Издание серии «Суперкомпьютерное образование» наглядно демонстрирует тот вклад, который внесли участники Суперкомпьютерного консорциума университетов России в создание национальной системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий, а также их четкое понимание ответственности за подготовку высококвалифицированных специалистов и формирование прочного научного фундамента, столь необходимого для эффективного использования суперкомпьютерных технологий на практике.

Ректор Московского университета
Президент Суперкомпьютерного консорциума
университетов России
академик РАН *В.А. Садовничий*

Оглавление

Глава 1. Введение 11
1.1. Современный компьютер — инструмент параллельной обработки данных 11
1.2. Области применения многопроцессорных систем 16
1.3. Рассматриваемые параллельные архитектуры 19
1.4. Пример параллельного алгоритма 20
1.4.1. Последовательный рекурсивный алгоритм 21
1.4.2. Параллельный рекурсивный алгоритм 21
1.4.3. Последовательное вычисление членов ряда 22
1.4.4. Последовательный матричный алгоритм 23
1.4.5. Параллельный матричный алгоритм 24

Глава 2. Основные понятия 28
2.1. Параллельная программа как ансамбль взаимодействующих последовательных процессов 28
2.2. Внутренний параллелизм 29
2.2.1. Сложение многоразрядных чисел 30
2.3. Ускорение и эффективность параллельных алгоритмов 38
2.4. Ускорение и эффективность относительно наилучшего последовательного алгоритма 41
2.4.1. Неравноправность условий выполнения — первая причина сверхлинейного ускорения 43
2.4.2. Алгоритмическая причина сверхлинейного ускорения 44
2.4.3. Формальное преобразование параллельного алгоритма в «наилучший» последовательный 47
2.5. Априорная оценка эффективности параллельного алгоритма 49

Глава 3. Модели параллельных программ 53
3.1. Вычислительные системы с распределенной памятью 54
3.2. Вычислительные системы с общей памятью 56
3.3. Гибридные архитектуры 58
3.4. Модель выполнения параллельной программы на распределенной памяти 59
3.5. Модель выполнения параллельной программы на общей памяти 60
3.6. Средства взаимодействия последовательных процессов 62
3.6.1. Свойства канала передачи данных 62
3.6.2. Методы передачи данных 63

3.6.3. Семафор ... 72
3.6.4. Барьерная синхронизация ... 75

Глава 4. Базовые параллельные методы ... 77
4.1. Метод сдваивания ... 78
4.1.1. Быстрый алгоритм выбора частичных сумм ... 78
4.1.2. Барьерная синхронизация на основе синхронных обменов ... 84
4.1.3. Стена Фокса ... 86
4.2. Метод геометрического параллелизма ... 88
4.3. Метод конвейерного параллелизма ... 88
4.4. Метод коллективного решения ... 98
4.5. Причины потери эффективности ... 107
... 114

Глава 5. Сортировка данных ... 117
5.1. Постановка задачи ... 117
5.2. Последовательные алгоритмы сортировки ... 120
5.2.1. Быстрая сортировка (*runtime qsort*, *wsort*) ... 123
5.2.2. Простое двухпутевое слияние (*dsort*) и слияние списков (*lsort*) ... 124
5.2.3. Пирамидальная сортировка (*hsort*) ... 126
5.3 Свойства последовательных алгоритмов ... 128
5.3.1. Сортировка методом простого двухпутевого слияния ... 128
5.3.2. Пирамидальная сортировка ... 135
5.3.3. Наилучший последовательный алгоритм сортировки *dhsort* ... 139
5.4. Масштабируемые алгоритмы сортировки ... 141
5.4.1. Сети сортировки ... 141
5.4.2. Сеть четно-нечетной сортировки ... 144
5.4.3. Сеть обменной сортировки со слиянием Бэтчера ... 145
5.4.4. Сортировка больших массивов ... 150
5.4.5. Сравнение алгоритмов сортировки ... 153
5.5. Результаты численных экспериментов ... 157

Глава 6. Генерация псевдослучайных чисел ... 161
6.1. Требования к генераторам псевдослучайных чисел для МВС ... 162
6.2. Линейно-конгруэнтные генераторы ... 168
6.3. М-последовательности ... 170
6.4. Проверка примитивности полиномов ... 176
6.5. Тестирование генераторов ... 177

Глава 7. Декомпозиция сеточных графов ... 180
7.1. Пример двумерной сетки ... 180
7.2. Критерии декомпозиции графов ... 182
7.2.1. Критерий 1: классический критерий декомпозиции графа ... 183
7.2.2. Критерий 2: выделение обособленных доменов ... 184
7.2.3. Критерий 3: минимизация максимальной степени домена ... 185
7.2.4. Критерий 4: обеспечение связности графов каждого из доменов ... 186
7.3. Декомпозиция на основе исходной нумерации узлов ... 189
7.4. Рекурсивная бисекция ... 191
7.5. Декомпозиция регулярных графов ... 192
7.6. Методы декомпозиции произвольных графов ... 196
7.6.1. Иерархическая декомпозиция ... 196
7.6.2. Спектральная бисекция ... 205
7.6.3. Алгоритм инкрементного роста ... 210
7.7. Декомпозиция больших сеток ... 217
7.7.1. Координатная рекурсивная бисекция ... 218
7.7.2. Двухуровневая стратегия обработки и хранения сеток ... 220

Глава 8. Динамическая балансировка загрузки процессоров ... 223
8.1. Стратегии балансировки загрузки ... 223
8.2. Метод диффузной балансировки ... 224
8.3. Моделирование горения метанового факела ... 229
8.3.1. Постановка задачи динамической балансировки ... 235
8.3.2. Алгоритм серверного параллелизма ... 236
8.4. Адаптивное интегрирование ... 248
8.4.1. Последовательные алгоритмы ... 249
8.4.2. Параллельные алгоритмы ... 256

Глава 9. Визуализация сеточных данных ... 278
9.1. Клиент-серверная технология ... 280
9.2. Online или Offline-визуализация: плюсы и минусы ... 281
9.2.1. Online-визуализация Offline-визуализация ... 281
9.3. Этапы визуализации ... 285
9.4. Визуализация изоповерхностей ... 290
9.4.1. Аппроксимация изоповерхности ... 293
9.4.2. Виды данных, описывающих триангуляцию ... 296
9.4.3. Метод редукции ... 299

9.4.4. Заполняющие пространство триангуляции ... 302
9.4.5. Параллельные алгоритмы построения аппроксимирующих триангуляций ... 303
9.4.6. Многоуровневое огрубление больших сеток ... 304
9.4.7. Примеры визуализации ... 306
9.5. Ввод-вывод сеточных данных ... 308
9.5.1. Соотношение времени чтения данных и времени их обработки ... 308
9.5.2. Распределенный ввод-вывод ... 309
9.5.3. Огрубление и сжатие скалярных сеточных функций ... 310

Список ссылок ... 317

Предметный указатель ... 323

Глава 1

Введение

Современный компьютер — надежный и мощный инструмент для решения самых разных задач. С его помощью можно как играть в компьютерные игры, так и планировать и выполнять сложные банковские операции. Как и положено любому востребованному инструменту, компьютер постоянно совершенствуется, в первую очередь, растет его производительность. На протяжении десятилетий вычислительная мощность компьютера росла за счет увеличения вычислительной мощности его процессора. Единожды написанная программа выполнялась на новых компьютерах быстрее, чем на старых именно потому, что новый процессор затрачивал на выполнение требуемого множества инструкций программы меньше времени. Так было раньше, но теперь ситуация существенно изменилась. Время работы последовательной программы перестало сокращаться, несмотря на то, что производительность процессора продолжает расти.

1.1. Современный компьютер — инструмент параллельной обработки данных

Рост производительности процессора определяется двумя основными факторами: рабочей частотой и архитектурой. На рис. 1 показаны характеристики процессоров Intel, обеспечивающих вычислительную мощность суперкомпьютеров верхней части списка пятисот крупнейших систем мира: top500.org [1]. На рисунке 1 приведены частота и производительность *одного процессорного ядра*. При построении графика рассматривались системы традиционной архитектуры, не содержащие

специализированных графических ускорителей, в том числе графических карт общего назначения. Производительность оценивалась как отношение пиковой производительности системы к общему числу процессорных ядер [2].

Таблица 1

Характеристики процессоров

Дата рейтинга	CPU, Intel	Частота (GHz)	Производительность ядра CPU (Gflops)	Число ядер	Производительность CPU (Gflops)	Позиция в top500
01.11.11	Xeon E5450 4C 3.00 GHz	2,93	12,8	4	47	7
01.11.10	Xeon 7500 Nehalem-EX	2,26	11,7	8	72	6
01.11.09 01.11.08	Xeon E54xx Harpertown	3	11,9	4	48	6 3
01.11.07	Xeon 53xx Clovertown	3	12	4	48	3
01.11.06	Xeon	3,6	7,2	2	14	6
01.11.05 01.11.04	Itanium 2	1,5	6	4	24	4 2
01.11.03	Pentium 4 Xeon	3,06	6,1	2	12	4
01.11.02	Pentium 4 Xeon	2,4	4,8	2	9,6	5
01.11.01 01.11.00	Pentium Pro	0,33	0,33	1	0,33	4 2

Как следует из таблицы 1, частота процессора (нижний график на рис. 1) не увеличивается уже несколько лет. Напротив, в последние годы отмечается ее снижение, обусловленное, в первую очередь, необходимостью уменьшения удельного энергопотребления вычислительных систем (отношения энергопотребления к вычислительной мощности). Производительность процессора, тем не менее, продолжает расти (рис. 2). Отметим крайне важную архитектурную особенность рассматриваемых процессоров: начиная с 2006 года они являются *многоядерными*. Это означает, что в одном корпусе на сверхбольшой интегральной схеме (СБИС) размещено несколько полноценных процессоров. Фактически следовало бы говорить «многопроцессорная СБИС», но вместо этого продолжают использовать термин *процессор* (или *многоядерный процессор*), содержащий несколько *процессорных ядер*.

Рассмотрим подробнее, за счет чего на протяжении последних десяти лет росла производительность процессора. На рис. 3 приведено отношение производительности одного процессорного ядра (верхняя кривая рис. 1) к частоте работы процессора (нижняя кривая рис. 1). Полученные числа

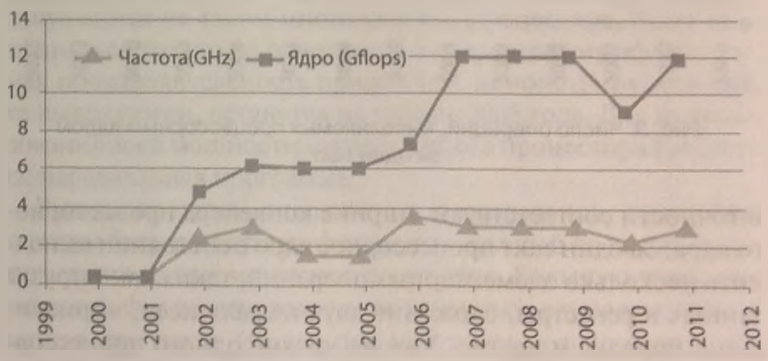


Рис. 1. Частота и производительность процессорного ядра

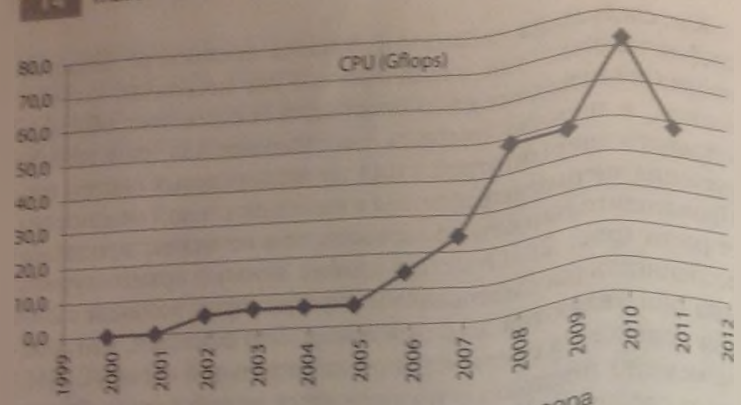


Рис. 2. Производительность процессора

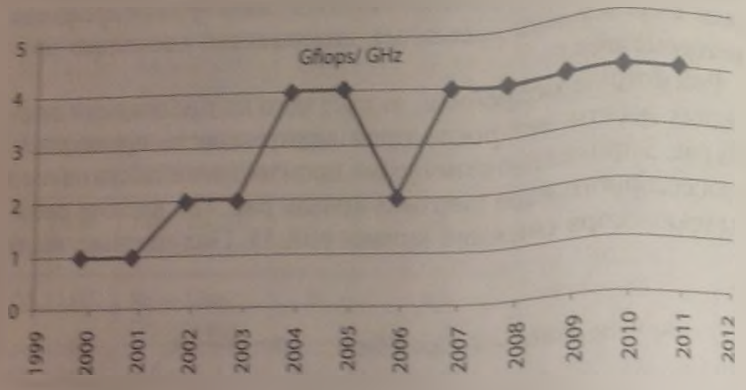


Рис. 3. Число операций, выполняемых процессорным ядром за один такт

в точности соответствуют ширине конвейера процессорного ядра. За один такт процессорное ядро в состоянии выполнить несколько элементарных операций, таких как загрузка данных в регистры, сложение двух целых чисел, нормализация порядка и другие. Уже на уровне одного процессорного ядра организована параллельная обработка данных, за использование которой отвечает компилятор и аппаратура

процессора. Далеко не всякий программный код содержит достаточное множество взаимно независимых элементарных инструкций, выполнять которые можно одновременно, поэтому фактическая производительность ядра при выполнении конкретной программы часто оказывается значительно ниже пиковой.

Итак, в 2001–2005 годах производительность процессора увеличивалась пропорционально тактовой частоте и ширине конвейера, но в дальнейшем и ширина конвейера, и тактовая частота перестали расти. Число размещаемых на одном кристалле процессора транзисторов продолжало увеличиваться, но новых способов использовать эти транзисторы для повышения производительности одного процессорного ядра не появилось, а старые себя исчерпали. Дальнейшее развитие процессоров идет экстенсивным путем — невостребованные транзисторы используются для изготовления дополнительных процессорных ядер. Последующий рост производительности процессора обеспечивается увеличением числа процессорных ядер, содержащихся в одном корпусе СБИС. Фактически, в одном корпусе теперь размещено несколько, независимых с логической точки зрения, процессоров. Последовательная программа, запущенная на таком многоядерном процессоре, будет выполняться только на одном из его ядер, поэтому эффективная производительность процессора, демонстрируемая при ее выполнении, останется на уровне 2005 года. Для использования всей мощности многоядерного процессора требуется параллельная программа.

Вычислительные операции в параллельной программе должны быть разбиты на фрагменты, каждый из которых выполняется на своем процессорном ядре. Большую часть времени эти фрагменты должны выполняться одновременно, не мешая друг другу. Иногда они могут взаимодействовать друг с другом, обеспечивая согласованное совместное решение задачи. Только в этом случае возможно использование для ре-

шения одной задачи вычислительной мощности нескольких процессоров или всей мощности многоядерного процессора.

В общем случае не существует способа автоматического преобразования последовательной программы в параллельную. Эффективные, хорошо зарекомендовавшие себя последовательные алгоритмы часто не поддаются распараллеливанию. В основе параллельной программы может лежать совершенно иной, неэффективный в последовательном случае алгоритм. Практически всегда создание параллельного алгоритма — задача прикладного специалиста. Методы построения параллельных алгоритмов, позволяющих организовать совместную работу нескольких процессоров над одной задачей, и составляют предмет данного курса.

1.2. Области применения многопроцессорных систем

Можно условно выделить основные цели, для достижения которых привлекаются многопроцессорные вычислительные системы:

1) сокращение времени решения вычислительно сложных задач;
2) сокращение времени обработки больших объемов данных;
3) решение задач реального времени;
4) создание систем высокой надежности.

К параллельным алгоритмам, в зависимости от преследуемой цели, предъявляются различные требования, что позволяет говорить о параллельных алгоритмах соответствующих классов.

В рамках настоящего курса, в подавляющем большинстве случаев будем ориентироваться на алгоритмы первого класса, обеспечивающие достижение цели номер 1. При выполнении научных расчетов, как правило, не устанавливаются

жесткие временные рамки. Сроки выполнения научных исследований достаточно лояльны, но присутствует потребность выполнения за определенное время большого числа расчетов, связанных с достаточно сложными моделями, описываемыми большим числом параметров. Очень часто такие расчеты выполняются на оборудовании центров коллективного пользования, чье руководство справедливо рассчитывает на то, что выделенные ресурсы используются эффективно. Если для расчета запрашивается некоторое число процессов, то все они должны активно участвовать в расчете в течение почти всего отведенного времени. Таким образом, в задачах этого класса необходимы алгоритмы, обладающие достаточно высокими показателями и ускорения, и эффективности использования процессорной мощности.

Иначе обстоит дело с алгоритмами второго класса, обеспечивающими достижение цели номер 2. Некоторые из них рассматриваются в главах, посвященных визуализации и декомпозиции сеток и сеточных данных. В алгоритмах этого класса во главу угла ставится возможность обработки за «разумное» время «произвольно» больших объемов данных. Например, в задачах визуализации многомерных результатов научных расчетов необходимо дать исследователю возможность интерактивного анализа массивов данных, размер которых на порядки превышает объемы оперативной памяти и дисковой подсистемы персонального компьютера на рабочем месте. Сделать это, не привлекая для работы многопроцессорный кластер, не представляется возможным. Однако в подобной системе существенным и самым медленным элементом является человек — тот, ради кого, собственно, и формируются визуальные образы. Большую часть времени при работе с системой визуализации он проводит, созерцая визуальные образы, изредка давая запросы на формирование новых образов. Для комфортной работы необходимо, чтобы визуализация выполнялась в интерактивном режиме. Это значит, что время реакции системы на

запрос пользователя должно измеряться секундами и долями секунд. Важным фактором, обеспечивающим высокую скорость реакции, является возможность хранения единожды прочитанных данных в оперативной памяти, поскольку чтение данных с медленных внешних устройств, занимает существенную часть от общего времени их обработки. Для интерактивной визуализации часто хватает значительно меньшего числа процессоров, чем для проведения самих расчетов, в ведущих вычислительных центрах для этих целей выделяются отдельные кластеры визуализации. Следовательно, неэффективное использование их процессоров вполне допустимо, поскольку основная цель — быстрая реакция на запросы пользователя, между запросами процессоры все равно простаивают. Таким образом, здесь основное требование к параллельному алгоритму — высокоскоростная обработка запроса, тогда как высокая эффективность отходит на второй план.

Алгоритмы третьего класса, обеспечивающие решение задач реального времени, дают разработчику максимальную свободу в плане эффективности использования процессорной мощности. При управлении технологической установкой решения должны приниматься в жестко установленные сроки. Если для этого требуется множество процессоров, часть которых почти всегда будет в состоянии ожидания, но в необходимый момент произведет требуемые вычисления, то система выполнит свое назначение. Можно считать, что предъявляются требования по производительности, а значит, по ускорению расчетов, но не по эффективности.

Наконец, алгоритмы четвертого класса востребованы при проектировании автономных вычислительных систем высокой готовности. Например, при разработке бортовой вычислительной системы космического аппарата необходимо учитывать, как минимум, три фактора:

- высокий радиационный фон на орбите служит причиной достаточно частых отказов компонент электронной техники, приводящих к ошибкам в вычислениях;

- необнаруженная ошибка в вычислениях может привести к необратимой потере аппарата;
- внешнее вмешательство в работу системы, ее перезапуск, подобно перезапуску настольного персонального компьютера, в случае отказа практически невозможно.

Традиционный способ радикального снижения вероятности возникновения необнаруженных ошибок — полное дублирование вычислений. Например, бортовая вычислительная система может содержать несколько процессоров, выполняющих одну и ту же программу. По окончании очередного этапа вычислений результаты, полученные каждым из процессоров, сравниваются между собой. В случае их совпадения нормальная работа продолжается. В случае отклонения результатов, полученных одним из процессоров от результатов, полученных другими, его результаты считаются ошибочными и не участвуют в дальнейшей работе, а сам процессор перезапускается, после чего продолжает принимать участие в общей работе. Такое решение обеспечивает живучесть системы и ее высокую готовность — несмотря на возникновение ошибок, производительность системы поддерживается на неизменном уровне. Требования к эффективности здесь тоже не предъявляются.

1.3. Рассматриваемые параллельные архитектуры

Существует множество вычислительных архитектур, но курс ориентирован только на две из них. Будут рассматриваться вычислительные системы с распределенной памятью и вычислительные системы с общей памятью. С одной стороны, это сужение класса рассматриваемых систем, но с другой, выделенные классы отличаются наиболее гибкими возможностями, что позволяет изучить широкий спектр параллельных алгоритмов различного вида. Вычислительные системы на основе графических ускорителей, программи-

руемых интегральных схем, нейрокомпьютеры и многие другие отличаются большей спецификой. Рассматриваемые в курсе алгоритмы могут быть использованы в качестве отправной точки для разработки параллельных приложений и для перечисленных вычислительных систем в том числе. Этот тезис находит свое подтверждение в практике создания параллельных программ для наиболее актуальных и интересных сегодня систем на основе графических ускорителей: как правило, к ним адаптируются имеющиеся программы, а не создаются программы с нуля. Более того, ведутся активные работы по созданию языков и методов программирования, обеспечивающих разработку одного программного кода, допускающего, после обработки соответствующим компилятором, исполнение на кластерных многоядерных вычислительных системах, оснащенных графическими ускорителями (DVM).

Цель курса

Основной целью курса является ознакомление с конструктивными подходами к созданию алгоритмов первого и второго из рассмотренных ранее классов. Основное внимание в его рамках уделено изучению методов создания параллельных программ, обеспечивающих либо существенное сокращение времени решение задачи, либо обработку за разумное время произвольно большого объема данных.

1.4. Пример параллельного алгоритма

В качестве примера, иллюстрирующего цель курса, рассмотрим задачу вычисления чисел Фибоначчи.

Для заданного $n > 1$ требуется определить значение элемента n последовательности Фибоначчи, задаваемой рекурсивным соотношением: $F(n) = F(n-1) + F(n-2)$, при $F(0) = 1$ и $F(1) = 1$.

1.4.1. Последовательный рекурсивный алгоритм

Оценим число операций, требуемое для решения задачи при использовании рекурсивного алгоритма:

Алгоритм А1

Последовательный рекурсивный алгоритм

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

Пусть $G(n)$ — число операций, необходимое для определения числа $F(n)$. Тогда, $G(n) = G(n-1) + G(n-2) + 1$, поскольку следует найти числа $F(n-1)$, $F(n-2)$, для чего потребуется выполнить $G(n-1)$ и $G(n-2)$ операций соответственно, а потом найти их сумму — еще одна операция. Так как значения $F(0)$ и $F(1)$ уже известны, то $G(0) = 0$ и $G(1) = 0$. Нетрудно видеть, что $G_{A1}(n) = F(n) - 1$, что позволяет указать оценку для $G(n)$, воспользовавшись известной асимптотикой для $F(n)$: при $n \to \infty$, $F_n \sim \frac{\varphi^n}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$ — золотое сечение.

$$G_{A1}(n) \sim 0{,}45 \cdot 10^{n/4.78}.$$

1.4.2. Параллельный рекурсивный алгоритм

Оценим потенциал распараллеливания рекурсивного алгоритма, записав его в виде А2. Предполагается, что вызовы функции $F(n-2)$ и $F(n-1)$ обрабатываются одновременно, каждый на своем процессоре.

Алгоритм А2

Параллельный рекурсивный алгоритм

```
F(n)
{
   [illegible]
```

Процессор 1	Процессор 2
a= F(n-2)	b= F(n-1)

```
   return a+b;
}
```

Максимальная глубина рекурсии составит $n - 2$. Найдется как минимум $(n - 2)$ операции, выполнить которые потребуется последовательно, одну за другой. Распараллеливание рекурсивного алгоритма требует большого (порядка $10^{n/5}$) числа процессоров и в лучшем случае позволит лишь приблизиться к производительности приведенного далее алгоритма А3. Оценка числа операций этого алгоритма составляет $(n - 2)$, поскольку именно такой будет глубина самой длинной ветви рекурсии, операции в которой выполняются строго последовательно.

1.4.3. Последовательное вычисление членов ряда

Рассмотрим еще один алгоритм последовательного вычисления чисел Фибоначчи.

Алгоритм А3

Последовательный алгоритм

```
F(n)
{
a=1; b=1; c=1;
for(i=2;i<=n;i++)
{
c=a+b;
b=a;
}
return c;
}
```

Потребуется выполнить $(n-2)$ операции сложения, соответственно, время выполнения алгоритма А3 будет значительно меньше, чем алгоритма А1: $G_c(n) = n - 2$. Это наиболее быстрый из рассмотренных алгоритмов, но в нем результат каждой очередной операции зависит от результата предыдущей. Таким образом, отсутствуют операции, выполнять которые можно было бы одновременно на нескольких процессорах. Иными словами, в алгоритме нет *внутреннего параллелизма*, поэтому на его основе не удастся предложить параллельный алгоритм.

1.4.4. Последовательный матричный алгоритм

Тем не менее, результат можно вычислить быстрее, воспользовавшись, например, матричной формой определения требуемого члена:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = A^n, \text{ где } A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Для возведения матрицы A в степень n следует использовать алгоритм бинарного умножения [3]. Алгоритм основан на рекурсивном применении правила:

$A^k = \left(A^{k/2}\right)^2$, при четном k, и $A^k = \left(A^{(k-1)/2}\right)^2 A$, при нечетном k.

Алгоритм А4

Последовательный матричный алгоритм

```
F(n)
{
  A[2][2]={{1,1},{1,0}};
  B=A;

  for(i=1;i<n;i*=2)
  {
    B=B*B;
```

```
    if(i&n) // истина, если в числе n двоичный разряд
            // с номером log i равен 1
    B=B*A;
  }
  return B[0][1];
}
```

Потребуется выполнить не более $G_{A4}(n) = 8\log_2 n$ арифметических операций, что пока является самым лучшим результатом среди всех рассмотренных. На основе именно этого, самого быстрого из рассмотренных последовательных алгоритмов, можно предложить параллельный алгоритм, который требует ещё меньшего времени для своего выполнения.

1.4.5. Параллельный матричный алгоритм

Алгоритм A4 использует операции матричного умножения, распараллеливание которых вполне возможно. Рассмотрим подробнее способ параллельного выполнения операций внутреннего цикла алгоритма A4 на четырех процессорах.

Пусть $B = \begin{pmatrix} p & q \\ q & r \end{pmatrix}$, тогда: $B^2 A^\beta = \begin{pmatrix} p & q \\ q & r \end{pmatrix}^2 \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^\beta$.

$$B^2 A^\beta = \begin{cases} \begin{pmatrix} p & \bar{q} \\ q & r \end{pmatrix}, \text{ при } \beta = 0 & \bar{p} = p^2 + q^2, \quad \bar{p} = \bar{p} + \bar{q}, \\ & \bar{q} = q(p + r), \quad \bar{q} = \bar{p}, \\ \begin{pmatrix} \bar{p} & \bar{q} \\ \bar{q} & \bar{r} \end{pmatrix}, \text{ при } \beta = 1 & \bar{r} = q^2 + r^2, \quad \bar{r} = \bar{q}. \end{cases}$$

Для вычисления $\bar{p}$, $\bar{q}$, $\bar{r}$ и $\bar{p}$ потребуются три такта (при $\beta = 0$):

Алгоритм A5

Параллельный матричный алгоритм

```
F(n)
{
    p=1; q=1; r=0; // B = (p q; q r) = (1 1; 1 0)
    for(i=1;i<n;i*=2)
    {
```

	Процессор 1	Процессор 2	Процессор 3	Процессор 4
Такт 1	p_p=p·p	q_q =q·q	r_r=r·r	p_r=p+r
Такт 2	p̄=p_p+q_q	r̄=q_q+r_r	q̄=q · p_r	
Такт 3	p̄=p̄+q̄			

```
    if(i&n) // если в числе n двоичный разряд
            //с номером log i равен 1
    {p=p̄; q=q̄; r=r̄}
      else {p=p̄; q=p̄; r=q̄}
    }
    return q;
}
```

Таким образом, для каждого двоичного разряда числа n, вместо 8-ми тактов последовательного выполнения алгоритма A3, потребуется только 3 такта при параллельном выполнении на 4-х процессорах алгоритма A5. Общее число тактов можно оценить как $G_{A5}(n) = 3\log_2 n$. На четырех процессорах параллельный алгоритм A5 будет выполняться в 2,66 раза быстрее наилучшего из рассмотренных последовательных алгоритмов A3. Говорят, что *ускорение* алгоритма A5 равно 2,66.

Мы ограничились рассмотрением только арифметических операций. Учет операций организации цикла, условных операций и им подобных, а также учет особенностей архитектуры процессора и вычислительного узла, несомненно, внесет коррективы в значение достигаемого ускорения, но подобный анализ выходит далеко за рамки настоящего курса.

Подчеркнем, что *масштабируемость* рассмотренного параллельного алгоритма ограничена четырьмя процессорами.

