кэш-памяти, аппаратурной организации стека и т.д. Аппаратные решения, касающиеся:

- кэш-памяти, не требуют программной поддержки;
- виртуальной памяти, требуют динамической поддержки;
- прямо адресуемых регистров, требуют предварительного распределения работы на этапе компиляции после проведения статического анализа программ

Очевидна тенденция: характерные особенности вычислений находят свое отражение в архитектурных особенностях компьютеров, а дополнительные аппаратурные возможности стимулируют развитие новых алгоритмов.

Примером могут служит векторные операции (разновидность групповых операций в математике), открытие которых положило начало появлению векторных компьютеров. К моменту широкого внедрения параллельных вычислительных систем в практику решения больших задач, вычислительная математика оказалась без нужных знаний. Таким образом, появление вычислительных систем параллельной архитектуры привело к развитию численных методов.

Но при несомненных достижениях в вопросах конструирования высокопроизводительных параллельных систем технология организации на них самих вычислительных процессов остаются на относительно низком

уровне. Например, языки программирования становятся все более зависимыми от особенностей архитектур вычислительных систем и начинают приобретать черты языков низкого уровня. Например, такие языки, как PVM и MPI по существу уже являются коммуникационными автокодами, хотя и на макроуровне. **На пользователя все в большей степени перекладывается забота об эффективности организации вычислительных процессов, включая обнаружение в программах и алгоритмах необходимых свойств параллельности и коммуникационных свойств.**

В связи с постоянной нехваткой должной поддержки со стороны языков программирования, компиляторов и операционных систем в обеспечении эффективности процессов решения задач возникла необходимость создания специализированных программных комплексов по анализу пользовательских программ и их преобразованию под требования конкретных параллельных языков программирования и собственно параллельных вычислительных систем. Эти комплексы реализуются на автономных компьютерах и не связаны с компиляторами целевых вычислительных систем. На них могут быть реализованы самые передовые методы исследования и преобразования программ. Комплексы представляют собой автономные программные системы и являются удобным инструментом для выполнения различных работ, когда программы одного вида нужно перевести в эквивалентные программы другого вида. Среди зарубежных автономных систем наиболее известна система PARAFRASE и созданная на ее основе серия пакетов KAP, система PORGE, отечественная система V-Ray.

Можно выделить следующие особенности таких систем:

Значительное уменьшение времени реализации программ на параллельных вычислительных системах. По сравнению с тем временем, которое показывают программы, пропущенные через штатные компиляторы без предварительной оптимизации, почти всегда удается получать ускорение

в несколько раз. Оно тем больше, чем больше и сложнее исходные пользовательские программы.

Необходимость применения, для достижения предельно возможного ускорения, существенно более сложные методы исследования и преобразования программ, чем те, которые традиционно включаются в компиляторы. Сейчас они настолько сложны, что об их ручном применении не может быть и речи. Однако для компьютерного и, тем более, автономного использования эти методы вполне приемлемы.

Для успешного освоения вычислительных систем параллельной архитектуры пользователям необходимо иметь возможность получать самые подробные сведения о самых различных деталях структуры своих алгоритмов и программ. Получение и использование таких сведений связано с изучением информационной структуры алгоритмов и программ (информационных связей на уровне отдельных операций или отдельных срабатываний отдельных операторов).

Информационная структура программы есть совокупность сведений о том, как отдельные элементы программы связаны между собой.

16.2. Графовые модели программ

Целью рассмотрения моделей для программ на последовательных языках программирования является решение таких вопросов, как:

- проведение реорганизации программы, с целью минимизации числа используемых переменных,

- оптимальное использование кэш-памяти,

- минимизация числа обменов с медленной памятью и т.д.

- проблемы, связанные с выявлением в программах скрытого параллелизма и свойств, необходимых для эффективного его использования.

Существует два подхода к изучению информационной структуры программ:

Денотационный подход опирается на исследование состояния памяти программ. В практических приложениях он используется редко.

Операционный подход, при котором исполнение (развитие) любой программы представляется в виде набора выполненных действий (операций), некоторым образом связанных между собой. Содержание действий, их вычислительная мощность и способ связи в рамках данного подхода не фиксируются. Они определяются в каждом конкретном случае по-своему в зависимости от целей исследования. Операционный подход в практических приложениях используется очень часто. Этот подход порождает класс моделей программ, называемых графовыми. В этом классе каждая модель представляет граф, в котором вершины соответствуют каким-то множествам действий программы, а дуги (ребра) так или иначе связаны с отношениями между ними. Диапазон уровней сложности графовых моделей огромен. Отдельная вершина может представлять и программу целиком, и какие-то ее большие или малые фрагменты и даже отдельные элементарные операции. Также приходится учитывать многообразие связей.