Использование большего числа процессоров не приведет к дополнительному сокращению времени решения задачи, независимо от того, насколько велико значение номера искомого члена последовательности Фибоначчи, и, соответственно, насколько много вычислительных операций потребуется выполнить. Обратим внимание, что процессоры используются не вполне эффективно. На последнем такте работает только один процессор, остальные простаивают. На трех тактах четыре процессора могли бы выполнить в общей сложности 12 операций, но выполняют только 8. Говорят, что эффективность алгоритма равна $\frac{8}{12} = 67\%$. Отметим, что полученная оценка справедлива для наихудшего случая, поскольку вычисление $\bar{p}$ требуется не всегда. Оно необходимо только в случае, если соответствующий двоичный разряд числа n равен 1.

Сравним время выполнения алгоритмов A5 и A1. Для выполнение алгоритма A5 требуется в $\frac{G_{A5(n)}}{G_{A1(n)}} = 0{,}16 \cdot \frac{10^5}{\log_2 n}$ раз меньше тактов. При $n = 10$ мы получим сокращение времени вычислений в 8 раз, при $n = 20$ — в 293 раза, при $n = 30$ — в 31 тысячу раз! Очевидно, что в этом результате нет заслуги ни многопроцессорности, ни параллельности алгоритма. Причина в том, что для сравнения выбран крайне неудачный последовательный алгоритм, что и привело к проявлению эффекта сверхлинейного ускорения. Прежде чем приступать к разработке параллельного алгоритма, следует убедиться, что исчерпаны возможности по созданию хорошего (наилучшего) последовательного алгоритма.

Нашей основной целью будет изучение методов создания параллельных алгоритмов, обеспечивающих существенное уменьшение времени решения задач при использовании многопроцессорных систем. Подчеркнем: параллельный алгоритм должен выполняться во много раз быстрее, чем различные из известных последовательных алгоритмов.

В рассмотренном примере именно самый быстрый последовательный алгоритм послужил основой быстрого параллельного алгоритма. Такое сочетание (высокой производительности и возможности распараллеливания) является скорее исключением, чем правилом. Часто в качестве основы хорошего параллельного алгоритма следует использовать далеко не самый быстрый последовательный алгоритм. Соответствующие примеры медленных, но масштабируемых на большое число процессоров алгоритмов, будут рассмотрены в последующих главах.

Подготовка настоящего учебного пособия выполнена на основе курса, читаемого с 1992 г. на базовой кафедре Математического моделирования Института прикладной математики им. М. В. Келдыша РАН и Института математического моделирования РАН в Московском физико-техническом институте. Автор глубоко благодарен Б. Н. Четверушкину, Е. И. Леванову, И. И. Галигузовой, благодаря вниманию и постоянной поддержке которых стало возможным возникновение и развитие курса. Автор считает своим приятным долгом выразить благодарность своим коллегам: С. В. Полякову за плодотворные дискуссии по широкому кругу затронутых в курсе вопросов; Е. Н. Головченко за многолетнее сопровождение практикума и за совместную работу над алгоритмами декомпозиции сеток; П. С. Крикову и С. В. Муравьёву за совместную работу над алгоритмами сжатия и визуализации многомерных данных; С. А. Сукову и М. А. Корнилиной за совместную работу над алгоритмами динамической балансировки загрузки. Автор благодарен Б. М. Шабанову, О. С. Андрюшову, П. Н. Телегину и другим сотрудникам МСЦ РАН за [illegible] и консультационную поддержку, позволившую успешно выполнить значительную часть численных экспериментов. Автор выражает свою искреннюю благодарность своим близким за понимание и поддержку, А. А. Антоновой и И. И. Антоновой за помощь в оформлении материалов книги, Е. Д. Кормилицыной за конструктивную критику и большую помощь, оказанную при подготовке и оформлении рукописи.

Глава 2

Основные понятия

2.1. Параллельная программа как ансамбль взаимодействующих последовательных процессов

Определимся со способом описания параллельно выполняемых действий. Каждая рассматриваемая в рамках настоящего курса параллельная программа представима в виде множества слабосвязанных последовательных процессов [4, 5]. Каждый из процессов выполняет свою, независимую от других последовательность операций. Некоторые операции обеспечивают взаимодействие процессов — синхронизацию и обмен данными.

Конечно, такой подход существенно сужает класс рассматриваемых параллельных алгоритмов. В его рамках теряется возможность описания многих актуальных методов параллельного решения задач, например, алгоритмов управляемых потоками данных, нейросетевых алгоритмов, схем программируемой логики, векторных вычислительных устройств и многих других. Взамен, в нашем распоряжении оказывается возможность использовать относительно привычный способ описания алгоритмов. Более того, с практической точки зрения этот способ вполне отвечает архитектуре современных суперкомпьютеров. Их вычислительная мощность представлена множеством связанных между собой процессоров архитектуры Дж. фон Неймана, на которых и выполняются взаимодействующие процессы.

Итак, поскольку параллельная программа — это множество взаимодействующих последовательных процессов,

разработка параллельного алгоритма сводится к разработке набора последовательных алгоритмов, каждый из которых выполняется на своем процессоре. Наша цель — разработка *масштабируемых* алгоритмов, т.е. алгоритмов, эффективное выполнение которых возможно на произвольно большом числе процессоров. Из этого, конечно, не следует, что потребуется написать отдельную последовательную программу для каждого процессора. На практике все участвующие в обработке процессы логически делятся на несколько групп. На всех процессах одной группы выполняется один и тот же программный код. Например, в одной группе — только один процесс, обеспечивающий ввод исходных данных и распределение работы, а все остальные процессы относятся ко второй группе. Они обеспечивают обработку передаваемых им первым процессом заданий. В этом случае потребуется написать только два программных кода — для управляющего и обрабатывающего процесса, вне зависимости от того, сколько обрабатывающих процессов будет запущено.

2.2. Внутренний параллелизм

Не любая задача поддается распараллеливанию, даже если для нее известен вполне хороший последовательный алгоритм. Сегодня разработка параллельных программ является скорее искусством, чем технологией. Она предполагает поиски и создание методов, обладающих значительным внутренним параллелизмом. Алгоритм обладает *внутренним параллелизмом*, если в нем присутствуют действия, для которых допустимо одновременное выполнение. Это свойство именно алгоритма, оно не имеет отношения ни к типу вычислительной системы, ни к виду используемого языка программирования. Если на всех его этапах достаточно много взаимонезависимых операций, то есть вероятность успешного создания параллельного алгоритма. Если таких опе-

раций нет, то для решения задачи на многопроцессорной системе необходимо либо преобразовать алгоритм, либо искать другой метод решения.

Степень параллелизма алгоритма — это число операций, выполнять которые можно в любом порядке.

Иногда употребляют термин *степень внутреннего параллелизма*, подчеркивая тем самым, что имеют в виду именно внутренние свойства алгоритма, не зависящие от вида и особенностей вычислительных систем. На разных этапах выполнения алгоритма степень параллелизма может быть различной. Важно, чтобы высокой степенью внутреннего параллелизма обладали этапы, на выполнение которых тратится основное время.

Рассмотрим пример построения *масштабируемого* параллельного алгоритма решения задачи сложения многоразрядных чисел. Хороший последовательный алгоритм решения этой задачи внутренним параллелизмом практически не обладает, поэтому рассматриваемый параллельный алгоритм основан на менее быстродействующем последовательном.

2.2.1. Сложение многоразрядных чисел

Пусть даны два многоразрядных целых положительных десятичных числа a и b, каждое из которых представлено одномерным массивом десятичных цифр длины n. Требуется разработать параллельный алгоритм определения их суммы — многоразрядного целого числа c, представленного массивом длинны $n + 1$, $c = a + b$. Будем считать, что $n >> p$, где p — число процессоров, и потребуем, чтобы параллельный алгоритм обладал высокой масштабируемостью. Время решения задачи должно сокращаться с увеличением числа процессоров: если $p_2 > p_1$ и $n >> p_2$, то $T_{p_1} < T_{p_2}$, даже при самых неблагоприятных исходных данных.

Для определения суммы с помощью последовательного алгоритма А6 потребуется выполнить $4n$ операций сложения, деления и нахождения остатка, времени потребуется $T_1 = 4n\tau_c$, где τ_c — время выполнения одной арифметической операции.

Мы пренебрегли операциями, связанными с организацией цикла (увеличения счетчика цикла, сравнения i и n, перехода к началу цикла), что не влияет на качественную картину проводимого анализа. Для подробного ответа на вопрос: «Чему будет равно время выполнения программы, если учесть эти операции?», потребуются решить, как минимум, три задачи. Во-первых, более детально описать не алгоритм, а именно программу: то, как она выглядит в кодах конкретного процессора (например, так, как это сделано в монографии [6]). Во-вторых, детально описать архитектуру процессора (наличие и число уровней кэш-памяти, регистров, числа конвейеров, состав каждого конвейера и пр.). В-третьих, подробно рассмотреть логику работы планировщика инструкций процессора, дисциплину использования строк кэша, особенности механизма предвыборки данных и другие подобные вопросы. Изучение трех указанных задач далеко выходит за рамки настоящего курса. Более того, сильная зависимость результатов подобного анализа от архитектуры конкретного процессора делает весьма сомнительной их ценность с точки зрения предсказания свойств алгоритма в общем случае.

Дополнительные, не связанные непосредственно с требуемыми вычислениями операции, могут существенно влиять на время выполнения программы (в связи с чем, эффективность выполнения программы на современном процессоре вполне сравнима с КПД паровоза [7]). Несмотря на это, в дальнейшем мы ограничиваемся анализом только основных вычислительных операций, инвариантных относительно типа конкретного используемого процессора.

Алгоритм А6

Последовательный алгоритм сложения многоразрядных чисел

```
...
...
...
...
...
...
...
...
...
...
```

Сразу отметим, что степень параллелизма алгоритма А6 мала. При вычислении d она равна 1, поскольку две операции сложения выполняются по очереди. Величины $c[i]$ и r могут быть вычислены одновременно, следовательно, на этом этапе степень параллелизма равна 2. Одновременное выполнение витков указанного цикла невозможно, поскольку для вычисления d на витке $i + 1$ требуется значение r, вычисленное на витке i. Таким образом, нет оснований ожидать высоких показателей ускорения от параллельной версии этого алгоритма. Тем не менее рассмотрим возможный параллельный вариант как некоторое приближение к окончательной успешной версии.

Определим, каким образом исходные данные будут распределены по процессорам и какие операции над ними какой процессор будет над ними выполнять. Подчеркнем: «каким образом данные *размещены* по процессорам», а не «каким образом они *передаются* на процессоры». Предполагается, что в начале работы данные уже распределены по процессорам нужным образом, а по окончании работы результат также распределен по процессорам.

В пользу таких допущений есть ряд весомых аргументов:

- обеспечивается потенциальная возможность обработки многоразрядных чисел произвольно большой длины;
- существует множество задач, например, символьной арифметики, в которых многоразрядные числа являются результатами промежуточных вычислений. Полученный результат может быть использован в следующей операции, и нет никаких оснований перераспределять его фрагменты по процессорам;
- если все процессоры имеют доступ к общей оперативной памяти, в которой хранятся обрабатываемые числа, то физически передавать данные процессорам не требуется, достаточно указать каждому процессору начало и конец обрабатываемых им фрагментов;
- если у каждого процессора отдельная оперативная память, и она не содержит в начале работы требуемые данные, то требуется их физическое перемещение в оперативную память каждого процессора. На выполнение операций сложения цифр требуется значительно меньше времени, чем для их передачи между процессорами. Таким образом, если по сути решаемой задачи необходимо сначала именно перераспределить данные между процессорами — выполнить копирование данных в оперативную память, то, вполне возможно, что рассматриваемый далее метод не принесет ожидаемого выигрыша во времени решения задачи, и, соответственно, выбор следует сделать в пользу последовательного алгоритма.

Пусть в начале работы разряды первого и второго чисел распределены по процессорам равными непрерывными фрагментами размера $n_1 = n/p$ (в предположении, что n/p — целое число). На процессоре с номером $rank$ размещены разряды числа с номерами от $rank \cdot n_1$ до $(rank + 1) \cdot n_1 - 1$ включительно. Тогда можно предложить следующий параллельный алгоритм.

Алгоритм А6

Последовательный алгоритм сложения многоразрядных чисел

```
char a[n],b[n],c[n+1];
...
r=0;
for(i=0;i<=n;i++)
  {
  d=a[i]+b[i]+r;
  c[i]=d%10;
  r=d/10;
  }
c[i]=r;
```

Сразу отметим, что степень параллелизма алгоритма А6 мала. При вычислении *d* она равна 1, поскольку две операции сложения выполняются по очереди. Величины *c*[*i*] и *r* могут быть вычислены одновременно, следовательно, на этом этапе степень параллелизма равна 2. Одновременное выполнение витков указанного цикла невозможно, поскольку для вычисления *d* на витке $i + 1$ требуется значение *r*, вычисленное на витке *i*. Таким образом, нет оснований ожидать высоких показателей ускорения от параллельной версии этого алгоритма. Тем не менее рассмотрим возможный параллельный вариант как некоторое приближение к окончательной успешной версии.

Определим, каким образом исходные данные будут распределены по процессорам и какие операции над ними какой процессор будет над ними выполнять. Подчеркнем: «каким образом данные *размещены* по процессорам», а не «каким образом они *передаются* на процессоры». Предполагается, что в начале работы данные уже распределены по процессорам нужным образом, а по окончании работы результат также распределен по процессорам.

В пользу таких допущений есть ряд весомых аргументов:

- обеспечивается потенциальная возможность обработки многоразрядных чисел произвольно большой длины;
- существует множество задач, например, символьной арифметики, в которых многоразрядные числа являются результатами промежуточных вычислений. Полученный результат может быть использован в следующей операции, и нет никаких оснований перераспределять его фрагменты по процессорам;
- если все процессоры имеют доступ к общей оперативной памяти, в которой хранятся обрабатываемые числа, то физически передавать данные процессорам не требуется, достаточно указать каждому процессору начало и конец обрабатываемых им фрагментов;
- если у каждого процессора отдельная оперативная память, и она не содержит в начале работы требуемые данные, то требуется их физическое перемещение в оперативную память каждого процессора. На выполнение операций сложения цифр требуется значительно меньше времени, чем для их передачи между процессорами. Таким образом, если по сути решаемой задачи необходимо сначала именно перераспределить данные между процессорами — выполнить копирование данных в оперативную память, то, вполне возможно, что рассматриваемый далее метод не принесет ожидаемого выигрыша во времени решения задачи, и, соответственно, выбор следует сделать в пользу последовательного алгоритма.

Пусть в начале работы разряды первого и второго чисел распределены по процессорам равными непрерывными фрагментами размера $n_1 = n/p$ (в предположении, что n/p — целое число). На процессоре с номером *rank* размещены разряды числа с номерами от $rank \cdot n_1$ до $(rank + 1) \cdot n_1 - 1$ включительно. Тогда можно предложить следующий параллельный алгоритм.

Алгоритм А7

Параллельный алгоритм сложения многоразрядных чисел

```
char a[n1],b[n1],c[n1+1];
...
if(rank !=0) Recv(rank-1);
else r=0;

for(i=0;i<n1;i++)
  {
  d=a[i]+b[i]+r;
  c[i]=d%10;
  r=d/10;
  }
if(rank<p-1) Send(rank+1,r);
else   c[i]=r;
```

Здесь: *Recv*(*proc*, *data*) — функция, приема данных *data* от процесса с номером *proc*, *Send*(*proc*, *data*) — функция передачи данных *data* процессу с номером *proc*.

Оценим время выполнения алгоритма А7. Каждый из процессов выполняет две операции приема/передачи данных и $4n/p$ арифметических операций. Пусть на выполнение одной операции передачи данных требуется время τ_s, тогда время затраченное каждым процессом (кроме первого и последнего), составит $t_p = \frac{4n}{p}\tau_c + 2\tau_s$. Однако, эта оценка не учитывает, что сразу приступить к выполнению вычислений может только нулевой процесс. Остальные в начале работы ожидают приема данных от процессов с меньшими номерами. Несмотря на то, что непосредственно на передачу числа требуется время τ_s, операция приема данных в процессе с номером 1 будет выполнена только после того, как процесс 0 закончит выполнение вычислений и выполнит операцию передачи данных. Таким образом, процесс 1 будет находиться в состоянии ожидания в течение времени $\frac{4n}{p}\tau_c + 2\tau_s$, затем выполнит

вычисления и передаст данные процессору 2. Следовательно, процесс с номером 2 будет находиться в состоянии ожидания в течение времени $2\frac{4n}{p}\tau_c + 2\tau_s$. Аналогично рассуждая, получим, что общее время, требующееся для получения окончательного результата:

$$T_p = p\frac{4n}{p}\tau_c + (p-1)\tau_s = 4n\tau_c + (p-1)\tau_s.$$

Предложенный алгоритм работает медленнее последовательного: $T_p > T_1$. Несмотря на то, что в работе участвует несколько процессоров, параллельным такой алгоритм назвать сложно — все вычисления выполняются последовательно. Как уже упоминалось, причина в том, что структура исходного последовательного алгоритма непригодна для создания хорошего параллельного алгоритма, поскольку не содержит действий, выполнять которые можно одновременно на нескольких процессорах. Исходный алгоритм не обладает достаточным ресурсом *внутреннего параллелизма*. Для создания хорошего параллельного алгоритма требуется выбрать другой метод расчета.

Сейчас есть смысл отложить чтение и потратить некоторое время на поиск такого метода. Это позволит либо получить удовольствие от решения нетривиальной задачи, либо оценить дистанцию между исходным алгоритмом и алгоритмом, предлагаемым ниже по тексту. Фактически, надо придумать новый алгоритм, существенно отличающийся от исходного.

Поиск алгоритма, который позволит сократить время вычислений на p процессорах почти в p раз, успехом, вероятно, не увенчается. Тем не менее, можно предложить метод, для которого в достаточно широком диапазоне чисел процессоров справедливо соотношение $\frac{T_{p_1}}{p_1} \approx \frac{T_{p_2}}{p_2}$. Начиная уже с двух процессоров, увеличение числа процессоров вдвое позво-

лит вдвое сократить время решения задачи — алгоритм будет *масштабируемым*.

Причина последовательного характера вычислений в алгоритме A7 кроется в том, что каждый из процессов, кроме нулевого, в начале работы ожидает значение величины r (переноса) от процессора с меньшим номером. Однако чему может быть равно значение переноса? Возможно только два варианта: перенос равен либо 0, либо 1. Именно это соображение дает ключ к решению — построению *спекулятивного* параллельного алгоритма суммирования. Под «спекулятивным» понимается опережающее выполнение операций — *до* того, как подтверждена необходимость их выполнения. Как следствие, результат выполнения некоторых выполненных действий может остаться невостребованным.

Проиллюстрируем идею на следующем примере: пусть $a = 99999999$, $b = 1$, обозначив за $[a]$ фрагмент числа a, обрабатываемый одним процессом (таблица 2).

Таблица 2

Сложение длинных чисел

	Процесс 3	Процесс 2	Процесс 1	Процесс 0
[a]	99	99	99	99
[b]	0	0	0	1
[a] + [b]	99	99	99	100
[a] + [b] + 1	100	100	100	—
[c]	100	00	00	00

Пусть каждый процесс, не дожидаясь получения данных от процесса с меньшим номером, находит две частичные суммы: $[a] + [b]$ и $[a] + [b] + 1$. Тем самым, одновременно все процессы вычислят два ответа — и на случай получения от предыдущего процесса 0, и на случай получения 1. На этом шаге взаимодействие между процессами не требуется. Да-

лее следует установить, какой из этих ответов соответствует искомой сумме. Для этого потребуется передать некоторые данные между процессами. Сделать это можно разными способами. Ниже приведем не самый быстрый из возможных, но самый простой метод (существенно более быстрый алгоритм обсуждается в разделе 4.1.1). Передадим разряд переноса по цепочке между соседними процессами аналогично тому, как это было сделано в алгоритме А7: от процесса 0 к процессу 1, затем от процесса 1 к процессу 2 и так далее.

Алгоритм А8

Параллельный спекулятивный алгоритм сложения многоразрядных чисел

```
r1=0; r2=1;

for(i=0;i<n1;i++)
{
  d1=a[i]+b[i]+r1; c1[i]=d1%10; r1=d1/10;
  d2=a[i]+b[i]+r2; c2[i]=d2%10; r2=d2/10;
 }

if(rank>0) Recv(rank-1,r0)
else   r0=0;

if(r0) { c=c1; r=r1; }
else { c=c2; r=r2; }

if(rank<p-1) Send(rank+1,r);
else   c[n1]=r;
```

Оценим общее время выполнения получившегося алгоритма А8. Каждый из процессов выполнит $8n_1$ вычислительных операций — вдвое больше, чем в алгоритме А7, но все процессы будут на этом этапе работать одновременно. Общее время выполнения этого этапа составит именно $8n_1\tau_c = 8\frac{n}{p}\tau_c$. Оно будет тем меньше, чем больше процессоров примут участие в выполнении расчета. На втором

этапе, как и в алгоритме A7, на передачу данных будет затрачено время $(p-1)\tau_s$. Окончательно получим:

$$T_p = \frac{8n}{p}\tau_c + (p-1)\tau_s. \qquad (1)$$

Определим, во сколько раз параллельный алгоритм будет выполняться быстрее последовательного.

$$S_p = \frac{T_1}{T_p} = \frac{4nt_c}{\frac{8n}{p}\tau_c + (p-1)\tau_s}.$$

Пренебрегая затратами на взаимодействие процессов (полагая τ_s), получим, что ускорение $S_p = \frac{p}{2}$. Это означает, что на ста процессорах задача будет решена не в сто раз быстрее, чем на одном процессоре, а только в пятьдесят. Эффективность использования вычислительной мощности равна 50%. Последовательный алгоритм использует вычислительную мощность процессора на все сто процентов, но он решает задачу долго. Чтобы решить задачу быстро, потребуется выполнить в общей сложности вдвое больше операций, чем в последовательном алгоритме, это и обуславливает потерю эффективности.

Итак, основной вывод из рассмотренного примера: для сокращения времени выполнения задачи может потребоваться создать новый алгоритм, возможно, выполняющий больше операций, чем наилучший последовательный алгоритм. Потеря эффективности — плата за масштабируемость и сокращение времени выполнения вычислений.

2.3. Ускорение и эффективность параллельных алгоритмов

Ускорение и *эффективность* — две основные характеристики, используемые для количественной оценки свойств параллельных алгоритмов.

Ускорение параллельного алгоритма — это отношение времени выполнения последовательного алгоритма T_1 ко времени выполнения параллельного алгоритма T_p на заданном числе процессоров p.

$$S_p = \frac{T_1}{T_p}.$$

Данное определение следует признать недостаточно строгим. Что подразумевается под «временем выполнения алгоритма»? Не *программы*, а именно *алгоритма*?

На практике используются два способа оценки этого времени:

1) Определяется число выполняемых алгоритмом операций и полагается, что время выполнения пропорционально этому числу.
2) Замеряется время выполнения программы, реализующей алгоритм на некоторой вычислительной системе.

Оба предложенных способа следует признать не вполне удачными.

Отметим недостатки первого. Как правило, указываемые при описании алгоритма операции являются существенно более высокоуровневыми, чем команды процессора. Не всегда можно с достаточной точностью оценить время их выполнения, не рассмотрев предварительно фрагмент соответствующего ассемблерного кода. И даже если известна правильная оценка времени выполнения каждой из операций алгоритма, остается проблема оценки времени выполнения *последовательности* этих операций. Это время может быть как больше, так и меньше прямой суммы времен выполнения одиночных операций.

На процессорах, оснащенных кэш-памятью и конвейерами выполнения команд, время выполнения последовательности операций может быть существенно *меньше* суммы времен выполнения каждой операции по отдельности.

Например, за счет меньшего числа обращений к медленной оперативной памяти, совмещения во времени обработки разных фаз выполнения разных команд, спекулятивного выполнения инструкций или за счет ряда других факторов.

На вычислительной системе с общей памятью время параллельного выполнения двух независимых последовательностей команд на двух процессорах может быть существенно *больше* времени последовательного выполнения (сначала первой, потом второй) этих же последовательностей на одном процессоре. Например, за счет конфликтов кэш-памяти.

Таким образом, между *числом операций и временем их выполнения* нет детерминированной, априори известной простой связи. В общем случае это не позволяет использовать для оценки времени выполнения оценку числа выполняемых действий.

При втором способе — замере времени выполнения программы, алгоритм необходимо предварительно запрограммировать. В результате, речь уже будет идти не о времени работы *алгоритма*, а о времени работы *программы*. Время выполнения программы может существенно зависеть от параметров используемой вычислительной системы, например, от объема доступной оперативной памяти. Соответствующее влияние может быть сколь угодно велико, что ставит под сомнение правомерность отождествления свойств алгоритма и экспериментально установленных свойств программы. Пример такого влияния приведен при обсуждении пирамидальной сортировки (рис. 31).

Будем преимущественно оценивать сложность алгоритмов с помощью первого способа, основанного на подсчете числа операций. Дополнительно при разработке программ решения прикладных задач мы будем использовать и второй способ, основанный на измерении времени работы программ. В этом случае будем анализировать причины расхождения результатов, полученных первым и вторым способами, что позволит оптимизировать разрабатываемые методы

Эффективность параллельного алгоритма — это отношение ускорения к числу процессоров, на которых оно достигнуто:

$$E_p = \frac{S_p}{p}.$$

При одинаковом числе процессоров большая эффективность алгоритма обеспечивает решение задачи за меньшее время. Однако алгоритмы с низкой эффективностью могут представлять существенный интерес в том случае, если эта низкая эффективность сохраняется неизменной при достаточно большом числе процессоров. Характерным примером подобного алгоритма является метод нечетно-четного слияния Бэтчера (рис. 51, 52). Итоговое ускорение $S_p = pE_{(p)}$ может быть сколь угодно велико, если $E_{(p)} > E > 0$ при $p \to \infty$, где E — некоторая положительная константа. В связи с этим полезно ввести в рассмотрение еще одну характеристику параллельного алгоритма — его *масштабируемость*.

Масштабируемость параллельного алгоритма — это наименьшее число процессоров, на которых достигается максимальное ускорение.

$$M = \min p^* : \forall p, S_{p^*} \geq S_p$$

Чем лучше масштабируем алгоритм, тем для большего числа процессоров будет наблюдаться сокращение времени решения задачи. Максимальное ускорение, таким образом, равно S_M.

2.4. Ускорение и эффективность относительно наилучшего последовательного алгоритма

Ускорение — формальный показатель. Его значение существенно зависит от того, с каким именно последовательным алгоритмом выполняется сравнение. В общем случае

выбор такого алгоритма не является очевидным. Например, для многих прямых методов решения систем алгебраических уравнений не найдены эффективные параллельные аналоги, но параллельные алгоритмы для их решения успешно создаются на основе итерационных методов. Они уступают прямым методам, если речь идет о решении на одном процессоре, но выигрывают, за счет большей степени внутреннего параллелизма, на многопроцессорных системах. Если интересует, какой выигрыш даст использование многопроцессорной системы, сравнивать следует с самым быстрым из последовательных алгоритмов, поэтому актуальны дополнительные определения ускорения и эффективности.

Ускорение параллельного алгоритма по сравнению с наилучшим последовательным — это отношение времени решения задачи *наилучшим последовательным алгоритмом* ко времени выполнения параллельного алгоритма на системе из p процессоров:

$$S_p^* = \frac{T_1^*}{T_p}.$$

Недостаток данного определения — в его неконструктивности. Что такое «наилучший» последовательный алгоритм? Как можно быть уверенным в том, что не существует алгоритма выполняющегося быстрее, чем тот, который объявлен наилучшим? В дальнейшем, не оговаривая это дополнительно, будем считать «наилучшим» последовательный алгоритм, затрачивающий на решение задачи наименьшее время среди всех имеющихся в нашем распоряжении алгоритмов.

Эффективность параллельного алгоритма по отношению к наилучшему последовательному — это отношение его ускорения по сравнению с наилучшим последовательным алгоритмом к числу процессоров, на которых это ускорение получено.

$$E_p^* = \frac{S_p^*}{p}.$$

Сверхлинейное ускорение — это ускорение, превышающее число используемых для расчета процессов: $S_p^* > p$. Можно предложить как минимум две причины, по которым ускорение может быть существенно больше числа процессоров. Одна из них обусловлена особенностями используемых вычислительных систем, другая — неудачным выбором последовательного алгоритма.

2.4.1. Неравноправность условий выполнения — первая причина сверхлинейного ускорения

Работая с вычислительной системой вообще и с параллельной вычислительной системой в частности не следует забывать, что они представляют собой объекты, обладающие нетривиальными свойствами. Необходимо учитывать эти свойства при оценке производительности систем.

Рассмотрим в качестве примера задачу обработки последовательно поступающих поисковых запросов к базе данных. Пусть размер базы десятикратно превышает объем оперативной памяти одного процессорного узла. В простейшем случае при обработке запросов одним процессором потребуется при каждом запросе просмотреть всю базу, каждый раз читая ее записи с медленного внешнего носителя данных. Основное время работы будет определяться именно многократным чтением данных. При использовании двух процессоров можно разделить базу данных на две части. Тогда каждый из процессоров сможет выполнять поиск в половине записей, что приведет к сокращению времени работы примерно в два раза. До тех пор, пока объем данных, обрабатываемых каждым из процессоров, превышает объем доступной ему оперативной памяти, рост ускорения будет линейным, поскольку время работы каждого процессора будет определяться временем чтения данных с внешнего но-

сителя. Ситуация принципиально изменится, как только совокупный объем оперативной памяти всех процессоров превысит размер базы данных. В этом случае после однократного чтения каждым процессором своей части базы, обращений к медленным внешним устройствам больше не потребуется. Весь поиск процессоры смогу выполнить, используя данные, хранящиеся в их быстрой оперативной памяти. Каждый процессор теперь обрабатывает больше запросов в единицу времени, чем обрабатывал при чтении данных с внешнего устройства. Фактически, система теперь представлена более быстрыми процессорами, что и приводит к сверхлинейному ускорению. Дальнейшее увеличение числа процессоров приводит к размещению всех данных уже не в оперативной памяти, а в быстрой кэш-памяти процессоров, что обуславливает второй скачек сверхлинейного ускорения.

2.4.2. Алгоритмическая причина сверхлинейного ускорения

Рассмотрим и вторую, принципиально отличающуюся возможную причину возникновения эффекта сверхлинейного ускорения. Она никак не связана с техническими особенностями вычислительных систем, а обусловлена исключительно внутренними свойствами используемых алгоритмов.

Сверхлинейное ускорение может проявляться в случае сравнения параллельного алгоритма с неудачным последовательным. Один из подобных примеров уже упоминался в разделе 1.4.5. Рассмотрим еще один наглядный пример параллельного алгоритма, ускорение которого пропорционально не первой, а второй степени числа используемых процессоров.

Пусть для упорядочивания массива a из n элементов используется метод простой вставки [9]. Общее число операций сравнения/копирования элементов, выполняемых согласно этому алгоритму, равно $T_1(n) = \frac{n(n-1)}{2}$.

Алгоритм A9

Последовательный алгоритм сортировки

```
for(i=n-1;i>0;i--)
for(j=0;j<i-1;j++)
  if(a[j]>a[j+1])
    swap(a[j],a[j+1])  // поменять местами элементы
                       // a[j] и a[j+1]
```

Разделим массив на две равные части и упорядочим каждую из частей с помощью отдельного процессора. Таким образом, при использовании двух процессоров, на сортировку половинок массива будет затрачено время, необходимое для выполнения $\frac{n(n-2)}{8}$ операций. На объединение упорядоченных половинок массива одним из процессоров потребуется n операций сравнения/копирования элементов (алгоритм A10).

Алгоритм A10

Параллельный алгоритм сортировки

```
// параллельное упорядочивание половин массива
// на двух процессорах
```

процессор 1	процессор 2

for(i=n-1;i>n/2;i--)
for(j=n/2;j<i-1;j++)
  if(a[j]>a[j+1])
  swap(a[j],a[j+1])
	for(i=n/2-1;i>0;i--)
	for(j=0;j<i-1;j++)
	  if(a[j]>a[j+1])
	  swap(a[j],a[j+1])

```
// последовательное слияние упорядоченных
// половинок одним из процессоров
i=0
k=n/2

for(i=0;i<n;i++)
{
if(i==n/2)  a[i]=a[k++]
  else
```

```
if(k==m)  a[i]=a[j++]
else
if(a[j]>a[k])  a[i]=a[k++]
else  a[i]=a[j++]
}
```

Таким образом, общее число тактов при использовании двух процессоров равно $T_2(n)=\frac{n(n-2)}{8}+n$. При сортировке половинок массива на каждом такте выполняются две операции, а при их слиянии — одна. В случае последовательного алгоритма на каждом такте выполняется одна операция. Считая, что время выполнения пропорционально числу тактов, определим достигнутое ускорение как отношение этих времен: $S_2=\frac{T_1(n)}{T_2(n)}\approx 4>2$. В общем случае при $p \gg n$, $T_p(n)=\frac{1}{2}\frac{n}{p}\left(\frac{n}{p}-1\right)+T_{\text{слияния}}=\frac{1}{2}\frac{n^2}{p^2}$, что и приводит к ускорению $S_p \approx p^2 > p$.

Причина сверхлинейного ускорения заключается в том, что в качестве последовательного выбран медленный алгоритм, имеющий оценку числа выполняемых операций $O(n^2)$. В результате, при разбиении массива на p частей, время обработки каждой части падает в p^2 раз. Следовало бы выбрать более быстрый последовательный алгоритм. В случае сортировки данных такие алгоритмы хорошо известны (некоторые из них обсуждаются в главе 5, посвященной параллельной сортировке данных), но как поступить, если параллельный алгоритм, демонстрирующий сверхлинейное ускорение, есть, а лучший последовательный метод неочевиден? Очевидно, что параллельный алгоритм уже содержит *все* необходимые действия, последовательное выполнение которых будет соответствовать некоторому приближению к наилучшему последовательному алгоритму. Рассмотрим

формальный метод его построения и покажем, что относительно него эффект сверхлинейного ускорения не возникнет. С одной стороны, это позволит при необходимости рассматривать последовательный алгоритм как частный случай параллельного, а с другой — даст полезный способ формального порождения «наилучшего» последовательного алгоритма.

2.4.3. Формальное преобразование параллельного алгоритма в «наилучший» последовательный

Пусть параллельный алгоритм предполагает выполнение на двух процессорах. Тогда, согласно принятому в курсе подходу, его выполнение сводится к выполнению двух слабо взаимодействующих последовательных процессов. Преобразуем параллельный алгоритм в последовательный:

Шаг 1. Выберем для выполнения первый процесс. Выполним все действия, предшествующие операции взаимодействия между процессами.

Шаг 2. Выберем для выполнения второй процесс. Выполним все действия, предшествующие операции взаимодействия между процессами.

Шаг 3. Выполним операции, соответствующие требуемому взаимодействию. Например, если предполагалась передача сообщения, установим значение принимаемой переменной равным значению передаваемой переменной.

Шаг 4. Перейдем к Шагу 1.

Выполнение шагов 1—4, вплоть до завершения обоих процессов, гарантирует, что все операции, выполнение которых было предусмотрено на двух процессорах, будут выполнены на одном процессоре в *конкурентном* режиме. Таким образом, в качестве наилучшего последовательного алгоритма можно взять параллельный алгоритм, но выполненный указанным способом. Аналогичным образом можно поступить

при преобразовании алгоритма, выполняющегося на любом конечном числе процессоров.

Вернемся к примеру сортировки массива методом простой вставки на двух процессорах (алгоритм A10) и преобразуем его к последовательному виду (алгоритм A11). Поскольку в явном виде операции передачи данных в параллельном алгоритме не упоминались, никаких дополнительных операций по эмуляции взаимодействия не потребуется.

Алгоритм A11

Последовательный алгоритм сортировки

```
// выполнить операции процессора 1
// выполнить операции процессора 2
// выполнить слияние упорядоченных половин массива
```

При сортировке с помощью этого алгоритма выполняется $T_2^*(n) = \frac{n(n-2)}{4} + n$ операций, что соответствует ускорению $S_2^*(n) = 2\frac{n-1}{n+2} < 2$, не превышающему числа используемых процессоров.

Рассмотрим эффективность произвольного параллельного алгоритма относительно его последовательной версии, построенной с помощью шагов 1—4. Пусть t_i — время выполнения программы процессором с номером i. Оно учитывает как время выполнения вычислительных операций, так и время, затраченное на взаимодействие процессов и на взаимное ожидание процессов, обусловленное возможной несбалансированностью вычислений.

$$T_p = \max_i t_i.$$

Тогда, предполагая, что на одном процессоре операции эмулирующие взаимодействие процессоров занимают не больше времени, чем само взаимодействие, получим следующую оценку времени работы последовательного алгоритма:

$$T' = \sum_{i=1}^{p} t_i$$

Полагая, без ограничения общности, что нулевой процесс работает дольше остальных: $\max_i t_i = t_0$, получим выражение для ускорения:

$$S' = \frac{\sum_{i=1}^{p} t_i}{t_0} \le \frac{pt_0}{t_0} = p,$$

из которого следует, что всегда можно формально сконструировать последовательный алгоритм, относительно которого эффект сверхлинейного ускорения наблюдаться не будет: ускорение не превысит общего числа используемых процессоров.

Подчеркнем, что сделанный вывод справедлив только тогда, когда сверхлинейное ускорение вызвано *неудачным выбором последовательного алгоритма*, а не *влиянием неравноправных условий выполнения* последовательного и параллельного алгоритмов.

2.5. Априорная оценка эффективности параллельного алгоритма

Эффективность параллельного алгоритма непосредственно зависит от того, насколько равномерно процессоры загружены работой. Рассмотренный выше пример (алгоритм A10) показывает, что на разных этапах работы может быть использовано разное число процессоров. На первом этапе две части массива обрабатывались двумя процессорами, но на втором этапе слияние выполнялось только одним процессором. Как правило, параллельный алгоритм на разных этапах обладает разной степенью внутреннего параллелизма. Рассмотрим полезную модель, позволяющую предварительно оценить максимально достижимую величину ускорения параллельного алгоритма.

Пусть T_1 — время выполнения алгоритма на одном процессоре. Обозначим через α долю операций алгоритма, выполнение которых невозможно одновременно с другими операциями — иными словами, долю операций, выполняемых со степенью параллелизма 1. Тогда время, требуемое для их выполнения, будет равно αT_1, независимо от числа используемых процессоров. Пусть остальные операции (их доля равна $1 - \alpha$) могут выполняться с произвольно большой степенью параллелизма. Тогда время, необходимое для их выполнения на системе из p процессоров, будет равно $\frac{(1-\alpha)T_1}{p}$. Учтем наличие в параллельном алгоритме дополнительных по отношению к последовательному алгоритму действий. Например, в параллельном алгоритме могут присутствовать операции синхронизации процессов, передачи данных между процессами — операции, отсутствующие в последовательном алгоритме. Будем считать, что они увеличивают время выполнения программы на величину, равную t_d. Тогда ускорение, достигаемое на p процессорах:

$$S_p = \frac{T_1}{\left(\alpha + \frac{1-\alpha}{p}\right)T_1 + t_d}.$$

Рассмотрим три характерных случая:

1) $\alpha = 0$, $t_d = 0$, $S_p = p$.

Идеальный случай. Все операции выполняются с максимальной степенью параллелизма, дополнительные действия отсутствуют, $S_p = p$.

2) $t_d > T_1$, $S_p < 1$.

Время дополнительных операций превышает время решения задачи на одном процессоре. Решение задачи на многопроцессорной системе с помощью рассматриваемого параллельного алгоритма лишено смысла, поскольку $T_p > T_1$. Возможно, выбран неудачный параллельный алгоритм

Возможно, рассматривается очень простая задача, решение которой требует малого времени. При непропорционально больших затратах на синхронизацию и обмен данными между процессами, вместо многопроцессорной системы целесообразнее использовать один единственный процессор.

3) $\alpha \neq 0,\ t_d = 0,\ S_p = \dfrac{1}{\alpha + \dfrac{1-\alpha}{p}}$.

Последнее из полученных соотношений выражает закон Амдаля или Уэра (Gene Amdahl, Ware). Оно записано в предположении, что все действия выполняются либо с минимальной, либо с максимальной степенью параллелизма, а потери, обусловленные дополнительными операциями, отсутствуют.

Рассмотрим поведение ускорения в зависимости от значения α при использовании 100-процессорной вычислительной системы (рис. 4). График показывает быстрый спад ускорения при малых значениях доли последовательных операций. Помеченная на графике точка соответствует $\alpha = 0{,}01$, ускорение при этом оказывается равным 50. Всего 1% операций, не подлежащих распараллеливанию, вдвое снижает эффективность выполнения алгоритма на 100-процессорной системе. Закон утверждает, что максимально

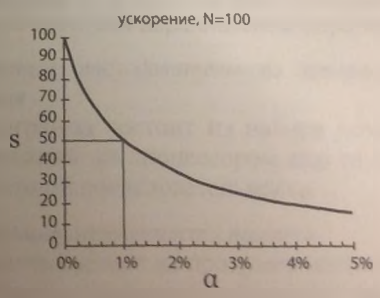


Рис. 4. Закон Амдаля

достижимое ускорение, независимо от числа используемых процессоров, ограничено величиной α^{-1}:

$$S_p < \frac{1}{\alpha}.$$

Разработка параллельных программ — достаточно затратная область деятельности. Прежде чем приступать непосредственно к программированию, следует иметь основания считать, что результат окупит затраченные усилия. Закон Амдаля как раз относится к числу способов предварительной оценки качества ожидаемого результата, позволяющих отбросить параллельные алгоритмы, заведомо не отвечающих требуемым характеристикам ускорения и эффективности.

Глава 3

Модели параллельных программ

Выделим те классы вычислительных систем, на которые ориентированы обсуждаемые далее параллельные методы. Будем рассматривать системы, вычислительная мощность которых образована множеством последовательных процессоров архитектуры фон Неймана. В первую очередь, это два больших класса, различаемых по принципу организации доступа процессоров к оперативной памяти: системы с общей и с распределенной оперативной памятью. Несмотря на то, что такой выбор охватывает далеко не все существующие параллельные архитектуры (обзор перспективных параллельных архитектур можно найти в монографии [8]), избранный круг представляет значительную практическую ценность. Например, системы списка пятисот крупнейших вычислительных систем мира [1] основаны на рассматриваемых архитектурах.

Прежде чем конкретизировать виды рассматриваемых систем, напомним те принципы фон Неймана, которые *не* выполняются в системах параллельной обработки данных:

ФН_1. Принцип последовательного программного управления
Программа состоит из набора команд, которые выполняются процессором друг за другом в определенной последовательности

ФН_2. Принцип адресуемости памяти
Память состоит из пронумерованных ячеек. Процессору в произвольный момент времени доступна любая ячейка памяти

3.1. Вычислительные системы с распределенной памятью

Масштабируемые системы массового параллелизма конструируют на основе объединения процессорных узлов каналами передачи данных. В каждом узле есть процессор, доступная ему оперативная память и сетевой адаптер передачи данных. Оперативная память вычислительного узла недоступна процессорам из других процессорных узлов (рис. 5). Взаимодействие процессов, выполняющихся на разных вычислительных узлах, возможно лишь путем передачи данных через каналы коммуникационной сети, связывающей вычислительные узлы в единый вычислительный ресурс. Сеть позволяет передать данные между любыми двумя процессорами. Как правило, будем предполагать, что время, необходимое для передачи сообщения между двумя процессорами, не зависит от того, насколько интенсивно используют сеть другие процессоры.

В системах этого класса не выполняются оба указанных принципа фон Неймана.

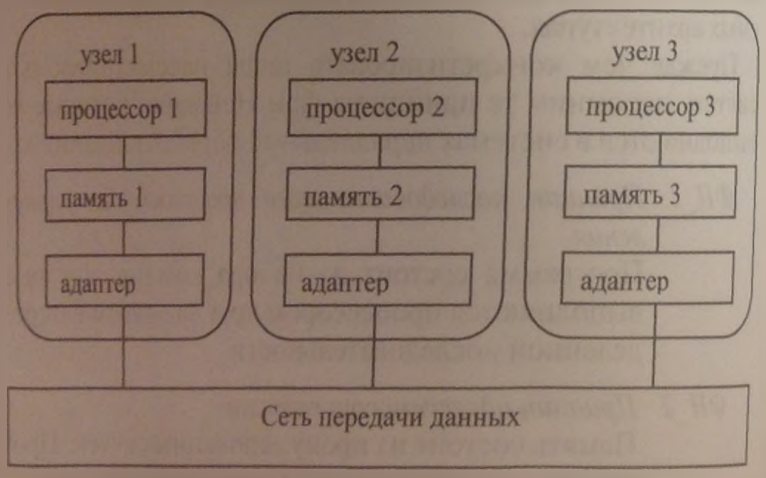


Рис. 5. Многопроцессорная система с распределенной памятью

Принцип последовательного программного управления нарушается, поскольку взаимный порядок выполнения операций процессами, выполняющимися на разных вычислительных узлах, не определен. Невозможно заранее так пронумеровать команды, выполняемые двумя процессами, чтобы нумерация совпала с порядком их действительного выполнения. Полное совпадение может быть только случайным. Последовательности выполнения команд будут изменяться от запуска к запуску. Более того, в системе с распределенной памятью может не существовать единого для всех узлов времени, которое можно было бы использовать для подтверждения того, что команды действительно выполнялись в некотором предопределенном порядке. Взаимный порядок выполнения двух команд может быть достоверно установлен только в том случае, если *между* выполнением этих команд было осуществлено взаимодействие двух процессов, например, путем передачи сообщения. Например, можно утверждать, что команда, выполненная первым процессом *до* отправки сообщения первым процессом, действительно выполнена раньше команды, выполненной вторым процессом *после* приема сообщения вторым процессом.

Принцип адресуемости памяти нарушается в силу того, что процессу, запущенному на некотором узле, недоступны ячейки памяти, расположенной на остальных узлах.

Отметим преимущества систем с распределенной памятью:

- Низкая стоимость — высокое значение отношения цены и к пиковой, и к реально достижимой производительности;
- Высокая масштабируемость — возможность построения систем практически неограниченной производительности;
- Возможность наращивания вычислительной мощности уже функционирующей системы путем установки дополнительных вычислительных узлов.

Сегодня системы с распределенной памятью — безусловные лидеры по пиковой производительности. К их недостаткам можно отнести сложность разработки эффективных параллельных алгоритмов.

3.2. Вычислительные системы с общей памятью

Второй класс систем — вычислительные системы с общей памятью.

В таких системах несколько процессоров имеют симметричный доступ к оперативной памяти (рис. 6).

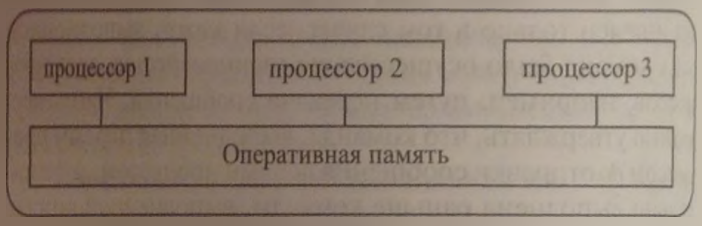


Рис. 6. UMA-модель системы с общей памятью

В полном соответствии со вторым принципом фон Неймана, все ячейки оперативной памяти пронумерованы, и каждый процессор имеет равный доступ к любой из них. Такая модель называется UMA (uniform memory access)-моделью и является сильно идеализированной. В отличие от UMA-модели, при большом числе процессоров время доступа конкретного процессора к конкретной ячейке оперативной памяти зависит от значений номеров процессора и ячейки. В идеальной модели все эти времена должны совпадать между собой, но на практике при большом числе процессоров они не вполне совпадают.

Более реалистичная схема системы с общей памятью представлена на рисунке 7. Оперативная память разде-

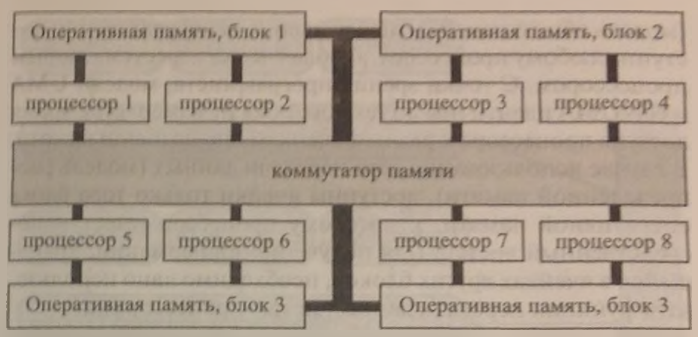


Рис. 7. NUMA, ccNUMA модели системы с общей памятью

на на независимые блоки. Каждый процессор имеет непосредственный доступ не ко всем блокам памяти, а только к одному из них. Например, первый и второй процессоры имеют непосредственный и равный доступ к первому блоку оперативной памяти. Время обращения первого процессора к ячейкам первого блока памяти меньше, чем к ячейкам, расположенных в других блоках. Обращение к блокам 2, 3 и 4 обеспечивается коммутатором оперативной памяти, что вносит дополнительную задержку. Такая модель, в зависимости от деталей реализации, получила название NUMA (неоднородный доступ к оперативной памяти) или ccNUMA (cache coherent non-uniform memory access, неоднородный кэш-когерентный доступ к оперативной памяти) [9].

Системы с общей памятью при равной пиковой производительности существенно дороже, чем системы с распределенной памятью. Они значительно хуже масштабируются. Тем не менее, известны ccNUMA-системы Altix UV 1000, масштабируемые до 2048 ядер.

Подчеркнем еще раз, что между «коммутатором оперативной памяти» и «сетью передачи данных» есть принципиальная разница. В случае использования коммутатора

(модель общей памяти), любая ячейка непосредственно доступна любому процессору. Любая ячейка адресуема любым процессором. С точки зрения программиста, модели UMA и ccNUMA идентичны до тех пор, пока не измеряется время доступа процессора к разным ячейкам оперативной памяти. В случае использования сети передачи данных (модель распределенной памяти), доступны ячейки только того блока оперативной памяти, к которому процессор имеет непосредственный доступ. Для получения информации, хранящейся в ячейках других блоков, необходимо явно использовать функции передачи данных между процессорами.

3.3. Гибридные архитектуры

Сегодня фактически нет систем, в точности соответствующих архитектуре распределенной памяти. Современный суперкомпьютер объединяет в себе большое количество узлов, в пределах каждого из которых процессоры имеют доступ к общей оперативной памяти (рис. 8). Как правило, каждый такой узел является полноценной UMA или ccNUMA-системой. Чтобы в полной мере использовать вычислительную мощность подобной системы, следует совместно использовать технологии работы и на общей, и на распределенной памяти.

За рамками рассмотрения остались многие не менее важные виды многопроцессорных и распределенных вычислительных систем, в том числе векторные, реконфигурируемые, GRID-системы и другие. В связи с этим отметим вычислительные системы, основная производительность которых обеспечена графическими ускорителями общего назначения (GPGPU). Каждый из ускорителей содержит большое число мультитредовых устройств, поддерживающих векторную обработку данных. Их высокое быстродействие во многом определяется наличием расширенного набора уровней оперативной памяти с разными временами

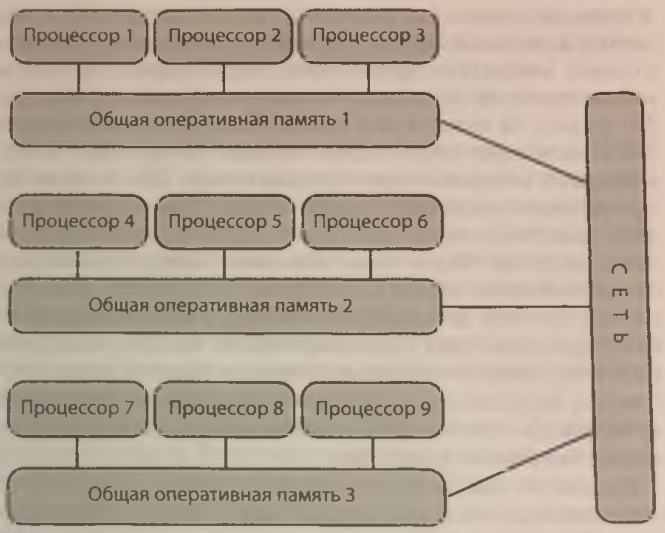


Рис. 8. NUMA, ccNUMA-модели системы с общей памятью

и правилами доступа. Эффективное использование таких систем требует создания не только новых параллельных алгоритмов, но и новых способов представления параллельных алгоритмов, однако сегодня и для этих систем программы пишутся в процедурном стиле, вполне согласующимся с принятым в настоящем курсе подходом.

3.4. Модель выполнения параллельной программы на распределенной памяти

Будем полагать, что выполнение параллельной программы на распределенной памяти начинается с запуска заданного числа одинаковых процессов. Соответственно,

в команде запуска указывается название программы и желаемое количество процессоров. Программе выделяется некоторое множество процессоров, число которых в общем случае может не совпадать с числом запущенных процессов. Например, на каждом узле вычислительной системы может быть несколько процессоров, каждый из которых может содержать несколько процессорных ядер. При запросе 10 процессоров могут быть предоставлены 2 вычислительных узла, каждый из которых оснащен двумя четырехядерными процессорами. Более того, возможен запуск нескольких процессов ни на одном одноядерном процессоре. В любом случае, с точки зрения программиста, у каждого запущенного процесса будет своя оперативная память, к которой не имеют доступа остальные процессы. Каждая копия программы запускается на отдельном *виртуальном процессоре*, не имеющем с логической точки зрения отличий от *физического одноядерного процессора*.

Каждая из запущенных копий программы получает от операционной системы 2 переменных. Первая — число запущенных процессов. Вторая — номер запущенного процесса. С помощью этой нумерации процессы могут обмениваться данными между собой. Подробнее средства взаимодействия программ на общей памяти будут рассмотрены в следующих разделах.

3.5. Модель выполнения параллельной программы на общей памяти

Определим *изолированный последовательный процесс* как последовательность действий, выполняемых автономно (независимо от окружающей обстановки).

Будем считать, что, кроме явных, достаточно редких моментов связи, процессы выполняются совершенно независимо друг от друга. Кроме того, будем считать, что такие про-

цессы связаны слабо, подразумевая под этим, что, кроме некоторых моментов явной связи, эти процессы рассматриваются как совершенно независимые друг от друга.

Будем называть такие процессы *слабо связанными последовательными процессами*.

Под *параллельной программой* будем понимать всю совокупность слабо связанных последовательных процессов, необходимых для выполнения требуемых вычислений. В однопроцессорной системе *программе*, как правило, соответствовал один выполняемый процесс — программный модуль, запускаемый под управлением операционной системы на ее единственном процессоре. Учитывая, что параллельная программа выполняется на наборе процессоров, ей соответствует множество *процессов*, запущенных на одном или на нескольких процессорах.

Примем следующую модель параллельной программы на общей памяти. Выполнение начинается с запуска единственного последовательного процесса. В тот момент, когда появляется возможность параллельного выполнения команд, вызывается функция порождения дополнительной нити кода (треда, легковесного процесса). Таким образом, появляется возможность параллельного выполнения команд. Число порождаемых нитей определяется требованиями алгоритма и числом доступных процессоров. В общем случае оно вполне может превышать число доступных процессоров, в этом случае нити будут выполняться не в параллельном, а в конкурентном режиме (режиме разделения времени). У каждой нити есть недоступные другим нитям локальные переменные, хранимые в стеке нити. Кроме локальных переменных, каждая из нитей имеет доступ к общим глобальным переменным. Они находятся в общем адресном пространстве. Каждая нить может читать и модифицировать глобальные данные. Подробнее средства взаимодействия программ на общей памяти будут рассмотрены в следующих разделах.

3.6. Средства взаимодействия последовательных процессов

3.6.1. Свойства канала передачи данных

Для взаимодействия программ, запущенных на системах с распределенной памятью, необходимо воспользоваться функциями передачи данных между процессорами, между копиями запущенной программы. Эти функции могут быть самыми разными, реализованы в различных средах, например, использовать быстрые или медленные сети, но независимо от этого, есть нечто общее, позволяющее ввести некоторые характеристики, отвечающие за время передачи данных.

К таким характеристикам относятся *пропускная способность сети*, которая говорит о том, сколько времени затрачивается на передачу 1 байта между двумя процессорами, и *латентность* — время, которое требуется, чтобы передача данных начала осуществляться. Таким образом, время, требуемое для передачи данных между двумя процессорами, определяется как сумма времени передачи одного байта (при условии, что передача уже началась), умноженного на количество передаваемых байт, и времени инициализации самой передачи данных (латентности):

$$T(n) = n * T_{\text{байта}} + T_{\text{латентности}}.$$

Графики зависимости времени передачи данных от количества передаваемых байт (рис. 9), показывают, что кривые на них достаточно пологие, время растет достаточно медленно, при этом начальный интервал задержки (латентности) является значительным. Например, для сети *Gbit Ethernet* латентность составляет около 50 мкс. Это означает, что на передачу 1 байта и 100 байт будет затрачено примерно одинаковое время. По возможности, следует заменить множество передач коротких сообщений передачей одного большого

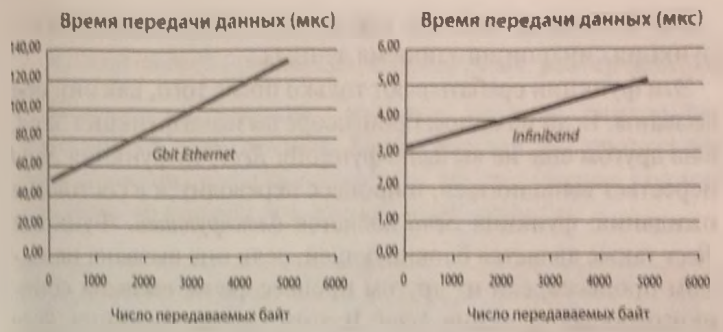


Рис. 9. Зависимость времени передачи данных от размера сообщения

сообщения. Например, при пересылке массива поэлементно, потребуется большое количество времени, определяемое латентностью: фактически, время передачи будет равно произведению количества элементов на время латентности (50 мкс). Если же получится передавать данные единым блоком, то латентность в этом процессе будет участвовать только один раз при инициализации посылки, а затем данные будут передаваться с высокой скоростью.