В программе можно выделить два типа действий:

-преобразователи, которые осуществляют переработку информации. Например, операторы присваивания. Их левая часть определяет область памяти программы, подвергающейся воздействию преобразователя, а переменные из правой части показывают на необходимые для выполнения данного действия аргументы.

- распознаватели, которые определяют последовательность срабатываний преобразователей в процессе работы программы. Например, альтернативные операторы: условные, разного рода переключатели и т. п. Основное их назначение состоит в выборе одной из нескольких возможных альтернатив дальнейшего следования.

В целях упрощения исследования будем предполагать, что множества операторов, соответствующих этим двум типам действий, не пересекаются.

Пример. Пусть решается система линейных алгебраических уравнений Ax = b с левой двухдиагональной матрицей порядка n. Обозначим через a_1, ..., a_n диагональные элементы матрицы A, через c_1, ..., c_n – ее поддиагональные элементы, через b_1, ..., b_n – правые части, через x_1, ..., x_n – координаты решения. Необходимо найти максимальную координату вектора-решения.

Запишем алгоритм 1:

```
y=b1/a1
x=y
DO i=2, n
 x=(bi-cix/ai)
if (x<=y) go to 7
y =x
END DO
```

После выполнения программы значение переменной *y* будет равно максимальному из чисел x_1, ..., x_n. Здесь операторы с метками 1, 2, 4, б являются преобразователями, операторы с метками 3, 5, 7 – распознавателями.

Между действиями программы существуют связи двух типов:

– связь по управлению или операционная связь определяется фактом выполнения одного действия непосредственно за другим.

– информационную связь существует в случае, если одно действие использует в качестве аргументов результатов выполнения других действий. Каждому оператору исходной программы поставим в соответствие вершину графа – экземпляр преобразователя или распознавателя в зависимости от типа оператора. Получим множество вершин, между которыми согласно исходной программе определим отношение, соответствующее передаче управления. Если текст программы допускает выполнение одного оператора непосредственно за другим, то соответствующие вершины соединим дугой, направленной от предшественника к последователю. Получим граф, который обычно называется графом управления, управляющим графом, графом

передач управления или просто **графом программы**. Его основным свойством является независимость от входных данных программы. Множества вершин и дуг для каждой программы фиксированы и образуют единственный граф. Этот граф задает одну из моделей программы. Для рассмотренного выше примера, граф представлен на рис. 53.

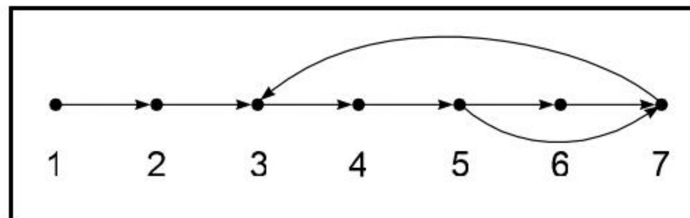


Рис. 53. Графовая модель алгоритма 1

При определении начальных данных программы и наблюдении за ее выполнением на обычном последовательном вычислителе, каждое срабатывание каждого оператора можно фиксировать отдельной вершиной. Получим множество, которое количественно почти всегда будет отличаться от множества вершин графа управления. Соединив вершины дугами передач управления, получим ориентированный граф, носящий название **операционно-логической истории программы** – последовательность срабатывания преобразователей и распознавателей исходной программы при заданных входных данных. Он является единственным путем от начальной вершины к конечной. В операционно-логической истории от входных данных зависит практически все: общее число вершин, количество вершин, соответствующих одному оператору, и даже набор присутствующих преобразователей и распознавателей. Граф управления свободен от подобных конкретностей. На рис. 54 представлена операционно-логическая история вышеприведенного примера программы для случая $n = 3$, $b_1 = 0$, $b_2 = b_3 = a_1 = a_2 = a_3 = c_2 = c_3 = 1$.