Вывод: следует стремиться к тому, чтобы данные передавались большими блоками.

3.6.2. Методы передачи данных

Кратко рассмотрим два основных метода передачи данных: синхронный и асинхронный.

Синхронный метод предполагает, что при передаче данных указывается адрес передаваемых данных, их размер (число байт или элементов) и номер процессора, с которым осуществляется взаимодействие.

Синхронные функции:

Send (номер процессора, адрес данных, размер данных) — функция синхронной передачи данных:

Recv (номер процессора, адрес данных, размер данных) — функция синхронного приема данных.

Эти функции срабатывают только после того, как они обе вызваны. Если на одном процессоре вызвана функция *Send*, а на другом еще не вызвана функция *Recv*, то функция *Send* перестает выполняться, и процесс переводится в состояние ожидания: функция *Send* является *блокирующей*. Функция *Recv* также является блокирующей, если она вызвана на одном процессоре, а на другом процессоре не вызвана соответствующая функция *Send*. В этом случае, функция *Recv* переходит в режим ожидания, пока не начнется передача данных.

Это базовые функции передачи данных, к которым сводятся остальные. Они затрачивают на передачу данных наименьшее время, но только при условии, что одновременно вызваны и передающая, и принимающая функции. В противном случае один из процессов оказывается в состоянии непроизводительного ожидания — простоя.

Для сокращения подобных непроизводительных потерь используются *асинхронные методы* передачи данных. Они отличаются от синхронных методов тем, что практически сразу возвращают управление вызвавшей их программе. Асинхронная функция передачи/приема данных не выполняет передачу/прием данных, а размещает заказ на эту операцию. Фактически, в этот момент порождается дополнительная нить программного кода, в рамках которой и осуществляется передача данных с помощью синхронных функций. Асинхронные методы делятся на *небуферизованные* и *буферизованные* методы, которые будут подробнее рассмотрены далее.

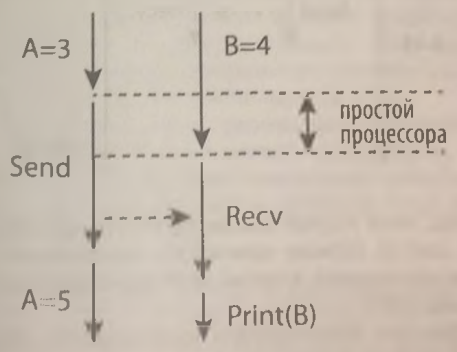
Небуферизованные функции:
ASend (номер процессора, адрес данных, размер данных);
ARecv (номер процессора, адрес данных, размер данных);
ASync (номер процессора).

Буферизованные функции:
ABSend (номер процессора, адрес данных, размер данных).

Разберем на примере основное отличие синхронных функций от асинхронных. Пусть имеется два процессора, выполняющих следующие наборы команд (первая колонка для первого процессора, вторая — для второго):

Процесс 1	Процесс 2
A=3 Send(2,&A) A=5	B=4 Recv(1,&B) Print(B)

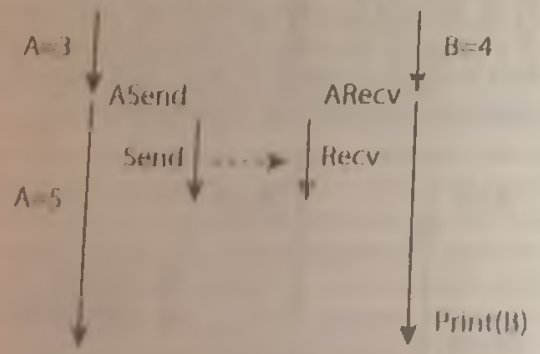
Первый процессор присваивает переменной *A* значение 3 и передает ее. Второй процессор получает некоторую переменную *B*, которая была равна 4. Так как функции синхронные, то *Recv* второго процессора ждет *Send* первого, и в результате выполнения будет напечатано число 3.



В случае использования асинхронных функций ситуация изменяется.

Процесс 1	Процесс 2
A=3 ASend(2,&A) A=5	B=4 ARecv(2,&B) Print(B)

Для, казалось бы, последовательного фрагмента кода первого процессора возникает новая нить, в которой выполняется функция *ASend*. При передаче данных указывается их адрес, т.е. то, с какого места в оперативной памяти они находятся. Аналогично для второго процессора, *ARecv* не выполняет прием, а размещает заказ на прием данных. Порождается дополнительная нить кода, в которой выполняется функция приема данных. Соответственно, результат *Print* может быть разным в зависимости от стечения обстоятельств: 3, 4 или 5.



В случае, если второй процессор успеет принять данные до того, как на первом произойдет присваивание нового значения переменной *A*, команда *Print* действительно напечатает значение 3.

Если функция *ARecv* вызвана после операции присваивания $A = 5$, то именно значение 5 и будет напечатано, передавшие данные и застынут. Это происходит в связи с тем, что на каждом процессоре выполняются две нити, одна выпол-

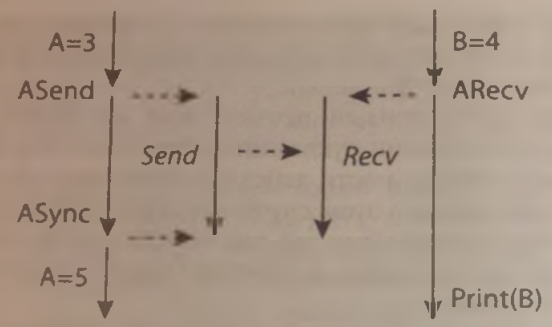
няет основные инструкции, а другая — передачу или прием данных. Взаимный порядок выполнения инструкций этими нитями заранее не определен.

Предположим, первый процесс еще не начал работу, а второй уже выполнил указанный фрагмент. В результате такой последовательности действий будет напечатано значение 4, поскольку в этом случае функция *ARecv* размещает заказ на прием данных, но еще не успевает их получить, а *Print* уже выполнилась, напечатав старое значение переменной *B* (это 4).

Таким образом, асинхронные функции позволяют, с одной стороны, оптимизировать работу системы, т.к. появляется возможность одновременно производить вычисления и передавать данные. С другой стороны, накладывают дополнительные ограничения на последовательность выполняемых действий. Ответственность за корректное использование асинхронных функций лежит на программисте, ему требуется приложить дополнительные усилия, чтобы получить прогнозируемый результат. Следует использовать дополнительную блокирующую *функцию синхронизации* — *ASync*. Напомним, что функции *ASend* и *ARecv* блокирующими не являются, для ожидания выполнения собственно передачи данных как раз и используется *ASync*.

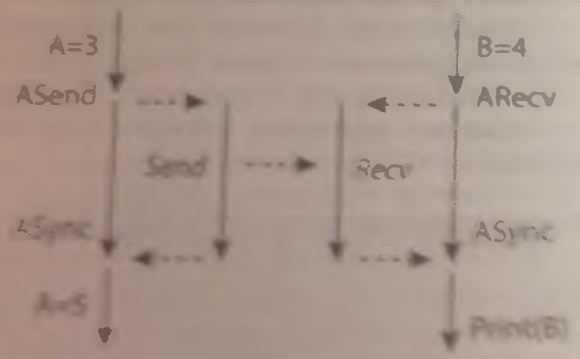
Так, в указанном выше примере при добавлении *ASync* после отправки сообщения первым процессором, но *перед* модификацией передаваемой переменной, в результате выполнения указанного фрагмента, значение 5 уже не сможет быть напечатано.

Процесс 1	Процесс 2
A=5 ASend (2, &A) ASync (2) A=6	B=0 ARecv (1, &B) Print (B)



Чтобы печаталось именно значение 3, необходимо добавить функцию синхронизации еще в одном месте — непосредственно перед печатью полученного значения на втором процессоре.

Процесс 1	Процесс 2
A=3 ASend(2, A) ... ASync(2) A=5	B=4 ARecv(1, &B) ... ASync(1) Print(B)



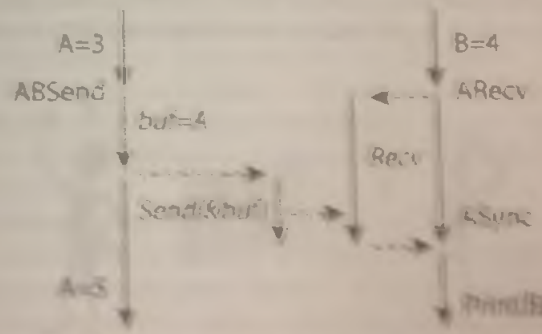
В этом случае будет напечатано именно значение 3.

Обратим внимание, что между *ASend/ARecv* и *ASync* могут быть дополнительные операции, полезные с вычислительной точки зрения. Поскольку современное сетевое оборудование может работать одновременно с основным процессором, возникает потенциальная возможность одновременно передавать данные и производить необходимые вычисления, тем самым сокращая время выполнения задачи, что и является основной целью использования асинхронных методов передачи данных.

При использовании буферизованной асинхронной передачи, вообще говоря, нет необходимости использовать дополнительную синхронизацию после ее вызова.

Процесс 1	Процесс 2
A=3 ABSend(2,&A) A=5	B=4 ARecv(1,&B) ASync() Print(B)

При выполнении данного фрагмента кода, значение 5 напечатано не будет. Изменение значения переменной *A* никак не повлияет на передаваемые данные: переменная сначала буферизуется, и потом передается содержимое именно буфера, которое уже не меняется.



Тем не менее, функции синхронизации на передающем конце может быть полезна, т.к. размер буфера ограничен

Явная синхронизация (контроль того, что пересылаемые данные действительно переданы) поможет избежать трудно диагностируемых логических ошибок, к которым может привести переполнение внутренних буферов.

Доступ к разделяемым переменным

Прежде чем описывать еще один метод синхронизации процессов — семафоры, рассмотрим пример, иллюстрирующий проблему, возникающую при совместном использовании разделяемых ресурсов.

Пусть в разных концах комнаты расположены две лампочки, самопроизвольно зажигающиеся в случайные моменты времени. Вспыхнувшая лампочка продолжает гореть до тех пор, пока ее не погасят. Двум наблюдателям поручено выключать зажигающиеся лампочки и учитывать общее число их включений, записывая его мелом на доске посередине комнаты (рис. 10).

Рассмотрим возможную последовательность действий:

- вспыхнула левая лампочка;
- первый наблюдатель прочел число, написанное на доске (365), и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366).

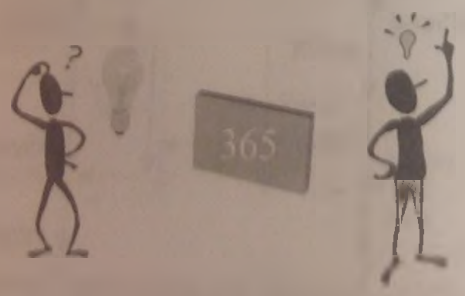


Рис. 10. Подсчет событий двумя наблюдателями

Результат на доске правильный, если второй наблюдатель на протяжении всего этого процесса к доске не подходил. В противном случае могло получиться следующее:

- вспыхнула левая лампочка;
- первый наблюдатель прочел число, написанное на доске (365), и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- вспыхнула правая лампочка;
- второй наблюдатель тоже прочел число, написанное на доске (там пока еще написано 365), и отправился гасить свою лампочку, прибавляя по пути единицу к прочитанному числу;
- погасив лампочку и вернувшись, первый исправляет число на доске, записывая получившуюся у него сумму (366);
- погасив лампочку и вернувшись, второй, в свою очередь, исправляет число на доске, записывая получившуюся у него сумму (366).

Полученный результат (366) не совпадает с правильным (367). Включались две лампочки, а общее число включений, записанное на доске, возросло только на единицу. Ошибка обусловлена тем, что между чтением первым наблюдателям числа с доски и записью им результата, второй наблюдатель получил доступ к разделяемому ресурсу — доске — и прочел уже обрабатываемое первым наблюдателем число.

Если бы все три действия — чтения числа, прибавление к нему единицы и запись исправленного числа — выполнялись как одна неделимая — *атомарная* — операция, ошибка бы не возникла.

Подробнее о том, как можно корректно решить подобную задачу, основываясь на пяти атомарных операциях: чтения, модификации, проверки, записи переменных и перехода (они поддерживаются любым традиционным процессором), можно прочитать в работах [4, 10]. Соответствующее решение является громоздким, неэффективным с точки зрения

использования процессорной мощности и плохо масштабируемым. На практике используют средства более высокого уровня [11], базовым из которых являются семафоры.

3.6.3. Семафор

Семафор — это средство синхронизации процессов. Различают бинарные семафоры (принимающие значения 0 или 1) и семафоры общего вида, принимающие произвольные неотрицательные значения. В дальнейшем будем рассматривать только семафоры общего вида, придерживаясь терминологии используемой у Дейкстры в работе [4].

Семафор — это целочисленная неотрицательная переменная, над которой можно выполнять только две атомарные операции P и V.

Атомарность понимается следующим образом: если некоторый процесс начал выполнение атомарной операции над переменной, то никакие другие процессы не будут иметь возможности обратиться к этой переменной до тех пор, пока выполнение начатой операции не будет завершено.

Операция V(S), где S — семафор, является атомарной *неблокирующей* операцией, которая увеличивает значение семафора на 1. Даже если два процессора одновременно инициируют выполнение операций V(S), значение семафора увеличится именно на 2, поскольку операции будут выполнены поочередно, в силу их атомарности.

Операция P(S), где S — семафор, является атомарной *блокирующей* операцией:

- если значение S положительно, то операция P(S) уменьшает значение семафора на 1;
- иначе процесс, выполняющий операцию P(S), переходит в режим ожидания. Управление ему будет возвращено только после того, как какой-нибудь другой процесс выполнит операцию V(S).

Очевидно, что S — это глобальная переменная, доступная всем процессам, выполняющим с ее помощью синхронизацию.

До выполнения над семафором операций P или V должна быть выполнена инициализация семафора — присвоение ему произвольного неотрицательного значения.

Наиболее естественным выглядит использование семафоров для синхронизации процессов, выполняющихся в вычислительных системах с общей памятью.

Несмотря на то, что семафор, по сравнению с командами процессора, является средством более высокого уровня, с практической точки зрения, это весьма низкоуровневое средство. Использование семафоров позволяет разрабатывать высокоэффективные программы, но их отладка сопряжена со значительными трудностями, поскольку ошибки использования семафоров сложны в локализации. Существенно сократить число ошибок позволяет идея выделения в программе *критических интервалов* и их реализация с помощью *мониторов*.

Критический интервал — это последовательность действий, выполняться которые должна как одна неразрывная операция.

Использование семафора дает нам возможность наглядно и компактно записывать решения задач взаимного исключения. Например, решение задачи о защите критического интервала запишется теперь в следующем виде:

```
Семафор Sem=1
Sum=365
```

Первый процесс	Второй процесс

while (1)
{
 	
P(Sem);
Sum=Sum+1; // критический
интервал
V(Sem);
}
	while (1)
	{
	
	P(Sem);
	Sum=Sum-1; // критический
	интервал
	V(Sem);
	}

Решение привлекательно не только своей лаконичностью, но и тем, что алгоритмы, выполняемые обоими процессами, совершенно идентичны между собой. Это дает возможность использования данного механизма при синхронизации произвольного числа процессов.

В принципе, семафоров вполне достаточно для защиты разделяемых ресурсов самого разного рода — общих переменных, файлов, каналов связи, нереентерабельных подпрограмм и т.д., однако человек прекрасен еще и тем, что ему свойственно ошибаться. Хорошо, когда вся программа уместилась на четверти листа, и единственная защищаемая переменная используется в одном — двух местах. Реальные программы, как правило, несколько больше, и разделяемые переменные могут использоваться в них достаточно интенсивно. Необходимость писать каждый раз, когда надо изменить значение такой переменной конструкции вида:

```
P(Sem);
// Действия с переменной;
V(Sem);
```

рано или поздно приведет к тому, что либо *P*, либо *V*, либо они оба будут забыты в каком-нибудь месте. Для уменьшения вероятности ошибок такого рода, рекомендуется защищать ресурсы не непосредственно с помощью семафоров, а с помощью построенных на их основе *мониторов*.

Монитор

Монитор — это набор подпрограмм, инкапсулирующих (включающих в себя) *все* обращения к защищаемому ресурсу. Доступ к ресурсу, защищенному с помощью монитора, возможен только с помощью подпрограмм монитора. Приведем пример монитора:

```
static Semaphore Sem;
static Sem;
```

Установка значения	Чтение значения	Изменение значения

Sum_Set (value)
{
P(Sem);
Sum=value;
V(Sem);
}
	Sum_Get()
	{
	int Stmp;
	P(Sem);
	Stmp=Sum;
	V(Sem);
	return Stmp;
	}
		Sum_Plus(value)
		{
		P(Sem);
		Sum=Sum+ value;
		V(Sem);
		}

Использование описателя *static* в первых двух строках и выделение приведенных подпрограмм в отдельный файл гарантирует (в рамках соглашений языка C/C++) невозможность доступа к переменным *Sem* и *Sum* иначе, чем через функции монитора *Sum_Set*, *Sum_Get* и *Sum_Plus*. В соответствии с правилами языков C/C++, эти переменные становятся невидимыми в других файлах с исходными текстами программы. Кажущееся увеличение затрат на написание текста программы окупается надежностью получаемого в результате программного кода. Например, защищенный доступ к критическому интервалу теперь можно записать короче:

```
Sum_Plus(1)
```

Область применения семафоров не ограничивается задачами взаимного исключения. Другие примеры их использования приведены в следующем параграфе и в разделе «Адаптивное интегрирование».

3.6.4. Барьерная синхронизация

Барьер является еще одним видом синхронизации взаимодействующих процессов.

Барьер — это функция, вызываемая всеми процессами, участвующими в акте взаимной синхронизации. Ни один из

вызвавших эту функцию процессов не завершит ее выполнение до тех пор, пока все процессы не начнут выполнение этой функции.

Удобно представить себе барьер как двухэтапную функцию: на первом этапе все процессы *должны* осуществить вход в барьер. На втором этапе они *могут* осуществить выход из него. Таким образом, второй этап может быть выполнен только после полного завершения первого этапа. Операция «барьер» предполагает взаимодействие множества процессов, поэтому для выполнения требует существенного времени, оценка которого приведена в разделе, посвященном описанию метода сдваивания.

В системах, поддерживающих менее затратные методы синхронизации (например, рассмотренные ранее), явного использования барьеров следует избегать. Тем не менее, барьеры полезны на этапах отладки программ, обладающих сложной логикой выполняемых операций. Использование барьеров позволяет явным образом разбивать такие программы на относительно независимые фрагменты, обособленная отладка которых проще чем отладка сразу всей программы. Следует иметь в виду, что работоспособность отлаженной с помощью барьеров программы может нарушаться в случае их удаления, поскольку барьеры не только упрощают отладку, но и маскируют, за счет дополнительной синхронизации, логические ошибки.

Глава 4

Базовые параллельные методы

> ... если для нас представляют интерес реально работающие системы, то требуется убедиться (и убедить всех сомневающихся) в корректности наших построений
>
> ... системе часто придется работать в невоспроизводимых обстоятельствах, и мы едва ли можем ожидать сколько-нибудь серьезной помощи от тестов
>
> *Э.В. Дейкстра, 1966*

Пожелание Дейкстры актуально сегодня не меньше, чем в далеком 1966 году. Практически значимых методов определения корректности параллельных программ по-прежнему не выведено, хотя в этом направлении и ведутся интенсивные работы. Недетерминированность поведения параллельной программы, невозможность преднамеренного устойчивого воспроизведения ошибочного состояния программы обуславливает тот факт, что тестирование совершенно не пригодно для установления правильности параллельной программы. Недетерминированность параллельных программ заставляет, по возможности, использовать иные подходы к созданию надежного программного обеспечения для многопроцессорных систем.

Можно указать широкие классы задач, для которых известны принципы создания эффективных параллельных алгоритмов. Их рассмотрение крайне важно, поскольку вооружает нас двумя возможностями:

- непосредственно решать множество актуальных задач соответствующих классов;
- конструировать на их основе новые алгоритмы.

Несмотря на отсутствие формальных доказательств корректности рассматриваемых в этой главе методов, их правильность подтверждена многолетней практикой. Их сила — в логической простоте. Именно простота обеспечивает их относительно легкую воспроизводимость и применимость для решения самых разных задач.

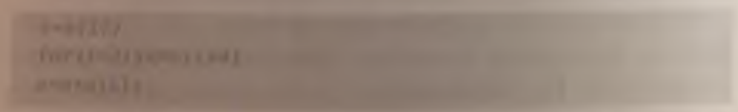
Перечислим эти методы:

- методы статической балансировки загрузки процессоров:
  - метод сдваивания;
  - метод геометрического параллелизма;
  - метод конвейерного параллелизма;
- методы динамической балансировки загрузки процессоров:
  - метод коллективного решения;
  - метод диффузной балансировки загрузки.

Далее для каждого из них будет описана область потенциального применения, особенности и такие основные свойства, как ускорение и эффективность. Метод диффузной балансировки будет рассмотрен отдельно, в главе 8, посвященной методам динамической балансировки загрузки процессоров.

4.1. Метод сдваивания

Рассмотрим в качестве модельного примера задачу определения суммы элементов массива a, содержащего n элементов. Оценим время работы последовательного алгоритма.



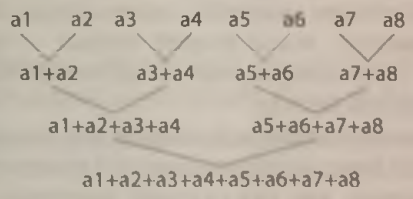
Примем во внимание только операции сложения чисел, пренебрегая дополнительными операциями, связанными с организацией цикла суммирования, чтением элементов

массива из памяти и им подобными. Эти операции могут существенно влиять на время выполнения программы, но, с точки зрения оценки свойств рассматриваемого метода, они не имеют принципиального значения.

Итак, пусть τ_c — время, затрачиваемое на сложение двух чисел, T_1 — время работы последовательного алгоритма. Тогда:

$$T_1 = (n-1)\tau_c.$$

Рассмотренный алгоритм предполагает сугубо последовательное выполнение операций. Степень его параллелизма равна 1. Для параллельной обработки требуется сконструировать другой алгоритм. Построим параллельный алгоритм, используя метод сдваивания. В качестве примера рассмотрим задачу определения с помощью 4-х процессоров суммы 8-ми элементов массива (рис. 11).



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. 11. Определение суммы элементов массива чисел на четырех процессорах методом сдваивания

В общем случае при размере массива $n = 2^q$ алгоритм сдваивания будет выполнен за $q = \log_2 n$ стадий. На первой

стадии выполняется параллельно $n/2$ действий, на второй $n/4$, ..., на последней — одно. Таким образом, степень параллелизма меняется от стадии к стадии и на стадии k равна $n/2^k$. Определим ускорение алгоритма сдваивания на системе из p процессоров. Пусть $p = n/2$, тогда:

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

Оценка показывает, что метод обеспечивает высокое ускорение, но достаточно низкую эффективность. Например, при $n = 1024$ ускорение равно 102, эффективность — 20%. Основная причина низкой эффективности заключается в том, что практически на всех стадиях большая часть процессоров простаивает. Действительно, только на первом процессоре вычисления выполняются на всех стадиях. Уже после первой стадии половина процессоров оказываются незадействованной, после второй стадии — три четверти процессоров простаивают, что существенно ограничивает масштабируемость метода.

Рассмотренный алгоритм предполагает, что для обработки n элементов будет использовано $n/2$ процессоров, большая часть которых, как уже сказано, практически не будет принимать участия в работе. В случае разработки вычислительной системы, основной целью которой является суммирование за заданное короткое время, такое положение может оказаться приемлемым и, более того, необходимым. Однако, в общем случае, более реалистичным выглядит другой подход, использующий существенно меньшее число процессоров.

В соответствии с ним, каждый из процессоров выполняет обработку в два этапа. Массив длины n разбивается на p равных частей, обработка каждой из которых выполняется на соответствующем процессоре. На первом этапе каждый из процессоров находит сумму n/p элементов соответствующей ему части элементов массива. На втором этапе с помощью

метода сдваивания определяется окончательная сумма. Отметим, что на первом этапе взаимодействие между процессорами не требуется (степень параллелизма равна p). Взаимодействие необходимо только на втором этапе. Оценим время, необходимое для вычисления суммы при использовании p процессоров.

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

$$\tilde{S}_p = \frac{n\tau_A}{\tau_A \cdot \frac{n}{p} + \tau_A \log_2 p} = p \frac{1}{1 + \frac{p}{n} \log_2 p},$$

$$\tilde{E}_p = \frac{1}{1 + \frac{p}{n} \log_2 p}.$$

Нетрудно увидеть, что при $p << n$ метод обеспечивает высокую эффективность и ускорение, близкое к используемому числу процессоров. В частном случае, при $p = \frac{n}{\log_2 n}$:

$$\tilde{S}_p = \frac{p}{2} \tilde{E}_p = 50\%.$$

При оценке использовано предположение о том, что n/p — целое число. В общем случае это неверно, и массив следует разбить на части, размеры которых отличаются не более чем на единицу. Тогда, предполагая, что n/p — большое число, дополнительным дисбалансом вычислительной нагрузки процессоров можно пренебречь.

Основные свойства метода уже описаны, но для полноты картины необходимо учесть и те новые операции, которые обеспечивают взаимодействие процессоров. Пусть для передачи числа от одного процессора другому требуется время τ_s. В случае системы с распределенной памятью, это время

определяется свойствами канала передачи данных — его латентностью и пропускной способностью. В случае системы с общей памятью, τ_s определяется временем обращения к синхронизирующему примитиву, например, к семафору. Итак, время работы алгоритма с учетом взаимодействия процессоров:

$$T_p = \tau_A \cdot \frac{n}{p} + (\tau_A + \tau_s) \log_2 n,$$

$$S_p = p \frac{1}{1 + \left(1 + \frac{\tau_s}{\tau_c}\right) \frac{p}{n} \log_2 n}.$$

При p, сравнимом с n, и при $\tau_s > \tau_c$, операции, обеспечивающие взаимодействие процессоров, могут существенно снижать ускорение и эффективность. Тем не менее, рассмотренный метод, в силу своей простоты и универсальности, широко применяется на практике. С его помощью удобно выполнять редукционные операции, такие как определение суммы или произведения множества чисел, поиск минимума или максимума набора данных и другие.

Аналогичный метод с успехом используется и для выполнения обратной операции — для быстрой рассылки одинаковых данных множеству процессоров. Пусть в системе с распределенной памятью требуется передать значение некоторой, известной на первом процессоре величины на все остальные процессоры. Последовательная передача от одного процессора всем остальным, согласно алгоритму A12, потребует времени $T_p = \tau_s (p-1)$.

Алгоритм A12

Последовательная рассылка

```
// p - общее число процессов.
// rank - номер выполняемого процесса,
// каждому процессу присвоен уникальный номер
// из диапазона [0,p-1]
```

```
if(rank==0)
{
  for(next=1;next<p;next++)
Send(next);
}
else
  Recv(0);
```

Использование алгоритма А13, основанного на методе сдваивания, позволяет сократить время рассылки до $T_p = \tau_s \log_2 p$, где $\lceil a \rceil$ — наименьшее целое число, не меньшее a. Схема рассылки данных, выполняемой согласно алгоритму А13, приведена на рисунке 12.

Алгоритм А13

Рассылка данных с использованием бинарного дерева

```
SendOneToAll()
{
  for(p1=1;p1<p;p1*=2);

  if(rank>0)
  {
// определим номер младшего в rank разряда,
// отличного от 0
for( delta =1; ( delta & rank ) == 0 ;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);
  }
}
```

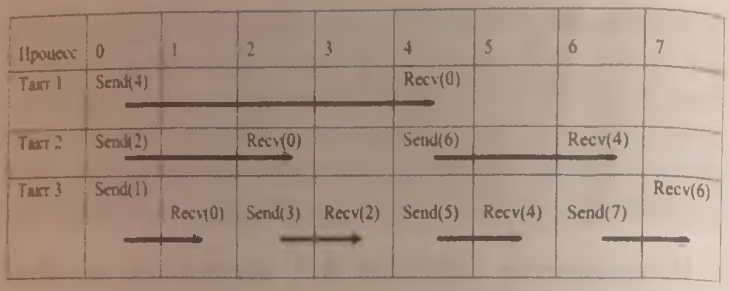


Рис. 12. Рассылка данных с использованием бинарного дерева, $p = 8$

4.1.1. Быстрый алгоритм выбора частичных сумм

Рассмотрим теперь возможность сокращения времени выбора правильных частичных сумм (раздел «2.2.1. Сложение многоразрядных чисел», алгоритм А8). Для упрощения описания предположим, что число процессов является степенью двойки. Выбор того, какая именно сумма из двух найденных является верной, может быть основан на следующей идее. Разобьем все множество процессов на две равные части: $[0, p/2 - 1]$ и $[p/2, p]$.

Пусть имеется алгоритм, определяющий на числе процессов k требуемый ответ за время $S(k)$. Тогда первой половиной процессов ответ может быть найден за время $S\left(\frac{p}{2}\right)$. На второй части процессов можно было бы найти ответ за такое же время при условии, что заранее известно, чему равен разряд переноса от процесса с номером $\frac{p}{2}-1$. В начальный момент это неизвестно, но будет известно к моменту окончания вычислений первой половиной процессоров. Если повторно воспользоваться принципом спекулятивной оптимизации, то можно одновременно на первой и на второй половине процессов выполнить определение правильных сумм для случая прихода 0 и для случая прихода 1. Далее для

определения окончательного ответа потребуется распределить по всем процессам второй половины разряд переноса от первой половины. На получение данных о разряде переноса потребуется одна операция, на распространение с помощью метода сдваивания этих данных по всем процессам второй половины — $\log_2 \frac{p}{2}$ операций. Общее время решения будет равно:

$$S(p) = S\left(\frac{p}{2}\right) + \log_2 \frac{p}{2} + 1.$$

Решение этого функционального уравнения достаточно просто определяется с помощью предельного перехода [12].

$$S(p) = S\left(\frac{p}{2}\right) + \log_2 \frac{p}{2} + 1 =$$

$$S\left(\frac{p}{4}\right) + \log_2 \frac{p}{4} + \log_2 \frac{p}{2} + 2 = \ldots$$

$$S(1) = 1 = \log_2 2.$$

$$S(p) = \log_2 2 + \log_2 4 + \ldots +$$

$$+ \log_2 \frac{p}{4} + \log_2 \frac{p}{2} + \log_2 p.$$

$$S(p) = 1 + 2 + 3 + \ldots + \log_2 p.$$

$$S(p) = \frac{\log_2 p(\log_2 p + 1)}{2}.$$

Окончательно, вместо (1) получим:

$$T_p = \frac{8n}{p}\tau_c + \frac{\log_2 p(\log_2 p + 1)}{2}\tau_s.$$

4.1.2. Барьерная синхронизация на основе синхронных обменов

Ранее был рассмотрен алгоритм ***SendOneToAll*** рассылки данных от одного процессора всем остальным (алгоритм А13). Изменим порядок выполнения тактов этого алгоритма на обратный и изменим направление каждой передачи информации на обратное (согласно рисунку 13). В результате получим алгоритм ***RecvOneFromAll***.

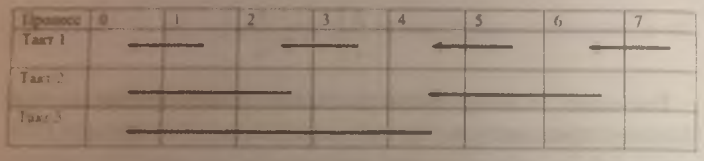


Рис. 13. Сбор данных с использованием бинарного дерева, $p = 8$

Барьерная синхронизация может быть реализована с помощью алгоритма А14.

Алгоритм А14

Рассылка данных с использованием бинарного дерева, $p = 8$

```
Barrier()
{
RecvOneFromAll()
SendOneToAll()
}
```

Вызов функции ***RecvOneFromAll*** гарантирует, что первый процесс не закончит ее выполнение до тех пор, пока все остальные процессы не выполнят ее полностью. Таким образом, функция обеспечивает реализацию первого этапа барьерной синхронизации — вход в барьер. Выполнение этого этапа гарантирует, что все процессоры начали выполнять функцию барьерной синхронизации. Следующим шагом вызывается функция ***SendOneToAll***. Тем самым, по всем процессорам будет распространена информация о том, что

барьерная синхронизация состоялась, и можно продолжать работу.

Время выполнения рассмотренного алгоритма барьерной синхронизации: $T = 2\tau_s \log_2 n$. Альтернативным алгоритмом может служить симметричный барьер типа «бабочка» [13]. Схема выполнения обменов согласно этому алгоритму приведена на рисунке 14. Ее корректность может быть доказана с помощью принципа математической индукции.

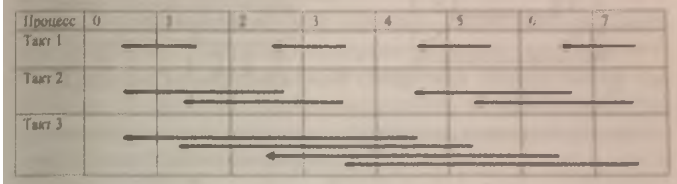


Рис. 14. Барьер типа «бабочка», $p = 8$

Требуемое число тактов $\log_2 n$ вдвое меньше, чем у барьера, основанного на использовании бинарного дерева, но время работы $T = \tau_s \log_2 n$ будет вдвое меньше только при условии, что на каждом такте все обмены могут выполняться одновременно. Число тактов меньше, чем у алгоритма A14, за счет большей нагрузки на коммуникационную подсистему. Для этого должны физически существовать все каналы, предусмотренные топологией гиперкуб (рис. 15), что может не соблюдаться на других топологиях. Учитывая, что на практике при большом числе процессоров некоторые каналы передачи данных обеспечивают взаимодействие множества пар процессоров, симметричный барьер может потребовать большего времени выполнения, чем алгоритм A14. Пример подобной топологии приведен на рисунке 16. Иерархический алгоритм выполнится на ней за 8 тактов, а симметричный — за 3 + 8 = 11 тактов. Обмены последнего

такта будут выполняться последовательно через единственный канал, связывающий две группы процессоров, каждая из которых содержит восемь процессоров.

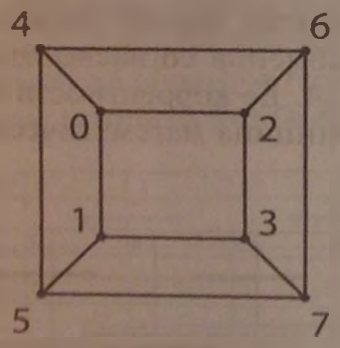


Рис. 15. Топология гиперкуб, $p = 8$

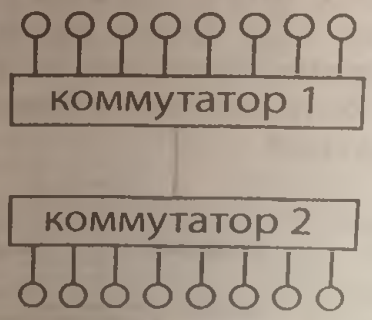


Рис. 16. Несимметричная топология связей процессоров, $p = 16$

4.1.3. Стена Фокса

Обсуждение методов организации параллельной обработки данных удобно обсуждать на примере задачи построения стены Фокса [14, 15, 16]. Представим себе высокую длинную стену, состоящую из рядов кирпичей. Укладку каждого кирпича будем рассматривать как элементарное задание. Обсудим некоторые способы согласованного построения такой стены группой каменщиков.

4.2. Метод геометрического параллелизма

Геометрический параллелизм (*domain decomposition*, параллелизм данных) является вторым (после метода сдваивания) рассматриваемым методом организации параллельных вычислений. Он широко применяется при решении задач математической физики, при обработке изображений и во многих других приложениях. Метод ориентирован на вы-

полнение множества однотипных действий над однородными *взаимосвязанными* данными.

Распределим данные по процессорам равными частями и поручим обработку каждой части соответствующему процессу. Подход эффективен при условии, что действия, выполняемые одним процессом, зависят лишь от небольшого, ограниченного объема данных, расположенных на других процессорах. Желательно, чтобы эти «чужие» для некоторого процесса данные были расположены только на небольшом количестве процессоров.

Указанное свойство будем называть свойством *локальности* алгоритма. Например, алгоритм решения произвольной системы линейных уравнений свойством локальности, как правило, не обладает. Изменение любого из коэффициентов уравнений может привести к кардинальному изменению всего решения. Значения всех переменных существенно зависят от значений каждого из коэффициентов матрицы системы. Напротив, алгоритм моделирования системы материальных точек, соединенных в цепочку упругими пружинками, свойством локальности обладать будет. При достаточно длинной цепочке изменение положения одной из точек (или упругости одной из пружинок) на левом конце цепочки слабо (и не сразу, если рассматривать динамику поведения системы) скажется на поведении точек на ее правом конце.

Решая задачу построения «стены Фокса», можно разбить кладку на равные по длине участки и поручить постройку каждого участка отдельному каменщику. В этом случае все каменщики могут начать работу одновременно, укладывая нижний слой кирпичей (рис. 17). Перед укладыванием очередного слоя каждому каменщику следует убедиться, что кирпичи предыдущего слоя уложены не только на его участке, но и на прилегающих участках. Если работа на соседних участках еще не закончена, возникают вынужденные паузы, связанные с синхронизацией работ на соседних участках. Отметим, что паузы могут возникать, даже если каменщики

работают с одинаковой скоростью, поскольку объем работ, вообще говоря, зависит от расположения участка — например, он разный для крайних и для внутренних частей стены.

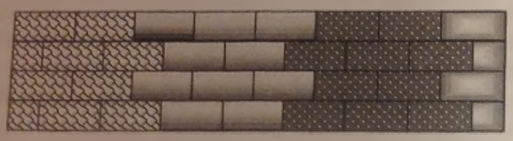


Рис. 17. Геометрическое решение

Оценим время, необходимое для постройки стены. 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 nk кирпичей:

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

В случае нескольких процессов, каждому выделяется вертикальный участок стены длиной $\frac{n}{p}$. Объем полезной работы составит $\tau_c \frac{n}{p}$. Однако, прежде чем укладывать новый кирпич на границах фрагмента стены, следует убедиться, что это действительно можно сделать, что соответствующий кирпич нижнего слоя уже уложен.

Для определенности предположим, что процессы выполняются на вычислительной системе с распределенной памятью, причем каждый процесс — на отдельном процессоре. Тогда перед укладкой очередного кирпича на левой границе фрагмента стены в процессе с номером *rank* должна быть выполнена функция получения сообщение от процесса с но-

мером *rank* – 1. Следовательно, в процессе *rank* – 1 должна быть выполнена функция передачи соответствующих данных. В общем случае, передать данные следует не только от процесса *rank* к процессу *rank* – 1, но и в обратную сторону. Таким образом, между каждыми двумя граничащими между собой процессами следует передать два сообщения. Всего процесс, не являющийся первым или последним в цепочке, должен обменяться со своими соседями четырьмя сообщениями. Наименьшее время работы составит:

$$T_p = \tau_c \frac{kn}{p} + 4k\tau_s. \qquad (2)$$

Однако, время работы будет именно таким отнюдь не при любом алгоритме передачи данных. Рассмотрим ряд реализаций изложенной идеи, в предположении, что мы используем синхронные функции передачи данных (алгоритм А15, рис. 18).

Алгоритм А15

Блокировка при передаче данных

```
for(шаг=0;шаг<k;шаг++)
{
for(кирпич=rank*n/p;кирпич<(rank+1)*n/p;кирпич++)
  Уложить(кирпич)
if(rank>0) Send(rank-1, кирпич уложен )
if(rank<p-1) Send(rank+1, кирпич уложен )
if(rank>0) Recv(rank-1, место готово? )
if(rank<p-1) Recv(rank+1, место готово? )
}
```

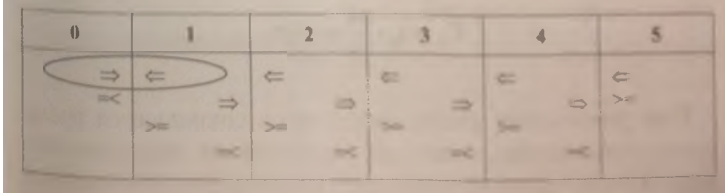


Рис. 18. Взаимная блокировка процесссов 0 и 1

Как видим, данное решение неработоспособно, поскольку первыми во всех процессах выполняются функции *передачи данных*. Никакой процесс не выполняет функцию *приема данных*, следовательно, все процессы будут находиться в состоянии блокировки (*тупика, клинча*). В частности, процессы 0 и 1 находятся в состоянии взаимной блокировки. Время выполнения такого алгоритма будет равно бесконечности.

Изменение порядка выполнения операции обмена данными существенно меняет ситуацию. Теперь алгоритм будет выполнен за конечное время, однако это время по-прежнему не будет совпадать с (2). Чтобы убедиться в этом, достаточно обратиться к рисунку 19, иллюстрирующему последовательность выполнения обменов.

Алгоритм 4.6

Неэффективная организация передачи данных

```
if(rank>0) Send(rank-1, данные узлам );
if(rank<p-1) Recv(rank+1, место хранения );
if(rank<p-1) Send(rank+1, данные узлам );
if(rank>0) Recv(rank-1, место хранения );
```

Время работы алгоритма составит

$$T_p = \tau_c \frac{nk}{p} + \tau_c kp. \qquad (3)$$

При увеличении числа процессоров сокращается время выполнения вычислений, но растет время передачи данных. Это существенно уменьшает максимально достижимое ускорение.

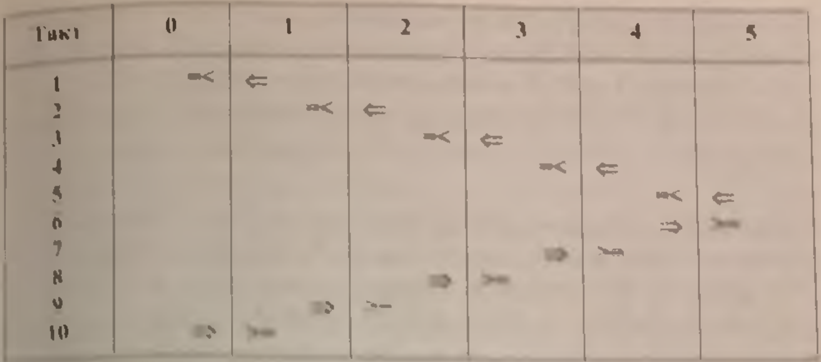


Рис. 19. Схема обменов при работе алгоритма A16

Обратим внимание на то, что процессы с номерами 2—5 могут осуществлять взаимную передачу данных на тех же тактах, что и процессы 0—1. Поскольку алгоритм обладает свойством локальности, взаимные обмены между далеко отстоящими друг от друга по цепочке процессорами не должны влиять друг на друга. Соответствующая модификация программы основана на идее разбиения процессоров на пары, осуществляющие обмены одновременно и независимо друг от друга. Сначала взаимодействуют процессы с номерами $2k$ и $2k + 1$, а затем передают между собой данные процессы с номерами $2k - 1$ и $2k$. Порядок выполнения операций обмена данными согласно алгоритму A17 показан на рисунке 20. Именно этот алгоритм обеспечивает выполнение оценки (2).

Алгоритм A17

Эффективная схема обменов

```
for(шаг=0;шаг<kmax;шаг++)
{
for(вершина=rank*n/p;вершина<(rank+1)*n/p;вершина++)
    Изменить(вершина)

if(rank%2)
{
```

```
if(rank>0) Send(rank-1, кирпич уложен )
if(rank>0) 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>0) Send(rank-1, кирпич уложен )
if(rank>0) Recv(rank-1, место готово? )
}
}
```

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

Рис. 20. Схема попарных обменов

Оценим ускорение и эффективность алгоритма A17.

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

Как и следовало ожидать, значения ускорения и эффективности не зависят от высоты стены, а только от ширины обрабатываемого каждым процессом участка стены. Точнее говоря, — от значения величины $\gamma = 4\frac{p}{n}\frac{\tau_s}{\tau_c}$. Уменьшение γ ведет к увеличению ускорения практически до p, а эффективности — практически до 100%. Внимательное рассмотрение структуры γ показывает, что даже при относительно

большом времени на передачу данных можно выбрать такое значение n, при котором γ будет существенно меньше 1, и эффективность будет высокой. Именно поэтому метод геометрического параллелизма позволяет эффективно использовать многопроцессорные системы, содержащие большое число процессоров.

Высокие показатели обусловлены тем, что, во-первых, при увеличении числа процессоров, время на их взаимодействие не меняется и остается относительно малым. Во-вторых, в данном методе фактически нет новых операций, снижающих эффективность расчетов. Еще до начала выполнения расчета определяется, какую именно часть работы будет выполнять каждый из процессов, и в ходе вычислений этот заранее принятый план не изменяется. Таким образом, метод относится к методам статической балансировки нагрузки процессоров. В этом его сила, но в этом и его слабость.

Все сделанные оценки справедливы только при выполнении ряда условий, ранее явно не указанных, а именно:

- кирпичи могут быть распределены между процессами строго поровну;
- число операций, необходимых для укладки каждого из кирпичей, не зависит от номера кирпича (от его положения в стене);
- все процессоры имеют одинаковую производительность — выполняют вычислительные операции с одинаковой скоростью;
- все каналы передачи данных имеют одинаковые характеристики (латентность и пропускную способность).

На практике эти условия либо выполняются приближенно, либо совсем не выполняются.

Например, из рисунка 17 видно, что последний каменщик укладывает меньше кирпичей, чем остальные, поскольку работу не удалось распределить строго поровну. Более внимательное изучение рисунка 17 показывает, что в нижнем

ряду (и не только) первый и последний кирпичи отличаются от остальных своим размером — они вдвое меньшей длины. При укладке стены некоторые каменщики тратят дополнительное время на изготовление половинок — раскалывают часть кирпичей. Таким образом, первый и последний каменщики будут тратить на укладку некоторых рядов не такое время, как остальные. В случае численного моделирования некоторого физического процесса, первый и последний процессы отличаются тем, что, кроме расчета внутренних точек, на них обрабатываются граничные точки. Они обрабатываются по иным, нежели внутренние точки, формулам. Больше времени займет эта обработка или меньше — вопрос второй, главное, что в системе будут процессы, затрачивающие различные времена на выполнение каждого из шагов (при укладке каждого из ряда кирпичей). Таким образом, даже при одинаковых процессорах возникает ситуация, в которой часть процессов уже выполнили свою работу и находятся в состоянии ожидания окончания работы остальными. Подобный непроизводительный простой снижает и ускорение, и эффективность.

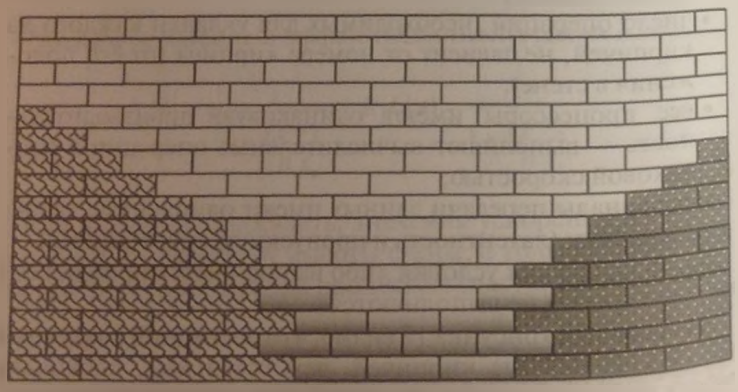


Рис. 21. Геометрическое решение

Предположим, что по некоторой причине только один из процессов тратит на выполнение своей части работы вдвое больше времени, чем каждый из остальных. Условно назовем этот процесс «медленным», хотя причиной больших затрат времени может быть не низкая производительность соответствующего процессора, а большая трудоемкость обработки выделенного процессу участка стены. К чему это приведет в системе из p процессов? Через некоторое время стена приобретет показанные на рисунке 21 очертания (медленный процесс обрабатывает середину стены). После укладки не более чем p рядов, все процессы начнут тратить на выполнение укладки каждого ряда столько же времени, сколько и «медленный» процесс. Это не значит, что они будут медленно выполнять вычисления, — это значит, что половину времени они будут простаивать в ожидании самого медленного процесса, но результат один — эффективная производительность всей системы снизилась вдвое. В рамках метода *статической* балансировки загрузки ситуацию можно исправить, только заранее зная и трудоемкость укладки каждого кирпича, и производительность каждого процессора. Например, если известно, что на обработку всех кирпичей требуется одинаковое время, а один из процессоров работает вдвое медленнее других, то именно этому процессору можно заранее распределить участок стены вдвое меньшего, чем другим, размера. Тогда все процессы будут тратить на выполнение работ одинаковое время, простоев не будет, и расчет будет эффективным. В противном случае следует привлекать методы динамической балансировки нагрузки процессоров, некоторые из которых рассматриваются в последующих главах.

Можно сделать вывод о том, что эффективность организации параллельной работы в рамках метода геометрического параллелизма будет тем выше, чем большие объемы обрабатывает процесс на каждом ряду данных (длиннее участки стены). При достаточно длинных участках можно ожидать

эффективности, близкой к единице. При коротких участках стены (много процессов при малой длине стены) эффективность будет невысокой. Время независимой работы каждого процесса будет невелико относительно времени, требуемого для взаимной передачи данных на стыках участков.

Эффективность этого метода не зависит от высоты стены. Главным фактором, определяющим эффективность, является достаточная протяженность участков стены, выделенных каждому из каменщиков.

4.3. Метод конвейерного параллелизма

Пусть требуется выполнить следующие вычисления

```
for(step=0;step<k;step++)
{
  fnew[0]=g(step)

   for(i=1;i<n;i++)
    fnew[i]=fnew[i-1]+f[i]

   for(i=1;i<n;i++)
    f[i]=fnew[i]
}
```

Непосредственно метод геометрического параллелизма использовать нельзя, поскольку при *step* = 0 возможно только последовательное определение величин *fnew*[*i*]. Рассматриваемые вычисления могут быть выполнены с помощью конвейерного метода организации работ. Проиллюстрируем его на примере построения стены Фокса.

Конвейерное распределение работ предполагает, что первый каменщик полностью укладывает первый слой кирпичей, второй каменщик — второй слой и так далее. На рисунке 22 слои одного цвета укладываются одним и тем же каменщиком. После того, как первый полностью уложит

первый слой, он продолжает укладку слоя $p + 1$. Очевидно, что при таком подходе стартовать может только первый каменщик, остальные вынуждены ожидать, пока будут уложены нижние слои. Дольше всех будет ожидать своей очереди последний каменщик. Аналогичная ситуация сложится в конце работы — первый каменщик закончит первым и будет простаивать, несмотря на то, что работа не закончена. Если стена достаточно высокая и длинная, большую часть времени все каменщики будут равномерно загружены.

Последнее утверждение верно при условии, что все каменщики работают с одинаковой скоростью и что объем работ не меняется от слоя к слою стены. На практике всегда есть некоторая разница как в скорости работы каменщиков, так и во времени, необходимом для обработки каждого слоя. Как и в случае геометрического параллелизма, условие одинаковости времени укладки каждого кирпича существенно. Если это время значительно различается, эффективность метода будет низкой.

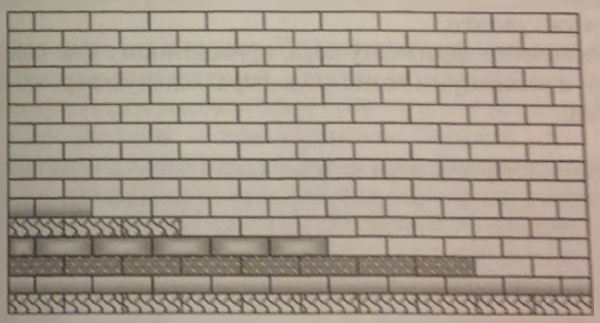


Рис. 22. Конвейерное решение

Если время обработки каждого кирпича совпадает, но каменщики имеют разную производительность, то время построения стены будет определяться самым медленным ка-

менщиком. В этом отношении ситуация также аналогична методу геометрического параллелизма. Поскольку оба метода являются методами статической балансировки загрузки, производительность системы полностью определяется процессором, затрачивающим на выполнение вычислений наибольшее время. Однако, если в случае геометрического параллелизма можно относительно просто обеспечить равное время обработки процессорами, обладающими разной производительностью (достаточно изначально распределить более медленным процессорам меньшие фрагменты стены), то в случае рассматриваемой стратегии, способ статического равномерного распределения работ не очевиден. Более того, внимательное изучение последовательности действий, необходимых для выполнения работы, показывает, что эффективность предложенной стратегии отнюдь не так высока, как у метода геометрического параллелизма. Главная причина низкой эффективности кроется в необходимости получения перед укладкой каждого нового кирпича подтверждения того, что соответствующий кирпич нижнего слоя уже уложен.

Вернемся к фрагменту кода, приведенному в начале раздела, и рассмотрим возможную параллельную версию алгоритма.

```
// rank - номер процессора

for(step=rank;step<k;step+=rank)
 {
  f[0]=g(step)

   for(i=1;i<n;i++)
{
    if(step>0)
Recv(rank-1, &(f[i]))

    f[i]=f[i]*f[i]
```

```
    if(step<k-1)
Send(rank+1, &(fnew[i]))
}

   for(i=1;i<n;i++)
    f[i]=fnew[i]
}
```

Центральный цикл содержит по две операции передачи данных при обработке каждого из узлов (за исключением крайних узлов). Следовательно, основные характеристики метода можно оценить следующим образом:

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

Эффективность не зависит от сложности решаемой задачи (от величин n и k) и целиком определяется отношением $\frac{\tau_s}{\tau_c}$, которое, в зависимости от конкретных условий, может быть как мало, так и велико. При использовании метода геометрического параллелизма, с ростом размера решаемой задачи сохранялась высокая эффективность для все большего числа процессоров — при достаточно большой сетке метод геометрического параллелизма хорошо масштабируем. В случае конвейерного решения ситуация иная: ускорение растет с увеличением числа используемых процессоров, но эффективность их использования останется низкой. Соответственно, ускорение, достигаемое при одном и том же числе процессоров, меньше, чем в методе геометрического параллелизма:

$$\frac{S_{\text{геом}}}{S_{\text{конв}}} = \frac{1+2\frac{\tau_s}{\tau_c}}{1+4\frac{p}{n}\frac{\tau_s}{\tau_c}} \approx 1+2\frac{\tau_s}{\tau_c}\bigg|_{n\gg p}.$$

Основная причина — высокие накладные расходы на передачу данных — обусловлена тем, что информация о каждом из узлов обрабатываемой сетки мигрирует с процессора на процессор. В методе геометрического параллелизма, напротив, информация о большей части узлов никогда не передается между процессорами. Информация о большинстве узлов хранится и обрабатывается на одном и том же процессоре, что значительно сокращает потери на передачу данных.

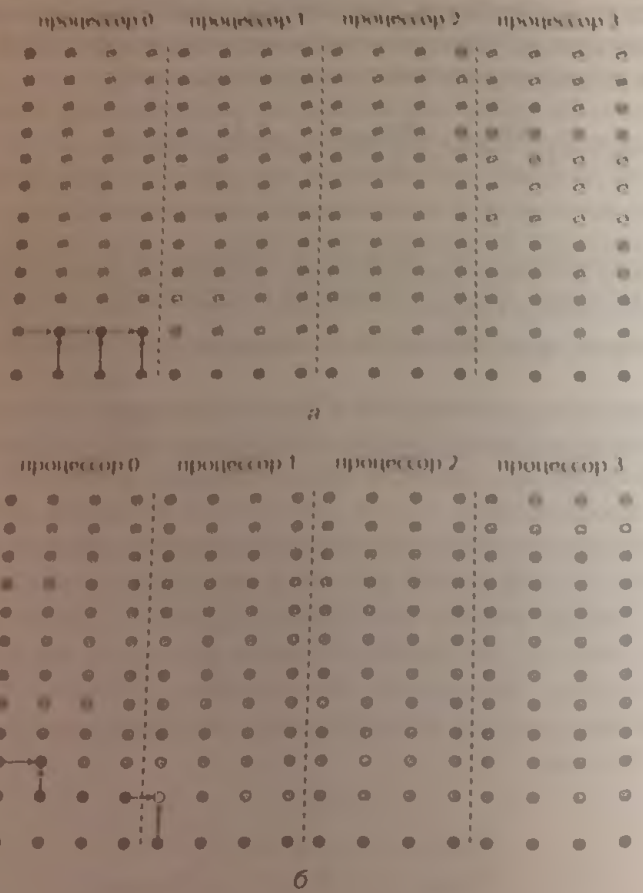


Рис. 23. Эффективное конвейерное решение

Рассмотрим другую стратегию распределения данных при решении с помощью метода конвейерного параллелизма. Распределим их точно так же, как и в методе геометрического параллелизма. Несмотря на новый вид распределения данных, по-прежнему, вначале выполнять вычисления можно будет только на процессоре 0 (рис. 23а). Остальные процессоры будут простаивать до тех пор, пока на нулевом процессоре не будут обработаны n/p точек (с номерами $\left[0, \frac{n}{p}-1\right]$). После того, как n/p точек будут обработаны, информацию о точке с номером $\frac{n}{p}-1$ следует передать процессору 1. Теперь продолжать работу смогут уже два процессора — процессор 0 будет вычислять значения третьего слоя в узлах $\left[0, \frac{n}{p}-1\right]$, а процессор 1 — значения второго слоя в узлах $\left[\frac{n}{p}, 2\frac{n}{p}-1\right]$ (рис. 23б).