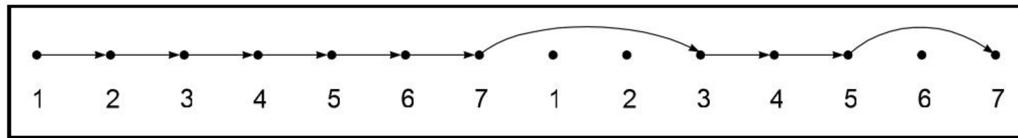


Рис. 54. Операционно-логическая история алгоритма 1 случая $n = 3$, $b_1 = 0$, $b_2 = b_3 = a_1 = a_2 = a_3 = c_2 = c_3 = 1$

Изменим графовую основу. Будем среди операторов принимать во внимание только преобразователи, а в качестве отношения между ними брать отношение информационной зависимости. Построим сначала граф, в котором вершины соответствуют операторам-преобразователям. Две вершины соединим информационной дугой, если между какими-нибудь срабатываниями соответствующих операторов теоретически возможна информационная связь. Полученный граф называется **информационным графом программы**. Как и в случае графа управления, информационный граф не зависит от входных данных. Информационный граф представляет одну из моделей программы. Эта модель используется в системе PARAFRASE и серии пакетов КАР. На рисунке 55 представлен информационный граф программы.

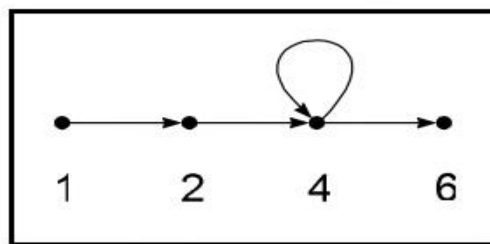


Рис. 55. Информационный граф алгоритма 1

Определим начальные данные программы и проведем наблюдение за ее выполнением на последовательном вычислителе. Каждое срабатывание каждого оператора-преобразователя будем фиксировать отдельной вершиной. Соединив вершины дугами передач информации, мы получим ориентированный граф, называемый **историей реализации программы**. На

рисунке 56 представлена история реализации программы для входных данных, соответствующих рис. 54

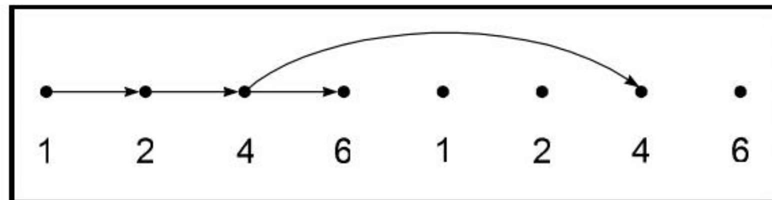


Рис. 56. История реализации программы для входных данных, соответствующих рис. 54

Объединение графов историей реализации программы даст информационным графом программы.

Рассмотренные типы моделей существуют для всех программ.

Трудности определения информационных связей между операциями по тексту программы привели к появлению моделей, называемых **графами зависимостей**. В этих моделях отношение информационной связи заменяется отношением зависимости. Именно, две операции или два оператора называются зависимыми, если при их выполнении имеют место обращения к одной и той же переменной (ячейке памяти). В таких моделях мы работаем только с операторами-преобразователями.

В рассмотренном примере программы имеются две изменяемые переменные x и y. Если в качестве вершин графа зависимостей взять операторы-преобразователи с метками 1, 2, 4, 6, то получим граф, представленный на рис. 57.

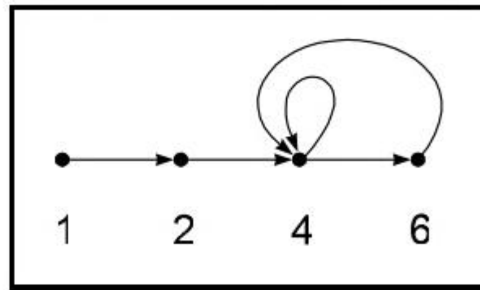


Рис. 57. Граф, полученный для случая, когда в качестве вершин графа зависимостей взяты операторы-преобразователи с метками 1, 2, 4, 6

Среди графовых моделей история реализации программы наиболее интересна.

Историю реализации программы часто называют **решетчатым графом алгоритма** или **графом алгоритма**. На основе методологии построения, исследования и применения графа алгоритма и связанных с ним других графов создана практически работающая автономная система V-Ray.

Задания

1. Что означает случай, когда в программе нет преобразователей? Имеют ли смысл такие программы?

2. Как выглядят управляющие и информационные графы для п.п. 1, 2?

Что означает несвязность управляющего или информационного графа с точки зрения параллельных вычислений?

Может ли быть управляющий граф несвязным, а информационный граф связным и наоборот?

Докажите, что если информационный граф несвязный, то граф алгоритма также несвязный?

Верно ли аналогичное утверждение относительно управляющего графа?