В результате, поскольку на каждом слое каждый из процессоров только один раз принимает данные и только один раз передает их (рис. 24а), оценка основных характеристик метода приобретает вид:

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

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 параллеллизма это принципиально не так. Слои, обрабатываемые разными процес-

сорами, имеют разные номера. Из этого, однако, не следует, что необходимо хранить информацию о всех вычисленных слоях. Точно так же, как и в методе геометрического параллелизма, достаточно на каждом процессоре хранить информацию только о двух соседних слоях. О слое, рассчитываемом на процессоре в данный момент, и о предыдущем слое (рис. 24б).

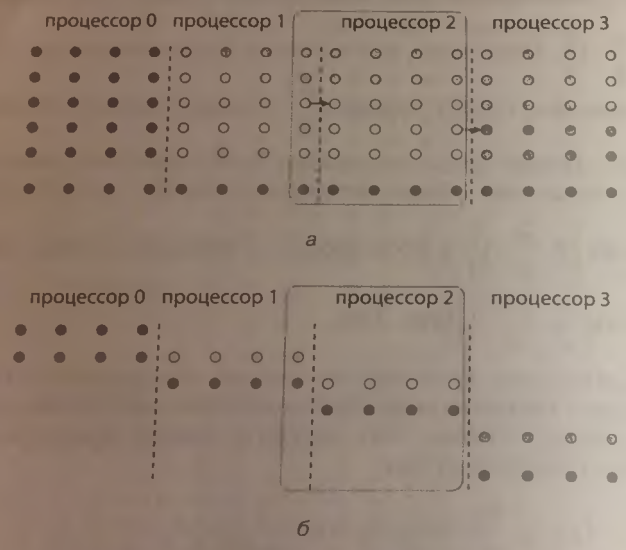


Рис. 24. Эффективное конвейерное решение

Напомним, что приведенные оценки справедливы только при соблюдении условий, указанных при обсуждении геометрического парллелизма (раздел 4.2). Кроме того, они не учитывают потерь, обусловленных начальным простем практически всех процессоров (при загрузке конвейера) и финальными простоями, вызванными тем, что про-

цессоры с меньшими номерами закончат работу раньше остальных.

Учтем влияние начальных и конечных простоев на ускорение и эффективность. Определим момент, после которого все процессоры включатся в работу, считая, что число рядов k кратно числу процессоров p.

Очевидно, что последним к работе приступит процессор с номером $p - 1$. Произойдет это после того, как будут обработаны $\frac{n}{p}(p-1)$ точек первого ряда:

$$T_{start} = (p-1)\left(\frac{n}{p}\tau_c + 2\tau_s\right).$$

Далее, до тех пор, пока первый процессор не закончит укладку самого верхнего ряда, в работе будут принимать участие все процессы. В таком режиме каждым процессором будет обработано $(p - k)$ рядов:

$$T_{middle} = (k-p)\left(\frac{n}{p}\tau_c + 2\tau_s\right).$$

В заключение потребуется время на укладку $\frac{n}{p}(p-1)$ точек последнего ряда:

$$T_{end} = (p-1)\left(\frac{n}{p}\tau_c + 2\tau_s\right).$$

Общее время выполнения работ будет равно сумме трех указаных времен:

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

Полагая, что $p + k >> 2$, окончательно получим

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

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

В отличие от метода геометрического параллелизма, 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.

Второе отличие заключается в величине максимальной степени внутреннего параллелизма. При геометрическом параллелизме она равна n, а при конвейерном — $\min(n, k)$. Соответственно, и максимально достижимое ускорение (в предположении нулевых затрат на передачу данных) в случае геометрического параллелизма равно n, а в случае конвейерного параллелизма равно $\min(n, k)$.

Конвейерный параллелизм часто используется при решении систем линейных алгебраических уравнений (СЛАУ) с помощью итерационных методов, например, с помощью метода верхней релаксации. В этом случае высота стены соответствует числу итераций. Следовательно, при конвейерной организации работ, число одновременно работающих процессоров не может превысить числа итераций, количество которых стремятся снизить, чтобы сократить общее время решения задачи. Таким образом, снижение числа итераций приводит к уменьшению числа эффективно используемых процессоров и снижению максимально достижимого ускорения.

4.4. Метод коллективного решения

Рассмотренные ранее методы эффективны при условии, что как время обработки каждого из узлов сетки, так и производительность каждого процессора известны заранее и не меняются от шага к шагу модельного времени. Возникает закономерный вопрос: как поступить в случае, когда априори нельзя сказать сколько-нибудь определенно ни о соотношении производительности процессоров, ни об относительной трудоемкости разных участков работы?

Метод «коллективного решения» позволяет эффективно организовать работу именно в этом случае. В отличие от методов геометрического и конвейерного параллелизма, накладывающих условие локальности на информационные зависимости между обрабатываемыми данными, метод коллективного решения предназначен для обработки множества не связанных между собой заданий с непредсказуемым временем обработки каждого задания на наборе процессоров с неизвестными, возможно изменяющимися производительностями. Таким образом, иллюстративный пример стены Фокса не вполне удачен для пояснения метода. Целесообразнее говорить об укладке не стены, а паркета, любая плитка которого может быть уложена независимо от остальных. Максимальная степень параллелизма равна общему числу плиток паркета nk (в случае построения стены максимальная степень параллелизма равна n).

В рамках метода не предполагается априорное жесткое распределение работы между процессами. Все процессы разбиваются на два множества. В первом множестве — единственный процесс (управляющий), который обеспечивает динамическое распределение работ. Во втором множестве — все остальные процессы (обрабатывающие), которые обеспечивают непосредственное выполнение работ. Топологией соединения процессов в рамках этой стратегии является звезда (рис. 25).

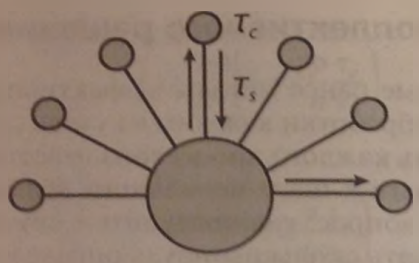


Рис. 25. Коллективное решение

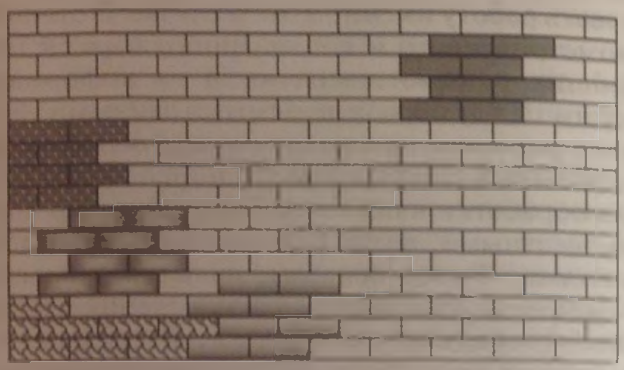


Рис. 26. Паркет

Каждому обрабатывающему процессу управляющий поручает укладку некоторой плитки паркета (рис. 26). Обрабатывающий процесс выполняет задание и вновь обращается к управляющему за получением следующего. В общем случае, каждая вновь полученная для обработки плитка может находиться сколь угодно далеко от ранее обработанных этим процессом плиток. Таким образом, метод коллективного решения является методом *динамической* балансировки загрузки процессоров. Невозможно заранее предсказать, какие именно плитки будут обработаны тем или иным процессом. Это зависит от

того, насколько быстро каждый из них будет выполнять обработку, и определяется непосредственно в ходе решения задачи.

Отметим основные причины, приводящие к простою процессов при таком подходе.

Во-первых, требуется синхронизация между управляющим и обрабатывающими процессами: при выборе номера очередной обрабатываемой плитки и при передаче процессу информации о выбранной плитке. На выполнение этих действий требуется время τ_{s1}, поэтому в начале работы все процессоры, кроме первого, будут простаивать в ожидании своего первого задания.

Во-вторых, каждый выполнивший очередное задание процесс затратит время τ_{s2} на передачу результата управляющему процессу. Если несколько процессов закончат работу одновременно, почти все они также будут находиться в состоянии ожидания готовности управляющего процесса к приему данных.

В-третьих, вероятен простой части процессов в конце работы, когда все задания уже распределены по процессам. Процессы, окончившие работу раньше других, новой работы не получат, и их вычислительная мощность не будет использована в это время для решения задачи.

```
// управляющий процесс
{
for(i=1;i<p;i++)
        Send(i,tasks[i])
for(;i<N;i++)
{
        from=Recv(FROM_ANY,result)
        Send(from,tasks[i])
}
for(i=1;i<p;i++)
{
        from=Recv(FROM_ANY,result)
}
}
```

```
// обрабатывающий процесс
{
for(;;)
{
        Recv(0,task)
        result=calc(task)
        Send(0,result)
}
}
```

Оценим эффективность метода при следующих предположениях:

- все процессоры имеют одинаковую производительность;
- все задания имеют одинаковую трудоемкость, каждое из них обрабатывается любым процессором за время τ_c;
- время на передачу данных не зависит ни от номера задания, ни от номера процесса и составляет $\tau_s = \tau_{s1} + \tau_{s2}$;
- работа выполняется на p обрабатывающих процессах и одном управляющем. Ускорение и эффективность мы будем оценивать относительно числа обрабатывающих процессов, пренебрегая наличием управляющего. Такой подход вполне оправдан с учетом того, что на некотором физическом процессоре могут быть запущены два процесса — управляющий и обрабатывающий. Поскольку управляющий процесс практически не выполняет вычислений, такое совмещение не приведет к замедлению работы.

При сделанных предположениях каждому процессу будет направлено на обработку $\frac{N}{p}$ заданий, где $N = kn$. Время, затраченное на получение данных о каждом задании и на его обработку, составит $(\tau_c + \tau_s)$. Будем считать, $p << N$ и пренебрежем простоями в начале и в конце работы. Обратите внимание, что это значительно более мягкое требование: $p << nk$, по сравнению с предположением, сделанным при обсуждении метода конвейерного параллелизма: $p << k$.

$$T_p = \frac{N}{p}(\tau_c + \tau_s),\ S_p = p\frac{\tau_c}{\tau_c + \tau_s},\ E_p = \frac{1}{1 + \frac{\tau_s}{\tau_c}}.$$

Если $\tau_s \ll \tau_c$, то метод обладает высокой эффективностью. В противном случае следует изменить стратегию распределения заданий, увеличив объем работы, передаваемой об-

рабатывающему процессу при каждом обращении к управляющему процессу. Пусть за одно обращение передается информация о r заданиях (плитках паркета). Назовем задание, содержащее r заданий, расширенным. Тогда время на получение данных и обработку одного расширенного задания составит $r\tau_c + T_s$, где T_s — время передачи данных о r заданиях. Число расширенных заданий, обрабатываемых каждым из процессоров, составит $\frac{N}{rp}$.

$$T_p = \frac{N}{rp}(r\tau_c + T_s),\ S_p = p\frac{1}{1+\frac{T_s}{r\tau_c}},\ E_p = \frac{1}{1+\frac{T_s}{r\tau_c}}.$$

Теперь следует выяснить, чему равно T_s. Эффективность возрастет только при условии, что T_s не зависит от числа элементарных заданий, передаваемых на обработку за одно обращение к серверу (управляющему процессору). Для ряда задач это действительно так. В частности, это справедливо для рассматриваемого примера укладки паркета. В качестве данных, описывающих расширенное задание, можно рассматривать координаты левого нижнего и правого верхнего участка паркета, укладываемого в рамках задания. Тогда, независимо от объема работы, достаточно 4-х чисел для описания участка обрабатываемых данных — координат упомянутых вершин прямоугольной области.

Аналогичная ситуация складывается при вычислении определенного интеграла одномерной функции (рис. 27). Весь отрезок интегрирования можно разбить на фрагменты. Чем длиннее фрагмент, тем больше вычислений потребуется для его обработки, но объем данных, передаваемых между управляющим и обрабатывающим процессором, фиксирован — это три числа: координаты начала и конца отрезка интегрирования и вычисленное значение интеграла. Объем данных не зависит от трудоемкости задания.

Итак, для задач, в которых $T_s = \tau_s$, можно увеличить эффективность, выбирая достаточно большое значение r.

$$T_p = \frac{N}{rp}(r\tau_c + \tau_s), \quad S_p = p\frac{1}{1+\frac{\tau_s}{r\tau_c}}, \quad E_p = \frac{1}{1+\frac{\tau_s}{r\tau_c}}.$$

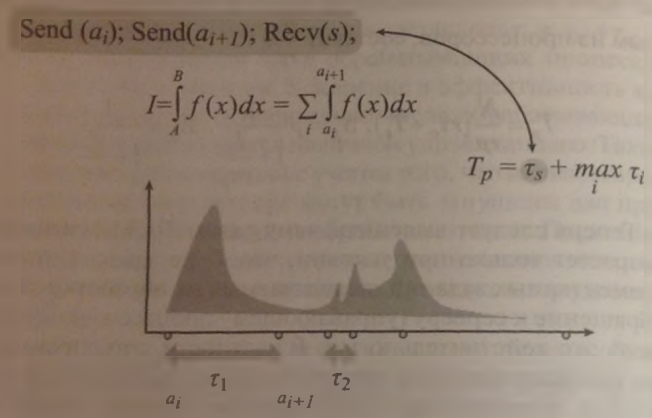


Рис. 27. Вычисление определенного интеграла

Следует иметь в виду, что с ростом величины r, падает степень параллелизма. Расширенных заданий в r раз меньше, чем исходных, поэтому и максимальное число процессоров, использование которых возможно в рамках такого подхода, не может превышать $\frac{N}{r}$, что ограничивает максимально достижимое ускорение. Однако, это не единственная причина ограничения максимально ускорения.

Следует ответить еще на один вопрос: чему равно максимальное число процессоров, использование которых действительно сокращает время решения задачи?

В рассматриваемой схеме присутствует немасштабируемый элемент — единственный управляющий процессор. При большом числе процессов, именно механизм распределения работ оказывается основным фактором, ограничивающим ускорение. Действительно, число процессоров, получивших задания за то время, что самый первый процессор полностью выполнит первое расширенное задание, равно $p_{\max} = \frac{r\tau_c}{\tau_s}$. Таким образом, ни в какой момент времени одновременно не будет работать больше, чем $p_{\max}$ процессоров. Привлечение большего, чем $p_{\max}$ числа процессоров не имеет смысла, поскольку приведет к тому, что первые процессоры, выполнив задания, будут простаивать, пока управляющий процесс будет распределять задания процессорам с большими номерами.

Теперь можно указать, чему равно максимально достижимое ускорение:

$$p_{\max} = \min\left(\frac{r\tau_c}{\tau_s}, \frac{N}{r}\right), \quad S_{p\max} = p_{\max}\frac{1}{1+\frac{1}{p_{\max}}}, \quad E_p = \frac{1}{1+\frac{1}{p_{\max}}}.$$

Формально, максимальное ускорение достигается при $r = \sqrt{N\frac{\tau_s}{\tau_c}}$

$$S_{\max} \approx p_{\max} = \sqrt{N\frac{\tau_c}{\tau_s}}.$$

Эта оценка получена в предположении одинаковости времени обработки каждого из элементарных заданий. Если они равны, то целесообразно использовать статическую балансировку загрузки, изначально распределив задания между процессорами поровну. Тогда, при использовании N процессоров, каждый из которых обрабатывает ровно одно задание, время вычисления с помощью N процессо-

ров частичных сумм равно τ_c. Чтобы каждый процессор мог определить координаты концов назначенного ему фрагмента, достаточно передать ему координаты концов всего отрезка, так как на каждом процессоре известны общее число процессоров и номер самого процессора. Распределение всем процессам одной и той же информации о координатах концов отрезка и финальное сложение частичных сумм потребует времени порядка $\tau_c \log N$. Общее время вычислений составит $\tau_A + \tau_c \log N$, а максимальное ускорение $S_{p=N} = \frac{N\tau_A}{\tau_A + \tau_c \log N} = N \frac{1}{1 + \frac{\tau_c}{\tau_A} \log N}$, что при больших N много больше, чем $\sqrt{N \frac{\tau_A}{\tau_c}}$.

Метод коллективного решения необходим именно тогда, когда время обработки заданий различно и априори неизвестно. Тогда, при использовании больших значений величины r может возникнуть ситуация, в которой некоторому процессору назначено для обработки расширенное задание, содержащее большое число трудоемких элементарных заданий. В результате, возможен существенный простой всех остальных процессоров в конце работы. Они могли бы принять участие в обработке сложных заданий, если бы было выбрано меньшее значение r. Таким образом, конкретное значение r при распределении каждого очередного задания следует выбирать, опираясь на дополнительную информацию о свойствах решаемой задачи и о числе еще не обработанных заданий.

4.5. Причины потери эффективности

До сих пор внимание было сосредоточено на изучении возможности распараллеливания основной вычислительной работы — вычислительного ядра алгоритма решения

некоторой задачи. Однако, кроме вычислительного ядра, есть другие, не менее важные с практической точки зрения действия, выполнение которых также требует определенного, иногда немалого, времени. Значимыми представителями такого рода действий являются ввод исходных данных и вывод результатов вычислений. Типичная схема моделирования некоторого физического процесса выглядит следующим образом:

```
for(step=0;step<100 000;step++)
{
// вычисления

// обмен данными между процессорами

if( step кратен 1000 )
{
// Вывод промежуточных результатов
}
}
```

Ранее рассмотрены особенности параллельного выполнения этапов «*вычисления*» и «*обмен данными между процессорами*», но этап «*вывод промежуточных результатов*» никак не учитывался. Как правило, операции ввода-вывода больших массивов данных требуют больших затрат времени. Далеко не всегда эти затраты могут быть сокращены за счет использования нескольких процессоров. Например, вывод всех значений некоторой сеточной функции в один файл может привести к необходимости последовательного обращения множества процессоров к одному файловому серверу. Число каналов доступа к такому серверу не масштабируется, либо масштабируется значительно хуже, чем число используемых в расчете процессоров. Некоторые методы сокращения подобных затрат обсуждаются в последующих главах. Сейчас отметим только то, что они увеличивают общее время реше-

ния задачи, тем самым дополнительно снижая эффективность использования вычислительной мощности.

Выделим основные причины потери эффективности параллельных алгоритмов:

- отсутствие достаточного ресурса внутреннего параллелизма на каждом из этапов решения задачи;
- дисбаланс вычислительной нагрузки процессоров;
- наличие операций, отсутствующих в наилучшем последовательном алгоритме:
  - дублирование операций;
  - операции передачи данных между процессами;
  - операции синхронизации процессов;
  - новые операции, обусловленные спекулятивными методами выполнения операций;
  - применение алгоритмов, основанных на иных, чем наилучший последовательный алгоритм, вычислительных методах;
- ограниченность в системах с общей оперативной памятью пропускной способности каналов доступа процессоров к оперативной памяти.

Подводя итог, отметим, что рассмотренные методы сдваивания, геометрического и конвейерного параллелизма, коллективного решения отличаются логической простотой, обеспечивающей возможность эффективного отображения широкого круга алгоритмов на большое число исполняющих устройств различной архитектуры.

Глава 5

Сортировка данных

В настоящей главе рассматривается ряд известных алгоритмов сортировки данных: пирамидальная сортировка, сортировка методом слияния, обменная сортировка со слиянием Бэтчера. На их основе разрабатывается «наилучший» с точки зрения времени выполнения последовательный алгоритм и параллельный алгоритм, позволяющий эффективно выполнять сортировку большого количества данных, объем которых ограничен лишь совокупной оперативной памятью многопроцессорной вычислительной системы.

5.1. Постановка задачи

Сортировкой, или упорядочиванием объектов, называется процесс их расположения согласно определенному линейному отношению порядка, такому как отношение «меньше или равно» для чисел.

Самые простые последовательные алгоритмы сортировки, основанные на сравнении элементов, имеют сложность порядка $O(n^2)$ операций, где n — число сортируемых элементов, что неприемлемо при больших значениях n. Лучшие из алгоритмов, основанных на сравнениях и перестановках элементов, обеспечивают сортировку за $O(n\log_2 n)$ операций. Именно эти методы взяты за основу «наилучшего» последовательного алгоритма сортировки. Известны методы поразрядной сортировки, обеспечивающие выполнение сортировки за $O(nk)$ операций, где k — разрядность сортируемых данных, но их реализация существенно зависит от типа сортируемых данных, и в настоящей главе они не рассматриваются.

В главе освещаются два основных вопроса:

- построение последовательного алгоритма, обеспечивающего время сортировки данных, не превышающее $Kn\log_2 n$ в широком диапазоне изменения n. Основное внимание уделяется уменьшению значения K;
- разработка на основе сетей сортировки и созданного последовательного алгоритма масштабируемого эффективного параллельного алгоритма.

Привлечение сетей сортировки обусловлено тем, что быстрые последовательные алгоритмы сортировки плохо поддаются распараллеливанию.

Для упрощения изложения далее рассматривается задача расположения в порядке неубывания n элементов массива целых 32-разрядных чисел. Если не оговорено иное, будем считать, что сортировка выполняется на вычислительной системе с распределенной памятью. Данное предположение, благодаря потенциальной возможности обработки произвольно большого набора данных, существенно расширяет область применимости рассматриваемых методов.

Пусть $T(n, p)$ — общее время сортировки массива из n элементов на p процессорах. Уточним, что будет пониматься под упорядоченным массивом. Будем предполагать, что исходный массив распределен по процессорам некоторыми порциями A_i, где i — номер процессора. По окончании сортировки элементы исходного массива распределены по процессорам некоторыми порциями B_i, каждая из которых упорядочена. Потребуем, чтобы размеры всех порций были одинаковы. Как будет показано, только при выполнении этого условия, рассматриваемые далее параллельные алгоритмы, могут быть использованы для сортировки массивов. В случае, если число элементов сортируемого массива не делится нацело на число процессоров, можно дополнить исходный массив фиктивными элементами, исключив их после окончания сортировки. Таким образом, далее полагаем,

что размеры исходных и отсортированных порций совпадают между собой: $|A_i| = n/p$, $|B_i| = n/p$.

Пусть B_i^k — k-ый элемент фрагмента массива, расположенного на процессоре i. *Массив B упорядочен*, если выполняются следующие соотношения:

$B_i^k \le B_i^j$, для любых i, при $k < j$;

$B_i^k \le B_j^l$, для любых j, k, при $i < l$.

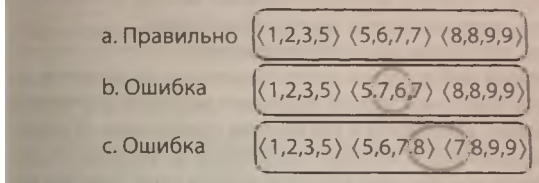


Рис. 28. Распределенный массив, $n = 12$, $p = 3$

На рисунке 28 в строке *a* показан пример упорядоченного массива, а в строках *b* и *c* — неупорядоченного. В строке *b* неупорядочен фрагмент массива на втором процессоре. В строке *c* на втором процессоре содержится элемент, имеющий большее значение, чем наименьший элемент на третьем процессоре.

Рассматриваемые далее параллельные алгоритмы предполагают выполнение сортировки в два этапа:

- сортировка каждого из распределенных по процессорам фрагментов массива;
- совместная обработка уже упорядоченных фрагментов массива.

Для уменьшения общего времени выполнения сортировки следует сократить время выполнения каждого из этих этапов. В связи с этим, будет рассмотрен последовательный алгоритм, обеспечивающий «наименьшее» время работы. Именно относительно него будет определяться эффективность алгоритмов параллельной сортировки.

Время сортировки массива зависит от множества факторов, среди которых не последнее место занимает характер распределения чисел в сортируемом массиве (степень его упорядоченности перед началом сортировки). При тестировании использовалось несколько наборов тестовых данных — массивов длины n, заполненных числами, распределенными следующими способами:

- псевдослучайные неупорядоченные числа;
- числа, расположенные по невозрастанию;
- числа, расположенные по неубыванию.

Указанный набор тестов исчерпывающим не является, но с практической точки зрения позволяет выделить достаточно эффективные алгоритмы и процедуры сортировки. В экспериментах, результаты которых приведены на рисунке 29, использовались все три типа наборов данных, при этом для построения графиков использовались результаты, полученные в наихудших, с точки зрения времени выполнения, случаях.

Подробное описание обсуждаемых далее алгоритмов можно найти в монографии [6]. Там же приведен анализ алгоритмов и оценка числа выполняемых операций.

5.2. Последовательные алгоритмы сортировки

В таблице 3 приведены асимптотические оценки зависимости количества операций последовательных алгоритмов $M(n, p = 1)$ от размера n сортируемого массива [6].

Таблица 3

Зависимость количества операций от размера массива [1]

Алгоритм сортировки	Среднее число операций	Максимальное число операций
Быстрая (qsort) [Алгоритм Q, 6]	$11.7n \log_2 n$	$O(n^2)$

Окончание табл. 3

Алгоритм сортировки	Среднее число операций	Максимальное число операций
Слияние списков (lsort) [Алгоритм L, 6]	$9n \log_2 n$	$O(n \log_2 n)$
Простое двухпутевое слияние (dsort) [Алгоритм S, 6]	$11n \log_2$	$O(n \log_2 n)$
Пирамидальная (hsort)	$16n \log_2 n + 0.01n$	$18n \log_2 n + 38n$

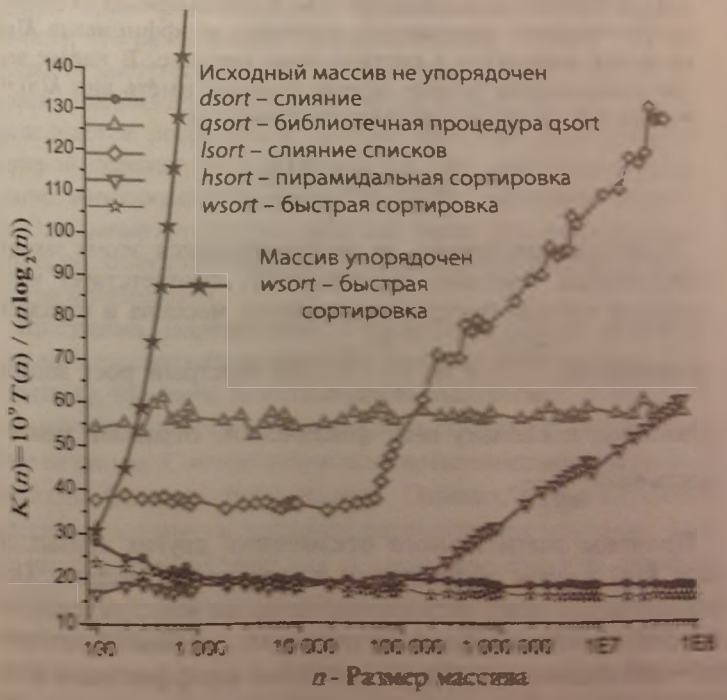


Рис. 29. Сортировки массива на Pentium IV, 2.4 GHz, 2 GByte ОП

Дальнейший анализ опирается на результаты, полученные при тестировании ряда программно реализованных алгоритмов, для каждого из которых экспериментально фиксировалось время выполнения. По результатам тестирования проверялась гипотеза соответствия времени работы алгоритмов зависимости $T(n) = 10^{-9}K(n)n\log_2 n$, поскольку рассматриваемые алгоритмы имеют оценку числа выполняемых операций $O(n\log_2 n)$. Дополнительный множитель 10^{-9} выбран потому, что характерное время выполнения одной операции при частоте процессора, измеряемой гигагерцами, составляет доли наносекунды. Каждая кривая на рисунке 29 отражает зависимость значения коэффициента $K(n)$ от числа элементов в сортируемом массиве. В идеале все эти зависимости (кроме одной) должны иметь вид $K(n) = \text{const}$ и быть горизонтальными прямыми.

$$K(n) = 10^9 \frac{T(n)}{n\log_2 n}.$$

Единственная кривая, не подчиняющаяся этому закону (отмеченная темными звездочками), соответствует наихудшему случаю быстрой сортировки массива и отражает зависимость $\frac{T(n)}{n^2}$, а не $\frac{T(n)}{n\log_2 n}$. Ее быстрый рост вполне объясним, поскольку она, фактически, отражает зависимость вида $\frac{n}{\log_2 n}$.

Причины значительного отклонения других кривых от вида $K(n) = \text{const}$ обсуждаются в следующих разделах. Основной целью этого обсуждения является конструирование алгоритма, выполняющего сортировку за наименьшее время — имеющего наименьшее значение коэффициента $K(n)$. Минимизация $K(n)$ позволит создать алгоритм, в котором $K(n) \sim \text{const}$.

5.2.1. Быстрая сортировка (runtime qsort, wsort)

Применение для сортировки больших массивов популярной и доступной в стандартной поставке компиляторов процедуры быстрой сортировки (***runtime qsort***) нецелесообразно по двум причинам.

Во-первых, время выполнения работы ***runtime qsort*** больше, чем время работы других рассматриваемых алгоритмов даже в среднем случае (рис. 29, кривая Qsort). Вероятно, это является платой за универсальность интерфейса. Процедура ***runtime qsort*** позволяет обрабатывать данные любого типа, что реализовано неэффективным, с точки зрения скорости доступа к элементам сортируемого массива, способом. Например, каждая операция сравнения элементов выполняется посредством вызова функции пользователя, которой передаются два параметра, что значительно медленнее непосредственного доступа к двум элементам массива. Остальные алгоритмы работают более чем в два раза быстрее, поэтому стандартная библиотечная функция дальше не рассматривается.

Программа, сортировки 32-разрядных чисел, основанная на классическом варианте быстрой сортировки [Алгоритм Q, 6], ***wsort,*** действительно выигрывает (в среднем случае) почти у всех сравниваемых программ, оправдывая свое название. Соответствующая зависимость на рисунке 29 показана белыми звездочками. Однако, при сортировке массивов, элементы которых расположены неудачно, время выполнения растет, как и следовало ожидать, пропорционально n^2 (уходящая вверх помеченная темными звездочками кривая на рис. 29), что не позволяет выполнять за приемлемое время сортировку произвольных больших массивов. Это вторая причина отказа от использования этого алгоритма в общем случае.

5.2.2. Простое двухпутевое слияние (dsort) и слияние списков (lsort)

Согласно таблице 3, в среднем случае, наилучшим выбором для последовательного алгоритма будет алгоритм слияния списков (*lsort*). Однако, результаты выполнения программ, написанных на основе этого алгоритма и алгоритма простого двухпутевого слияния (*dsort*), показывают обратную картину: алгоритм *dsort* при больших значениях n работает значительно быстрее.

Укажем две причины, по которым возможны подобные расхождения:

- зависимость времени выполнения алгоритма от особенностей конкретной программной реализации, в том числе от системы команд используемого процессора;
- ограниченная применимость асимптотических оценок.

Первая причина обусловлена тем, что константы, полученные в монографии Дональда Кнута, соответствуют вполне конкретным реализациям алгоритмов сортировки (программам), написанным для вполне конкретной машины MIX [17]. Программы, используемые при построении графиков рисунка 29, отличаются от реализаций Дональда Кнута, поскольку написаны на языке высокого уровня и выполняют иную последовательность операций на машине, обладающей другим набором инструкций. Таким образом, само число выполненных инструкций может на практике отличаться от оценок [6].

Вторая причина проиллюстрирована на рисунке 30. Асимптотическая оценка вида $T(n) = O(f(n))$ не несет информации о том, начиная с какого n, можно при оценке величины $T(n)$ ориентироваться на значения функции $f(n)$. При одном и том же асимптотическом поведении кривых f и g, соотношение их значений в точке n_0 отличается от соотношения их значений при больших n.

Несмотря на то, что, начиная с некоторого n, выполняется соотношение $f(n) < g(n)$, при $n = n_0$ картина обратная. К зна-

чению функций f и g в окрестности точки n_0 асимптотика отношения не имеет. Ориентироваться на нее при оценке значений функций около этой точки не следует.

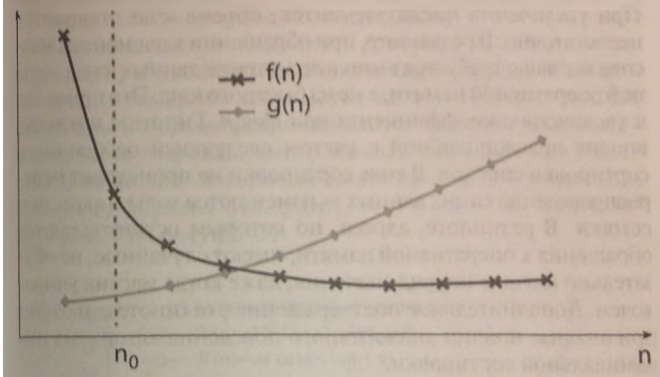


Рис. 30. Сравнение функций

Именно такая ситуация складывается при сортировке больших объемов данных. Асимптотические оценки времени сортировки, показывая, как будет меняться время работы алгоритмов при обработке бесконечно больших объемов данных, совершенно не отражают ситуацию, складывающуюся при обработке тех объемов данных, которые могут быть отсортированы с учетом фактически доступной оперативной памяти. Характерный объем оперативной памяти современных процессорных узлов недостаточно велик для того, чтобы поведение алгоритмов соответствовало асимптотическим оценкам (рис. 29).

Рассмотрим, например, поведение $K(n)$ для алгоритма слияния списков (кривая L-sort на рис. 29). Обращает на себя внимание нелинейный характер этой зависимости. Вплоть до 100 000 элементов значение этого коэффициента примерно постоянно, но потом наблюдается значительный рост. Веро-

ятной причиной этого является влияние архитектурных особенностей используемого процессора. При небольшом числе элементов весь массив размещен в кэш-памяти процессора — промежуточном буфере с большой скоростью доступа. При увеличении числа элементов, объема кэш становится недостаточно. В результате, при обращении к элементам массива все чаще требуется выполнить чтение данных из медленной оперативной памяти, а не из быстрого кэш. Это приводит к увеличению коэффициента при $n\log_2 n$. Гипотеза выглядит вполне правдоподобной с учетом следующей особенности сортировки списков. В ходе сортировки не происходит перераспределения самих данных — изменяются только адресные ссылки. В результате, адреса, по которым осуществляются обращения к оперативной памяти, имеют случайные, не обязательно идущие подряд значения, даже когда массив упорядочен. Дополнительное подтверждение эта гипотеза находит при анализе причин аналогичного поведения алгоритма пирамидальной сортировки.

Алгоритм простого двухпутевого слияния ***dsort*** лишен подобного недостатка. Он показывает лучшие, по сравнению с алгоритмом слияния списков, результаты, поэтому в дальнейшем при обработке больших массивов используется именно он. При больших значениях числа элементов, алгоритм ***dsort*** показывает наименьшее значение константы $K(n)$, но при малых значениях n, пирамидальная сортировка ***hsort*** выигрывает по этому параметру. Далее подробнее рассмотрены свойства и структура этих двух алгоритмов с целью создания на их основе наилучшего последовательного алгоритма ***dhsort***.

5.2.3. Пирамидальная сортировка (hsort)

На рисунке 31 приведены кривые зависимостей

$$K(n)=10^9\frac{T(n)}{n\log_2 n}\ \text{и}\ R(n)=\frac{M(n)}{n\log_2(n)},$$

где $T(n)$ — время выполнения алгоритма, $M(n)$ — число выполненных операций. Кривые получены при выполнении

алгоритма пирамидальной сортировки ***hsort***. Аналогично кривой $K(n)$, показывающей характер изменения *времени* сортировки, зависимость $R(n)$ показывает характер изменения *числа операций*, выполненных при сортировке.

Численный эксперимент подтверждает, что $R(n)$ действительно является константой (рисунок 31). Под *числом операций* при экспериментальном определении $R(n)$ понимается суммарное количество выполненных операций сравнения и перемещения элементов массива. Таким образом, *число операций*, выполняемых при сортировке массива с помощью алгоритма пирамидальной сортировки, составляет порядка $O(n\log_2(n))$, $R(n) \approx 2{,}7$ — горизонтальная прямая на рисунке 31.

Рис. 31. Пирамидальная сортировка. Число операций и время выполнения

Разница в числовых значениях $R(n)$, полученных в эксперименте и приведенных в таблице 3 ($R'(n) \approx 16$), вполне объяснима тем, что, кроме операций сравнения и перемещения элементов, в алгоритме выполняется ряд иных действий, число которых оценивается в [6], но не подсчитывается в проведенном эксперименте.

Совершенно иначе выглядит кривая зависимости *времени* выполнения пирамидальной сортировки от размера массива (рис. 31). При обработке массивов, число элементов в которых не превышает 10^5, наблюдается хорошее соответствие времени выполнения ожидаемой оценке (время измерено в секундах).

$$T(n) \approx K(n)10^{-9} n \log_2(n),$$
$$K(n) \approx 21.$$

Однако, при дальнейшем увеличении размера сортируемого массива (с 10^5 до 10^8 элементов), коэффициент $K(n)$ возрастает более чем в три раза — с 21 до 65. Несмотря на то, что *число операций* возрастает пропорционально $n\log_2(n)$, *время* выполнения растет быстрее чем $n\log_2(n)$.

5.3. Свойства последовательных алгоритмов

Рассмотрим подробнее свойства двух конкурирующих с точки зрения производительности последовательных алгоритмов — алгоритма простого двухпутевого слияния и алгоритма пирамидальной сортировки. У первого меньшее время обработки больших массивов, а у второго — массивов небольшого размера.

5.3.1. Сортировка методом простого двухпутевого слияния

Алгоритм простого двухпутевого слияния выполняет упорядочивание массива из n элементов за $O(n\log_2 n)$ действий (алгоритмы A18, A19). Он основан на идее слияния упорядоченных фрагментов массива и может быть описан как в рекурсивным (A18), так и в нерекурсивным образом (A19):

Алгоритм A18

Рекурсивный алгоритм сортировки слиянием

```
сортировать ( массив mas, число элементов n)
 {
 если (n > 1)
 {
 сортировать ( mas, n/2); // сортировка первой половины
                          // массива
 сортировать ( mas+n/2, n-n/2); // сортировка второй
                                // половины массива

 слияние (mas, n/2, mas+n/2,n-n/2); // объединить
                    // отсортированные части массива
 }
 }
```

Непосредственное использование рекурсии в алгоритме A18 требует значительных накладных расходов, обусловленных многократным вложенным вызовом функции ***сортировать***. Нетрудно вычислить, что общее число вызовов составит $2+4+...+\frac{n}{2}=n-2$. Этих потерь можно избежать, используя нерекурсивную версию алгоритма (алгоритм A19). В нерекурсивной версии алгоритма для хранения промежуточных результатов используется предварительно распределенный массив *array_second*, а копирование данных из дополнительного в основной массив выполняется не более одного раза — при окончательном размещении результата — что значительно сокращает время обработки.

Алгоритм A19

Нерекурсивный алгоритм слияния dsort

```
Dsort(intsort *array, int n)
{
 a=array;    // сортируемый массив
 b=array_second; // вспомогательный массив

 for(i=1;i<n;i=i*2) // размер объединяемых фрагментов
```

```
    // начало первого
                    // из объединяемых фрагментов
    }
   r=j+i; // начало второго из объединяемых фрагментов

   n1=min(i,n-j);
   n2=min(i,n-r);
   if(n1<0) n1=0;
   if(n2<0) n2=0;

   // слияние упорядоченных фрагментов
   for(ia=0,ib=0,k=0;k<n1+n2;k++)
    {
     if(ia>=n1) b[j+k]=a[r+ib++];
     else
     if(ib>=n2) b[j+k]=a[j+ia++];
     else
     if(a[j+ia]<a[r+ib]b[j+k]=a[j+ia++];
     else    b[j+k]=a[r+ib++];
     }
   }

  c=a;a=b;b=c;
  }

 c=a;a=b;b=c;
// копирование, если результат размещен не в основном,
// а во вспомогательном массиве
 if(b!=array)
  memcpy(array,b,n*sizeof(intsort));
}
```

Параллельная сортировка сдваиванием

Нерекурсивный алгоритм ***dsort*** допускает параллельную обработку на основе метода сдваивания. Проиллюстрируем идею на примере сортировки массива из $n = 16$ элементов на $p = 8$ процессорах (рис. 32).

Вначале на каждом из процессоров одновременно упорядочиваются размещенные на них массивы из $\frac{n}{p}$ элементов каждый, на что требуется $\frac{n}{p}\log_2\frac{n}{p}$ операций.

На последующих шагах с номерами $j \geq 1$:

- процессор с номером $2^j i$ передает процессору с номером $2^{j-1}(2i-1)$ отсортированный фрагмент массива длиной $2^{j-1}\frac{n}{p}$, что требует времени, необходимого для передачи $\frac{n}{p}\cdot 2^{j-1}$ чисел;
- процессоры с номерами $2^j i$ выполняют слияние пар фрагментов, длинной $\frac{n}{p}\cdot 2^{j-1}$, что требует $\frac{n}{p}\cdot 2^j$ операций.

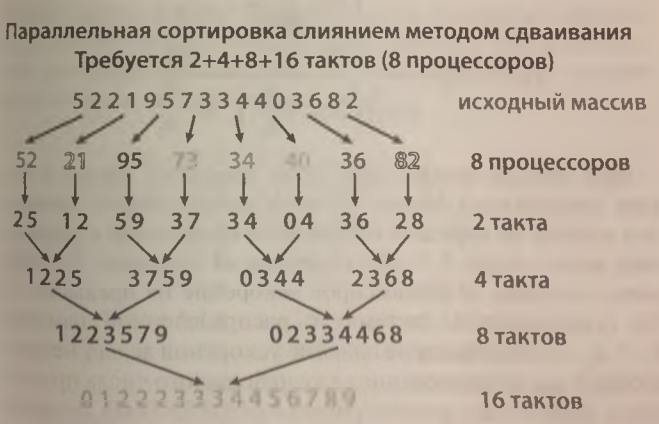


Рис. 32. Порядок выполнения слияния фрагментов массива методом сдваивания на восьми процессорах

Таким образом, общее время выполнения сортировки, при $p = 2^k$ и n кратном p составит:

$$T(n,p)=\frac{n}{p}\left(k_1\log_2\left(\frac{n}{p}\right)+k_2(p-1)+2k_3(p-1)\right).$$

Положим $k_1 \approx k_3$, поскольку на соответствующих шагах выполняются сходные операции слияния массивов.

Таким образом, ожидаемое ускорение равно:

$$S(n,p)=\frac{p}{1+\left(2(p-1)-\log_2 p+\frac{k_2}{k_1}(p-1)\right)\log_n 2}.$$

Определим ускорение при сортировке миллиарда элементов на 4-х и на 32-х процессорах.

$$S\left(10^9;4\right)\approx\frac{4}{1{,}13+\frac{1}{30}\frac{k_2}{k_1}}<3{,}5;$$

$$S\left(10^9;32\right)=\frac{32}{1+\frac{1}{30}\left(56+31\frac{k_2}{k_1}\right)}\approx\frac{32}{3+\frac{k_2}{k_1}}<10.$$

При четырех процессорах, даже имеющих доступ к общей оперативной памяти (в этом случае можно считать, что затраты на передачу массивов от процессора к процессору равны нулю, $k_2 = 0$), ускорение не превысит 3,5. При использовании 32 процессоров ускорение не превысит 10. На существующих системах с распределенной памятью $k_2 >> k_1$, поэтому быстрое падение ускорения делает нецелесообразным использование даже небольшого числа процессоров. Кроме того, рассмотренный параллельный алгоритм не позволяет обработать массив, размер которого превышает оперативную память одного процессорного узла, поскольку заключительное слияние двух половин всего массива выполняется на одном процессоре.

Таким образом, сортировка, основанная на распараллеливании простого двухпутевого слияния методом сдваива-

ния, проста в реализации, но эффективна только при обработке массивов ограниченного размера на небольшом числе процессоров, объединенных общей памятью.

Алгоритм двустороннего слияния

На основе алгоритма слияния можно предложить эффективный немасштабируемый алгоритм, выполняющийся строго на двух имеющих доступ к общей оперативной памяти процессорах.

Для слияния на одном процессоре двух упорядоченных массивов по 8 элементов потребуется 16 тактов. При этом единственный процессор выполняет обработку с младших номеров элементов к старшим, выбирая на каждом такте тот из двух элементов, что имеет меньшее значение. Несколько первых тактов работы такого алгоритма показаны на рисунке 33. В двух левых столбцах приведены еще не обработанные элементы двух массивов, а в правом столбце — первые элементы результирующего массива.

```
12235579  02334468  0
12235579  02334468  01
12235579  02334468  012
12235579  02334468  0122
12235579  02334468  01222
12235579  02334468  012223
12235579  02334468  0122233
12235579  02334468  01222333
...
```

Рис. 33. Первые такты слияния одним процессором

В двухпроцессорной системе с общей памятью второй процессор может одновременно с первым обрабатывать массивы, начиная с элементов со старшими номерами, вы-

бирая на каждом такте элемент с максимальным значением. Параллельное двустороннее слияние (рис. 34) двумя процессорами потребует 8-ми тактов, тем самым обеспечив двукратное ускорение и стопроцентную эффективность.

1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0	9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1	8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2	7 8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2 2	6 7 8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2 2 2	5 6 7 8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2 2 2 3	5 5 6 7 8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2 2 2 3	4 5 5 6 7 8 9
1 2 2 3 5 5 7 9	0 2 3 3 4 4 6 8	0 1 2 2 2 3 3 4 4 5 5 6 7 8 9	

Рис. 34. Двустороннее слияние массивов двумя процессорами

Приведенные на рисунках 33, 34 примеры позволяют сделать важный вывод. Каждый из процессоров выполняет обработку элементов данных, расположенных в оперативной памяти последовательно. Пусть в каждой линии кэш-памяти процессора помещается *m* элементов сортируемых данных.

Перечислим две цепочки адресов, по которым были расположены считываемые первым процессором элементы массивов (рис. 35).

1 2 3 4 5 ...

9 10 11 12 13 ...

Рис. 35. Адреса считываемых элементов

В результате последовательного доступа к элементам массивов, количество обращений к медленной оперативной памяти у каждого процессора будет в *m* раз меньше, чем при хаотичных обращениях к оперативной памяти. При считы-

вании элемента с номером, кратным m, в линию кэш попадут следующие за ним $m - 1$ элементы. При их обработке дополнительный доступ к оперативной памяти не понадобится. Таким образом, алгоритм является эффективным с точки зрения использования кэш. Именно этим объясняется отсутствие у алгоритма ***dsort*** увеличения множителя при $n\log n$ с ростом объема обрабатываемых данных. Совершенно иначе обстоит дело при использовании алгоритма пирамидальной сортировки.

5.3.2. Пирамидальная сортировка

Пирамидальная сортировка также является методом, быстродействие которого оценивается как $O(n\log_2 n)$. В ее основе лежит специальная структура данных — упорядоченная пирамида, позволяющая:

- добавлять новый элемент за $O(\log_2 n)$ операций;
- выбирать элемент, имеющий максимальное значение за $O(1)$ операций;
- удалять выбранный элемент с сохранением свойств структуры за $O(\log_2 n)$ операций.

Назовем пирамидой сбалансированное бинарное дерево (АВЛ-дерево [18]), в котором никакой правый потомок не выше левого потомка. Пусть высота пирамиды равна k. Тогда высота любого потомка равна либо k, либо $k - 1$. Уровень $k - 1$ полностью заполнен, а на уровне k заполнена его левая часть. Указанные свойства предоставляют простой и эффективный способ хранения пирамиды (рисунки 36, 37).

Любой одномерный массив является пирамидой. Никаких дополнительных данных, кроме размера массива, не требуется для определения связей между узлами пирамиды. Элементы массива пронумерованы последовательностью натуральных чисел. Корень пирамиды хранится в элементе a_1. Потомки любого узла с номером i имеют номера $2i$ и $2i + 1$. Левый и правый потомки элемента a_i хранятся, соответственно, в элементах a_{2i} и a_{2i+1}.

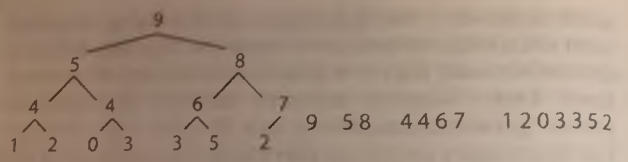


Рис. 36. Упорядоченная пирамида

Рис. 37. Линейное представление пирамиды (рис. 36)

Назовем пирамиду упорядоченной, если значение, хранящееся в любом корне не меньше, чем значения, хранящиеся в каждом из его потомков. В упорядоченной пирамиде для любых допустимых значений i справедливы соотношения:

$$a_i \geq a_{2i+1} \text{ и } a_i \geq a_{2i+2}.$$

Пирамида, показанная на рисунках 36, 37, удовлетворяет этому свойству. Подробно процесс построения упорядоченной пирамиды и полное описание алгоритма пирамидальной сортировки можно найти в [6].

Пирамидальная сортировка имеет две фазы: построение пирамиды и формирование результата. Определим характер обращений к оперативной памяти в ходе выполнения пирамидальной сортировки. Поскольку характер обращений к оперативной памяти сходен для обеих фаз, рассмотрим только вторую из них.

Согласно свойствам упорядоченной пирамиды, в ее корне всегда находится элемент, имеющий наибольшее значение. Отсюда вытекает алгоритм фазы 2:

1. Поменять местами первый и последний элементы массива $a[1] \leftrightarrow a[n]$.
2. Уменьшить число элементов в оставшейся пирамиде $n = n - 1$.
3. Восстановить упорядоченность оставшейся пирамиды.
4. Перейти к шагу 1, если $n > 1$.

Очевидно, что в результате выполнения первого шага, в конец пирамиды, а значит, в начало сформированного фрагмента массива, каждый раз попадает максимальный из оставшихся в пирамиде элементов. Требуемая упорядоченная последовательность формируется, начиная с элементов, имеющих большие значения (рис. 38).

Перестановка первого и последнего элементов может приводить к тому, что значение корня станет меньше, чем значение одного из его потомков, но упорядоченность левой и правой пирамид при этой операции сохраняется. Для восстановления упорядоченности всей пирамиды достаточно установить на нужное место элемент, расположенный в корне, проведя его вниз по цепочке ребер до требуемого уровня. Для этого потребуется не более чем $\log_2 n$ раз поменять местами элемент в текущем корне с наибольшим из потомков, переходя каждый раз на уровень ниже до тех пор, пока перемещаемое значение меньше значения одного из потомков. Расстояния между элементами, инцидентными каждому из проверяемых ребер, различаются не менее чем в два раза: i и $2i$ или i и $2i + 1$, а значит, в ходе этого процесса обращения к оперативной памяти носят весьма хаотичный характер. Это хорошо видно на рисунке 38. Перечислим адреса, по которым осуществлялись обращения при выполнении указанных на этом рисунке действий:

1 14 1 3 3 7 1 13 1 3...

В отличие от алгоритма простого двухпутевого слияния A19, пирамидальная сортировка не отличается эффективностью использования кэш-памяти — обращений к расположенным рядом элементам массива практически нет. В результате, при превышении размером массива объема доступной кэш-памяти, производительность алгоритма преимущественно определяется временем доступа к медленной оперативной памяти, что и объясняет значительный, не соответствующий оценке $O(n\log_2 n)$ рост времени выполнения.

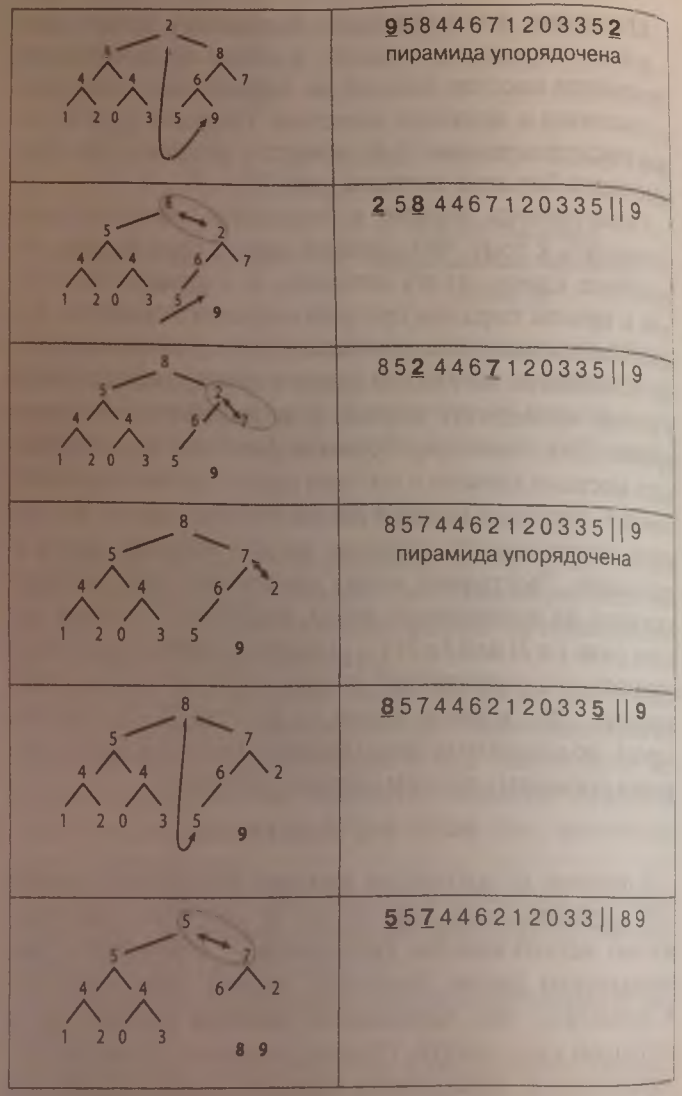


Рис. 38. Фаза 2 алгоритма пирамидальной сортировки ***hsort***

5.3.3. Наилучший последовательный алгоритм сортировки dhsort

К настоящему моменту в нашем распоряжении есть два алгоритма, один из которых — ***hsort*** — эффективен при обработке коротких массивов, а другой — ***dsort*** — при обработке длинных (рисунок 40а). Очевидное дальнейшее развитие этих алгоритмов заключается в предварительной сортировке алгоритмом ***hsort*** коротких фрагментов массива, а затем — в слиянии алгоритмом ***dsort*** уже подготовленных упорядоченных фрагментов (рис. 39).

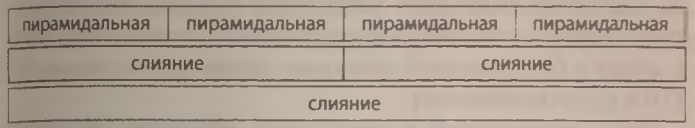


Рис. 39. Схема оптимальной сортировки

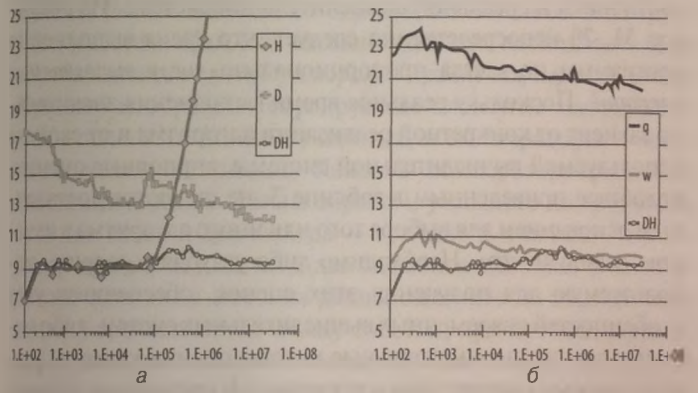


Рис. 40. Сортировки массива

Полученная таким образом процедура ***dhsort*** затрачивает наименьшее, по сравнению с остальными рассмотренными процедурами, время (рис. 40а, кривая DH) в широком диа-

пазоне объемов сортируемых данных. Это позволяет рассматривать ее в качестве «наилучшего последовательного алгоритма». Пороговое значение — максимальный размер блока, сортируемого пирамидальной сортировкой, определяется экспериментально. При обработке коротких массивов алгоритм ***dhsort*** совпадает с алгоритмом ***hsort***, а большие массивы обрабатывает быстрее метода ***dsort*** за счет более эффективной обработки коротких массивов. Отметим, что разработанный алгоритм не уступает по производительности обсуждаемой ранее процедуре быстрой сортировки 32-разрядных чисел ***wsort***. На рисунке 40б показаны результаты тестирования при одинаковых условиях процедур ***wsort*** и ***dhsort*** и библиотечной функции ***runtime qsort*** (кривые W, DH и q соответственно).

Итак, приведенный в монографии [6] анализ посвящен оценке числа операций, необходимых для сортировки, – именно *числа операций*, а не *времени выполнения* процедуры сортировки на реальной вычислительной системе. Из рисунков 31, 29 непосредственно следует, что *время* выполнения программы не всегда пропорционально *числу выполняемых операций*. Поскольку реальное время выполнения значительно зависит от конкретной реализации алгоритма и от свойств используемой вычислительной системы, априорные оценки, подобные приведенным в таблице 3, не являются достаточным основанием для выбора того или иного алгоритма в качестве «наилучшего». Необходимо либо углублять теорию, используемую для получения этих оценок, обеспечивая учет особенностей современных вычислительных систем, либо использовать экспериментальные методы оценки качества разрабатываемых алгоритмов и программ. В настоящем разделе на основании экспериментального изучения свойств ряда программ сортировки, сконструирован наилучший последовательный алгоритм ***dsort***. Далее этот алгоритм используется в качестве отправной точки для построения параллельных алгоритмов и для определения их ускорения и эффективности.

5.4. Масштабируемые алгоритмы сортировки

Ранее были рассмотрены два параллельных алгоритма сортировки: слияние методом сдваивания и двустороннее слияние. Оба они не позволяют эффективно использовать большое число процессоров и не обеспечивают возможность обработки данных, размер которых превышает объем оперативной памяти одного процессорного узла.

Рассмотрим подход, основанный на понятии «сеть сортировки». Именно он позволяет создавать масштабируемые параллельные алгоритмы сортировки больших объемов данных.

5.4.1. Сети сортировки

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*. Принято изображать сортируемые элементы массива горизонтальными линиями данных, а компараторы — вертикальными отрезками. Каждый компаратор соединяет две линии данных. Таким образом, 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 41 показана сеть сортировки трех элементов и пример упорядочивания массива (9, 4, 1). Каждому элементу массива соответствует линия данных, значение на которой меняется по мере срабатывания компараторов.