Докажите, что если пересечение множеств переменных, находящихся в правых и левых частях преобразователей, пустое, то граф алгоритма также пустой, т. е. не имеет ни одной дуги.

Если граф алгоритма пустой, то означает ли это пустоту управляющего и/или информационного графа?

Приведите примеры, когда граф влияния не совпадает с графом зависимостей.

Приведите примеры, когда граф алгоритма связный, а граф влияния пустой.

16.3. Классы программ

Граф алгоритма существует для любой программы, но затруднительно разработать эффективный метод его построения и исследования для любой программы. В этой связи целесообразно выделить класс программ, который должен быть достаточно широким, чтобы покрывать значительную часть или все встречающиеся программы или их наиболее значимые фрагменты. Также нужно учитывать, что чем шире класс, тем труднее провести одинаково глубокий анализ.

Класс анализируемых программ имеет двухуровневую структуру. Первый или базовый уровень образует строго описанный **линейный класс**. Для любой его программы проводится самый тщательный анализ информационной структуры. Второй уровень открытый. В него входят все программы, которые каким-либо способом сводятся к программам базового уровня.

Алгоритм может быть записан с помощью следующих средств языка:

- в программе может использоваться любое число простых переменных и переменных с индексами;
- единственным типом исполнительного оператора может быть оператор присваивания, правая часть которого есть арифметическое выражение; допускается любое число таких операторов;
- все повторяющиеся операции описываются только с помощью циклов БО; структура вложенности циклов может быть произвольной; шаги

изменения параметров циклов всегда равны +1; если у цикла нижняя граница больше верхней, то цикл не выполняется;

- допускается использование любого числа условных и безусловных операторов перехода, передающих управление "вниз" по тексту; не допускается использование побочных выходов из циклов;
- все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются, в общем случае, неоднородными формами, линейными как по параметрам циклов, так и по внешним переменным программы; все коэффициенты линейных форм являются целыми числами;
- внешние переменные программы всегда целочисленные, и вектора их значений принадлежат некоторым целочисленным многогранникам; конкретные значения внешних переменных известны только перед началом работы программы и неизвестны в момент ее исследования.

Программы, удовлетворяющие описанным условиям, будем называть линейными или **принадлежащими линейному классу**.

Подход к изучению программ называется **статическим**, если анализ информационной структуры программ из линейного класса, опирающийся только на анализ текста программ, без привлечения каких-либо дополнительных сведений как о самих программах, так и об описанных ими алгоритмах. Его основное достоинство заключается в том, что вся информация о структуре программы получается до ее реализации и, следовательно, может быть использована наиболее эффективно. Главная трудность проведения статического анализа связана с тем, что тексты программ почти всегда зависят от внешних переменных, значения которых не известны. Поэтому все исследования приходится проводить для параметризованных объектов.

1. Рассмотрите любой известный вам язык программирования. Как на этом языке описывается линейный класс программ?

2. Приведите различные примеры программ, которые формально не являются линейными, но становятся таковыми после каких-то эквивалентных преобразований.

3. 'Пусть программа содержит вызовы процедур. Предложите различные способы замены этих вызовов программными фрагментами, при которых исходная программа становится линейной и сохраняет все прежние информационные зависимости.

4. Приведите различные примеры программ, которые формально не являются линейными и которые вы не можете привести к линейным с помощью эквивалентных преобразований.

5. Как выглядят графы алгоритмов для примеров п. 4?

6. Насколько различны графы алгоритмов для примеров п. 4?

7. Как часто примеры п. 4 встречались в вашей практике?

16.4. Эквивалентные преобразования параллельных алгоритмов и программ

Эквивалентным преобразованием будем называть любое преобразование программы в эквивалентную ей программу.

Программы будем называть эквивалентными по графам зависимостей, если:

- пространства итераций программ эквивалентны;
- соответствие эквивалентности является изоморфным для графов;
- сопоставляемые при изоморфизме дуги графов зависимостей относятся к входным переменным с одними и теми же номерами.

Графы зависимости эквивалентных программ назовем графами эквивалентности

Программы, графы алгоритмов которых эквивалентны, являются различными формами одного и того же алгоритма. Однако эти программы могут иметь различный объем памяти для вычислений и отличаться способами создания массивов, а также порядком в наборах входных и выходных данных

Массовый параллелизм в преобразованной программе представлен в форме циклов ParDo. Преобразование выполняется над пространством итераций и изменяет используемые переменные.

Последовательность преобразования:

1. Выполняется эквивалентное преобразование пространства итераций.
2. Для новой программы выбираются такие переменные, чтобы при преобразовании пространства итераций разные переменные переходили в разные, одинаковые — в одинаковые.
3. В новом пространстве итераций устанавливается лексикографический порядок, при котором образ графа зависимостей будет лексикографически правильным.

Эквивалентное преобразование программы осуществляется с целью описания параллелизма в программе при помощи специальных циклов.

16.5. Оптимизация программ. Система V-Ray

Для успешного использования параллельных вычислительных систем необходимо создание эффективных параллельных программ. Эффективность параллельных программ связана с их оптимизацией, которая существенно упрощается благодаря наличию инструментария, подобного системе V-Ray, которая является инструментом для выполнения различных работ в процессе перевода программы одного вида в эквивалентные программы другого вида.

Проблема анализа и повышения эффективности параллельных программ — комплексная и вопросы, связанные с ней могут возникать на самых разных уровнях при анализе:

- конфигурации компьютере;
- эффективности системного и программного обеспечения;
- прикладной программы;
- алгоритмического подхода.

Например, некоторая программа, написанная для векторно-конвейерного компьютера Cray C90, при эксплуатации пользователем

выдавала время работы на 30% больше заявленного. Детальный анализ показал, что истинная причина крылась не в программе: процессор второго компьютера, в отличие от первого, имел лишь три канала, а не четыре, как предписано стандартной конфигурацией.

Система V-Ray предназначена для изучения параллельной структуры последовательных программ. Применение этой системы не требует никакой предварительной информации ни о структуре программы, ни о ее математическом содержании. V-Ray не привязана к архитектуре вычислительной системы.

Особенность системы – заключается в том, что впервые система анализа и адаптации программ под архитектуру параллельных компьютеров базируется на использовании истории реализации программ [2]. В отличие от существующих методов, данный подход опирается на компактное параметрическое описание информационной истории реализации программы, прогнозируя динамику поведения программ по их статической форме – исходному тексту. Разработанные методы, не только дают максимально точный ответ, но и позволяют гарантировать **неулучшаемость** полученных результатов.

Заключение

Вопросы, рассмотренные в настоящем пособии касаются основ параллельных вычислений. Каждый из этих вопросов требует более глубокого изучения, что не являлось целью данного учебного пособия. Однако степень детализации изучаемых понятий является вполне достаточной для получения базовых знаний в области параллельных методов и алгоритмов. Параллельные вычисления являются постоянно развивающейся областью. Рост сложности практических задач требует применения все более мощных вычислительных средств, а следовательно более мощного математического и программного обеспечения.

В настоящее время существуют сверхсложные вычислительные задачи, решение которых возможно только при помощи суперкомпьютеров [6]: предсказания погоды, климата и глобальных изменений в атмосфере; науки о материалах; построение полупроводниковых приборов; сверхпроводимость; структурная биология; разработка фармацевтических препаратов; генетика человека; квантовая хромодинамика; астрономия; транспортные задачи; гидро- и газодинамика; управляемый термоядерный синтез; эффективность систем сгорания топлива; разведка нефти и газа; вычислительные задачи наук о мировом океане ; разпознавание и синтез речи; разпознавание изображений.

Представленный в учебном пособии инструментарий является необходимым для решения указанных задач.

Список литературы

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ – Петербург, 2002. - 608 с.: илл.

2. Якобовский М. В. Введение в параллельные методы решения задач: Учебное пособие / Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2013. – 328 с., илл.

3. Д.И. Батищев, Е.А. Неймарк, Н.В. Старостин. Применение параллельных вычислений для повышения эффективности генетического поиска - Международный научный журнал «ИННОВАЦИОННАЯ НАУКА». № **4**/2015.

4. Учебно-методическое обеспечения самостоятельной работы студентов по курсу «Структуры и алгоритмы данных», 2007. – Красноярск, 51с.: илл., files.lib.sfu-kras.ru.

5. Лаборатория параллельных вычислений Научно-исследовательского вычислительного центра Московского государственного университета имени М.В.Ломоносова. https://parallel.ru/.

Учебное издание

Александра Владимировна Волосова

Параллельные методы и алгоритмы

Редактор В.В. Виноградова

Подписано в печать 13.01.2020.

Уч.-изд. л. 11. Объем данных 5,35 Мбайт.
Московский автомобильно-дорожный государственный
технический университет (МАДИ)
http://www.madi.ru, info@madi.ru, 125319, Москва, Ленинградский пр-т, 64.