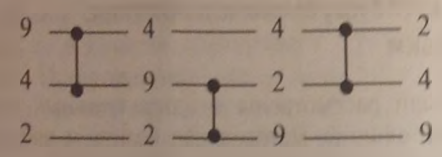


Рис. 41. Сеть сортировки трех элементов

В данном тривиальном случае проверка правильности сети может быть выполнена анализом соответствующего ей алгоритма. Нетрудно увидеть, что это хорошо известный метод простой вставки. Первый компаратор сортирует массив из двух элементов, а оставшиеся два компаратора обеспечивают вставку третьего элемента в нужное место.

Сеть сортировки рассмотренного вида легко наращивать, обеспечивая возможность обработки массива произвольной длины. Для этого достаточно добавить к существующей сети из p элементов (рис. 42) еще одну линию данных снизу и еще одну цепочку из p компараторов справа: $(p+1, p)$, $(p, p-1)$, $(p-1, p-2)$, ..., $(1, 2)$. Добавленные компараторы обеспечат вставку нового элемента в нужное место уже отсортированной части массива из p элементов.

Каждая сеть сортировки характеризуется временем работы — числом шагов, требуемых для выполнения сортировки. Фактически, сеть сортировки задает расписание, согласно которому некоторые компараторы могут срабатывать одновременно. Например, в сети, показанной на рисунке 42, на шаге 5 одновременно могут выполняться компараторы (1, 2), (3, 4) и (5, 6). Благодаря такой параллельной обработке сортировка будет выполнена за 9 шагов.

Представляет интерес вопрос построения сетей, обладающих наименьшим временем работы. Для значений $n \le 10$ известны сети, обладающие минимально возможным временем [6]. В частности, для шести процессоров известна сеть сортировки, выполняющаяся всего за 5 шагов (рис. 43).

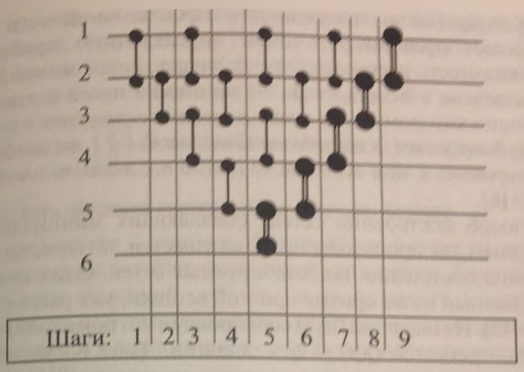


Рис. 42. Сеть сортировки шести элементов

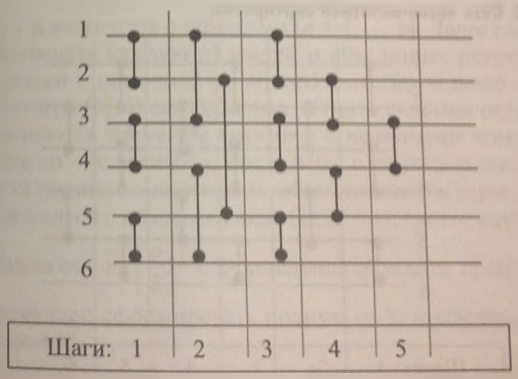


Рис. 43. Наиболее быстродействующая сеть для 6 процессоров [1]

Доказательство правильности произвольной сети сортировки представляет собой определенную проблему. Правильность рассматриваемых дальше сетей может быть установлена с использованием принципа нулей и единиц, согласно которому: *если сеть с p входами сортирует в порядке неубывания все 2^p последовательности из 0 и 1, то она будет сортировать в том же порядке любую последовательность p чисел* [6].

Способ построения сетей, обладающих минимальным временем для произвольного p, неизвестен, но известен ряд методов построения масштабируемых сетей. Один из них, основанный на алгоритме простой вставки, уже рассмотрен (рис. 42). На параллельную сортировку с его помощью p элементов требуется $Q(p) = 2p - 3$ шагов. Далее обсуждаются два других метода с лучшими, чем у рассмотренного, показателями: метод четно-нечетной перестановки (рисунок 44) и метод обменной сортировки со слиянием Бэтчера (рисунки 45, 46).

5.4.2. Сеть четно-нечетной сортировки

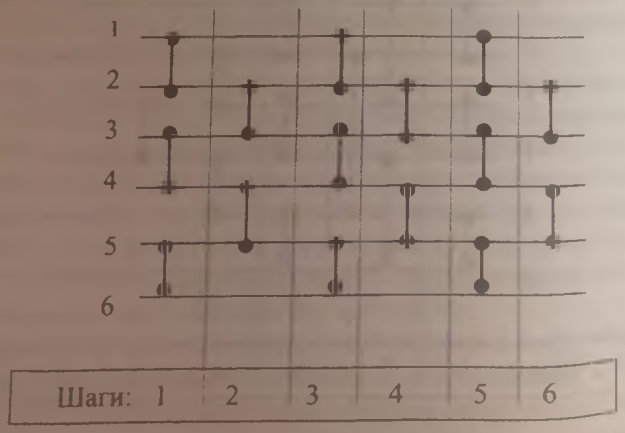


Рис. 44. Сеть четно-нечетной сортировки

Сеть четно-нечетной сортировки состоит из p групп компараторов. Компараторы нечетных групп соединяют пары линий с номерами вида $(2i - 1, 2i)$, а четных — существующие пары с номерами вида $(2i, 2i + 1)$, где i — натуральные числа от 1 до $\left\lfloor \frac{p}{2} \right\rfloor$. Время работы подобной сети $Q(p) = p$.

5.4.3. Сеть обменной сортировки со слиянием Бэтчера

Сети Бэтчера — наиболее быстродействующие из рассматриваемых масштабируемых сетей сортировки. Для построения сети обменной сортировки со слиянием можно использовать описываемый далее рекурсивный алгоритм или приводящий к аналогичному результату нерекурсивный алгоритм [19].

Для сортировки массива, содержащего p элементов с номерами $[1, ..., p]$ следует разделить его на две части. В первой оставить $n = \left\lceil \frac{p}{2} \right\rceil$ элементов с номерами $[1, ..., n]$, а во второй $m = p - n$ элементов с номерами $[n + 1, ..., p]$. Далее следует отсортировать каждую из частей и объединить результаты сортировки с помощью (n, m)-сети нечетно-четного слияния Бэтчера [6]. В сети нечетно-четного слияния отдельно объединяются элементы массивов с нечетными номерами и отдельно — с четными, после чего, с помощью заключительной группы компараторов, обрабатываются пары соседних элементов с номерами вида $(2i, 2i + 1)$, где i — натуральные числа от 1 до $\left\lfloor \frac{p}{2} \right\rfloor - 1$. Формальное описание процедуры, позволяющей сформировать полную сеть, приведено в алгоритме А20.

Общая структура алгоритма формирования сети сортировки массива длинны p приведена на рисунке 45. Для формирования сети важны только номера элементов (линий

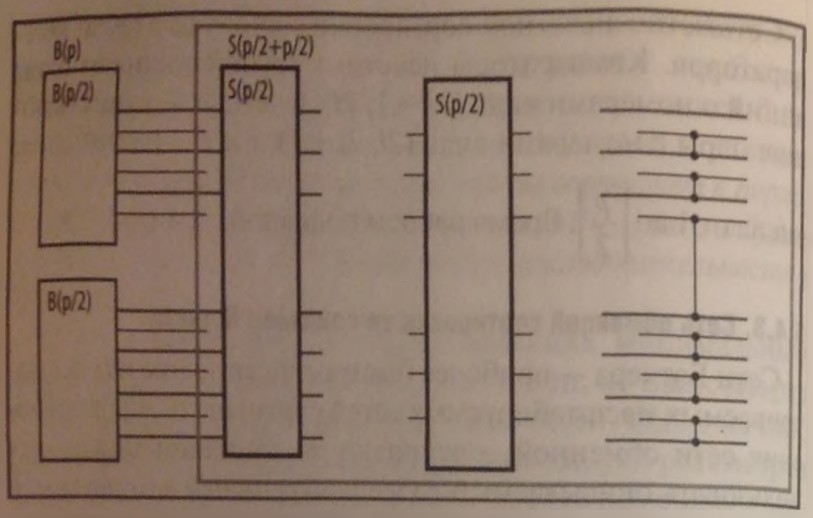


Рис. 45. Общая структура алгоритма построения сети обменной сортировки со слиянием Бэтчера

данных), а не сами элементы. Группу линий удобно описывать с помощью тройки чисел вида (*first*, *step*, *count*). В такую группу будут входить линии с номерами *first*, *first* + *step*, ..., *first* + (*count* – 1) * *step*. Множество входящих в сеть компараторов формируется с помощью рекурсивных процедур **B** и **S**.

B(*first*, *step*, *count*) — процедура рекурсивного построения сети сортировки группы линий (*first*, *step*, *count*).

S(*first1*, *first2*, *step*, *count1*, *count2*), — рекурсивная процедура слияния двух групп линий (*first1*, *step*, *count1*) и (*first2*, *step*, *count2*). Тексты этих процедур приведены в алгоритме А20.

Поскольку все линии исходного массива описываются тройкой (1, 1, *p*), вызов процедуры *B*(1, 1, *p*) обеспечит построение требуемой сети сортировки.

Алгоритм А20

Последовательный матричный алгоритм

```
if(n==2)
{
добавить компаратор (first,first+step);
return;
}

count1=[count/2]; // число элементов в первой половине
                  // массива
B( first,step,count1); // упорядочить первую половину
                       // массива
B( first+step*count1, step,count-count1);
            // упорядочить вторую половину массива
S( first, first+step*count1, step, count1, count-count1);
            // объединить упорядоченные части
}

S(a, b, d, n, m)
{
int n1,m1,i;

if(n*m<1) return;
if(n==1 && m==1)
{
comparator (a,b);
return;
}

n1=n-n/2; // количество нечетных строк в массиве a
m1=m-m/2; // количество четных строк в массиве b
S(a , b , 2*d, n1 , m1 ); // объединить нечетные линии
S(a+d, b+d, 2*d, n-n1, m-m1); // объединить четные
                              // линии

// добавить цепочку компараторов,
// начиная со второй линии

// компараторы между линиями первого массива
for(i=1;i<n-1;i+=2)
добавить компаратор (a+d*i,a+d*(i+1));

if(n%2==0)
```

```
 {
 добавить компаратор (a+d*(n-1),b); // компаратор между
                                    // массивами
 i=1;
 }
 else
 i=0;

 // компараторы между линиями второго массива
 for(;i<m-1;i+=2)
добавить компаратор (b+d*i,b+d*(i+1));
}
```

Например, сформированная в результате вызова процедуры **B(1, 1, 6)** сеть сортировки массива из шести элементов показана на рисунке 46, а список компараторов (в порядке их формирования процедурой) — на рисунке 47.

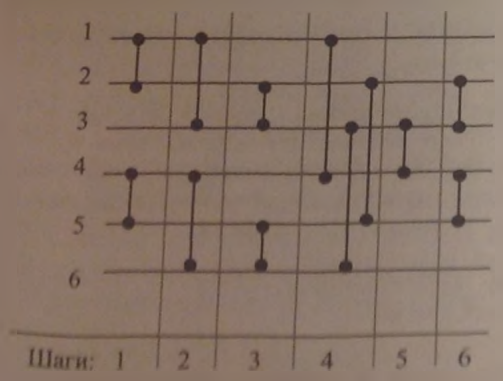


Рис. 46. Порядок выполнения слияния фрагментов массива методом обменной сортировки со слиянием Бэтчера

(1,2)(1,3)(2,3)

(4,5)(4,6)(5,6)

(1,4)(3,6)(3,4)

(2,5)

(2,3)(4,5)

Рис. 47. Результат работы процедуры B(1,1,p)

На рисунке 48 показан второй пример — фрагмент сети объединения двух упорядоченных массивов: {2, 4, 6, 8, 9, 13} и {1, 3, 5, 7, 10, 11, 12}. Вертикальные прямоугольные блоки обозначают сети слияния, обрабатывающие нечетные и четные группы строк массивов.

В результате объединения сетью слияния элементов, имеющих нечетные номера {2, 6, 9} и {1, 5, 10, 12}, получен упорядоченный массив {1, 2, 5, 6, 9, 10, 12}.

В результате объединения сетью слияния элементов, имеющих четные номера {4, 8, 13} и {3, 7, 11} получен упорядоченный массив {3, 4, 7, 8, 11, 13}.

В результате выполнения последней группы компараторов, получен полностью отсортированный массив: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}.

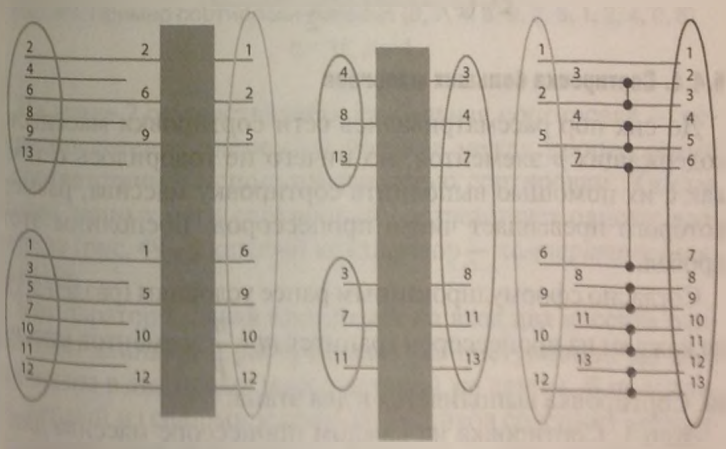


Рис. 48. Сеть нечетно-четного слияния Бэтчера

Анализируя рисунок 45, можно определить количество шагов сети сортировки Бэтчера при $p = 2^k$. Пусть $Q_s(p)$ — число шагов работы сети сортировки, $Q_s(p)$ — число шагов

работы сети слияния упорядоченных групп нечетных и четных линий, в каждой из которых $\frac{p}{2}$ линий. Тогда справедливы следующие функциональные уравнения:

$$Q_S(p)=Q_S\left(\frac{p}{2}\right)+1 \qquad Q_S(2)=1,$$

$$(p)=Q_B\left(\frac{p}{2}\right)+Q_S(p) \qquad Q_B(2)=1.$$

Решением этих уравнений являются функции

$$Q_S(p)=\log_2 p, \qquad Q_B(p)=\frac{\log_2 p(\log_2 p+1)}{2}.$$

При $p \neq 2^k$ время работы не превышает

$$Q_B(p)=\frac{\lceil\log_2 p\rceil\lceil\log_2 p+1\rceil}{2}.$$

5.4.4. Сортировка больших массивов

До сих пор рассматривались сети сортировки массивов, содержащих p элементов, но ничего не говорилось о том, как с их помощью выполнить сортировку массива, размер которого превышает число процессоров. Восполним этот пробел.

Согласно сформулированным ранее условиям (раздел 5.1), на каждом из процессоров хранится $m=\frac{n}{p}$ элементов массива. Сортировка выполняется в два этапа:

Этап 1. Сортировка на каждом процессоре массива длиной m.

Этап 2. Слияние отсортированных фрагментов в соответствии с расписанием, заданным используемой сетью сортировки p элементов.

На этапе 1 каждый процессор независимо от других выполняет процедуру последовательного упорядочивания

элементов расположенного на нем массива длиной m. Сортировка выполняется наилучшим из доступных последовательных алгоритмов. К настоящему моменту это описанный ранее алгоритм ***dhsort***, затрачивающий на сортировку m элементов время:

$$T_{seq}(m) = Km \log_2 m.$$

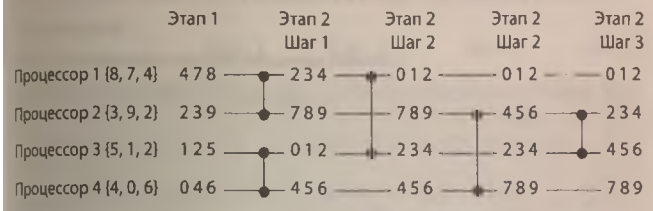


Рис. 49. Пример сортировки массива {8, 7, 4, 3, 9, 2, 5, 1, 2, 4, 0, 6} $n = 12$, $p = 4$

На этапе 2 осуществляется глобальная сортировка — перераспределение элементов массива между процессорами в соответствии с используемой сетью сортировки. Каждая линия данных сети сортировки соответствует одному *процессору* (рис. 49), а каждый компаратор — *компаратору слияния* [19].

Компаратор *слияния* принимает на вход два массива одинаковой длины m и распределяет все поступившие на вход элементы в два новых массива такой же длины. В результате, первый из сформированных массивов содержит элементы с меньшими значениями, в второй — с большими.

В случае использования вычислительной системы с распределенной памятью добавляется этап передачи данных между процессорами:

- каждый из процессоров пересылает хранящийся на нем фрагмент массива процессору, подключенному к друго-

му входу компаратора. В результате на каждом из процессоров оказываются оба фрагмента;

- процессор с меньшим номером (алгоритм A21) выделяет из двух фрагментов m наименьших элементов, формируя новый отсортированный фрагмент с длины m. Одновременно с этим, процессор с бо́льшим номером (алгоритм A22) выделяет m наибольших элементов, формируя новый отсортированный фрагмент с длины m.

Алгоритм A21

Слияние меньших элементов массивов *a* и *b*

```
// формирование массива c, содержащего меньшие элементы
// массивов a и b
  for(ia=0,ib=0,k=0;k<m;)
    {
    if(a[ia]<b[ib])  c[k++]=a[ia++];
    else  c[k++]=b[ib++];
    }
```

Алгоритм A22

Слияние больших элементов массивов *a* и *b*

```
// формирование массива c, содержащего большие элементы
// массивов a и b
  for(ia=m-1,ib=m-1,k=m-1;k>=0;)
    {
    if(a[ia]>b[ib])  c[k--]=a[ia--];
    else  c[k--]=b[ib--];
    }
```

На выполнение операций, соответствующих одному компаратору слияния, затрачивается время, требуемое на пересылку m элементов и на выбор из двух упорядоченных массивов m элементов:

$$T_{send\_join}(m) = m(\tau_s + K).$$

Используя принцип нулей и единиц, можно показать, что описанный алгоритм правильно сортирует произвольные

массивы при условии равенства длин объединяемых фрагментов. Это условие является не только достаточным, но и необходимым. Подтверждение его необходимости приведено на рисунке 50, содержащем пример неправильной сортировки массива {(1), (1), (0), (0, 0)}, распределенного по четырем процессорам неравными порциями. После выполнения алгоритма, основанного на правильной сети сортировки, массив остается неупорядоченным: {(0), (0), (1), (0, 1)}.

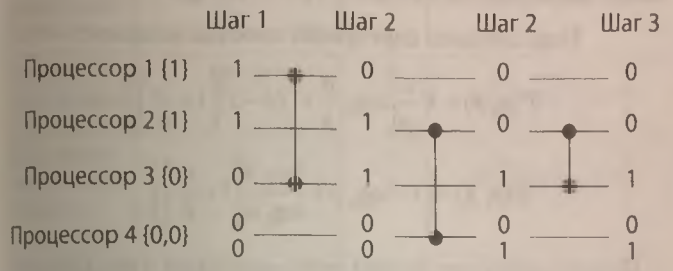


Рис. 50. Пример некорректной сортировки массива при разной длине фрагментов {1, 1, 0, 0, 0}, $n = 5$, $p = 4$, результат: {0, 0, 1, 0, 1}

Таким образом, для правильной сортировки массива следует при необходимости дополнять массив фиктивными элементами, чтобы длина непосредственно сортируемого массива была кратна числу процессоров.

5.4.5. Сравнение алгоритмов сортировки

Сравним характеристики параллельных алгоритмов, основанных на рассмотренных сетях сортировки. Время упорядочивания массива длины n на p процессорах складывается из двух составных частей. Во-первых, из времени сортировки каждым процессором фрагмента массива: $T_{seq}(m)$. Во-вторых, из произведения числа шагов сети сортировки $Q(p)$ на время срабатывания одного компаратора слияния: $Q(p)T_{send\_join}(m)$.

$$T(n,p) = T_{seq}\left(\frac{n}{p}\right) + Q(p)T_{send\_join}\left(\frac{n}{p}\right) =$$

$$= K\frac{n}{p}\left(\log_2\frac{n}{p} + Q(p)\left(1+\frac{\tau_s}{K}\right)\right).$$

Определим время выполнения и эффективности алгоритмов относительно наилучшего последовательного метода *dhsort*, время работы которого $T(n) = K n \log_2 n$.

Параллельная сортировка простой вставкой:

$$T(n,p) = K\frac{n}{p}\left(\log_2\frac{n}{p} + (2p-3)\left(1+\frac{\tau_s}{K}\right)\right),$$

$$E(n,p) = \left(1 - \log_n p + \frac{2p-3}{\log_2 n}\left[1+\frac{\tau_s}{K}\right]\right)^{-1}.$$

Параллельная сортировка четно-нечетной перестановкой:

$$T(n,p) = K\frac{n}{p}\left(\log_2\frac{n}{p} + p\left(1+\frac{\tau_s}{K}\right)\right),$$

$$E(n,p) = \left(1 - \log_n p + \frac{p}{\log_2 n}\left(1+\frac{\tau_s}{K}\right)\right)^{-1}.$$

Параллельная обменная сортировка со слиянием Бэтчера:

$$T(n,p) = K\frac{n}{p}\left(\log_2\frac{n}{p} + \frac{\lceil\log_2 p\rceil\lceil\log_2 p+1\rceil}{2}\left(1+\frac{\tau_s}{K}\right)\right),$$

$$E(n,p) = \left(1 - \log_n p + \frac{\lceil\log_2 p\rceil\lceil\log_2 p+1\rceil}{2\log_2 n}\left(1+\frac{\tau_s}{K}\right)\right)^{-1}.$$

Таблица 4

Зависимость количества операций от размера массива и числа процессоров

Алгоритм	Число шагов сети сортировки $Q(p)$	Объем памяти процессора	Оценка числа операций
Параллельная сортировка сдваиванием	—	$2n$	$O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$
Сеть простой вставки	$2p-3$	$3\frac{n}{p}$	$O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$
Сеть четно-нечетной перестановки	p	$3\frac{n}{p}$	$O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$
Сеть обменной сортировки со слиянием Бэтчера	не больше $\frac{\lceil\log_2 2p\rceil\lceil\log_2 p\rceil}{2}$	$3\frac{n}{p}$	$O\left(\frac{n}{p}\left(\log_2\frac{n}{p}+\frac{\lceil\log_2 p\rceil^2}{2}\right)\right)$

В таблице 4 приведены оценки числа операций, выполняемых рассмотренными методами.

Асимптотические оценки методов параллельной сортировки сдваиванием и алгоритмов, основанных на сетях простой вставки и четно-нечетной перестановки, одинаковы: $O\left(\frac{n}{p}\left(\log_2\frac{n}{p}+n\right)\right)$. Однако, между первым из них и двумя остальными есть принципиальная разница.

При сортировке первым способом максимальный размер массива ограничен размером оперативной памяти одного процессорного узла, а при сортировке с помощью сетей — размером совокупной памяти всех используемых процессо-

ров. В алгоритмах, основанных на сетях сортировки ни в какой момент времени, ни на каком процессоре не требуется хранить объем данных больший, чем $3\frac{n}{p}+O(1)$ элементов массива. Таким образом, единственное ограничение на размер массива — совокупный объем оперативной памяти вычислительной системы. Он должен не менее чем в три раза превышать объем сортируемых данных, что обеспечивает возможность, при использовании достаточного числа процессорных узлов, упорядочить произвольно большой массив.

Несмотря на то, что часть вычислительной работы в сетях простой вставки и четно-нечетной перестановки выполняется параллельно, ускорение и эффективность этих методов малы, особенно на вычислительных системах с распределенной памятью, где $\tau_s \gg K$. Основная причина — в большом объеме последовательно передаваемых между процессорами данных. Но и в системах с общей памятью часть работы фактически выполняется последовательно, на что указывает член $O(n)$.

Именно благодаря отсутствию в оценке времени выполнения члена вида $O(n)$, алгоритм обменной сортировки со слиянием Бэтчера значительно эффективнее остальных. В его рамках, при увеличении числа используемых процессоров, сокращается время работы *всех* этапов выполнения сортировки. Этому алгоритму присуще другое ограничение: степень внутреннего параллелизма существенно меняется от шага к шагу работы сети сортировки. Эффективность алгоритма обменной сортировки со слиянием Бэтчера меньше 100%, поскольку уже в расписании обменов между процессорами, заложены простои некоторых из них.

Например, из приведенной на рисунке 46 схемы следует, что на четвертом шаге одновременно выполняются компараторы слияния следующих пар процессоров: (1, 4), (2, 5), (3, 6). На этом шаге задействованы все процессоры. Одна-

ко, на остальных шагах это не так. Например, на пятом шаге простаивают все процессоры, кроме пары (3, 4). За 6 шагов могли бы выполниться 18 компараторов слияния, но выполняются только 12; совокупная эффективность при работе сети равна 66%.

Максимально возможная эффективность всего алгоритма, полученная в предположении отсутствия затрат на передачу данных (например, при сортировке на системах с общей памятью), составляет:

$$E^{max}(n, p) \approx \left(1 + \frac{\log_n p}{2} \log_2 \frac{p}{2}\right)^{-1}.$$

Тем не менее, именно такое жесткое расписание обменов обеспечивает высокое ускорение рассмотренного алгоритма.

5.5. Результаты численных экспериментов

В таблице 5 приведены результаты, полученные при сортировке ста миллионов 4-байтовых целых чисел на первом Российском суперкомпьютере МВС 1000М (МСЦ РАН), достигнувшем терафлопной производительности. В его составе было 768 процессоров Alpha21264A тактовой частотой 667 МГц, объединенных сетью Myrinet2000, обеспечивавшей обмен данными между двумя процессорными узлами со скоростью 110—150 Мбайт/сек.

Таблица 5

Сортировка 108 4-байтовых целых чисел на системе МВС 1000М

p	T, сек	B	S	E^{max}	S^{max}	Q(p)
1	83,51	100,00%	1,00	100%	1,0	0
2	46,40	90,00%	1,80	100%	2,0	1
3	35,93	77,48%	2,32	96%	2,8	3
4	29,68	70,35%	2,81	96%	3,9	3

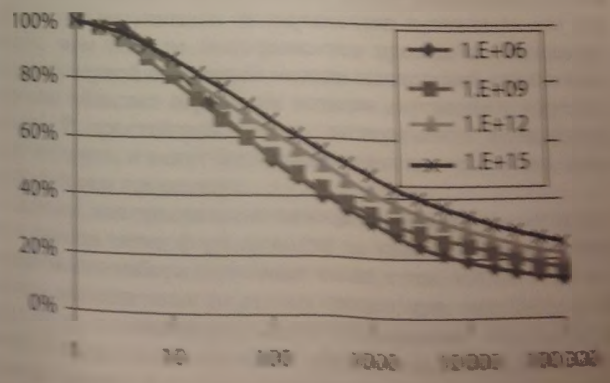
Окончание табл. 5

p	T, сек	E	S	E^{max}	S^{max}	Q(p)
5	24,45	68,33%	3,42	90%	4,5	5
6	22,16	62,80%	3,77	90%	5,5	6
7	21,82	54,67%	3,83	90%	6,2	6
8	19,95	52,32%	4,19	90%	7,2	6
16	12,36	42,22%	6,75	82%	13,1	10
27	9,32	33,20%	8,97	64%	20,0	15
32	7,85	33,24%	10,64	64%	23,3	15
48	6,45	26,97%	12,95	64%	31,9	21
64	4,92	26,53%	16,98	64%	40,9	21
128	3,19	20,47%	26,20	56%	71,5	28
192	2,52	17,29%	33,19	49%	98,2	36
256	1,99	16,41%	42,02	49%	124,6	36
384	1,63	12,38%	51,20	49%	157,0	45
512	1,29	12,64%	64,74	42%	217,4	45
640	1,21	10,76%	69,02	37%	264,7	52

Здесь T — время сортировки, E, S — эффективность и ускорение, полученные при численном расчете, E^{max}, S^{max} — максимально возможные эффективность и ускорение, обусловленные степенью внутреннего параллелизма используемого алгоритма. Массив, содержащий 100 миллионов целых 4-байтовых чисел, отсортирован на 640 процессорах за 1,21 секунды. Одному процессору этой системы

для сортировки того же массива потребовалось 83,51 сек. Максимальное ускорение, которое могло бы быть достигнуто на 640 процессорах в отсутствие потерь на передачу данных, составляет 264,7. Достигнутое ускорение составило 69, что является хорошим результатом, учитывая малое число элементов массива и большое число использованных процессоров. Высокую масштабируемость метода подтверждает факт отсутствия насыщения даже при 640 процессорах: время выполнения сортировки неуклонно уменьшается при увеличении числа используемых процессоров.

В заключение приведены графики, подтверждающие медленный спад максимально возможной эффективности при сортировке массивов размером от 10^6 до 10^{15} элементов на вычислительных системах, содержащих до ста тысяч процессоров (рис. 51а). Благодаря тому, что эффективность снижается медленно, максимально достижимое ускорение в широком диапазоне чисел используемых процессоров растет практически линейно (рис. 51б).



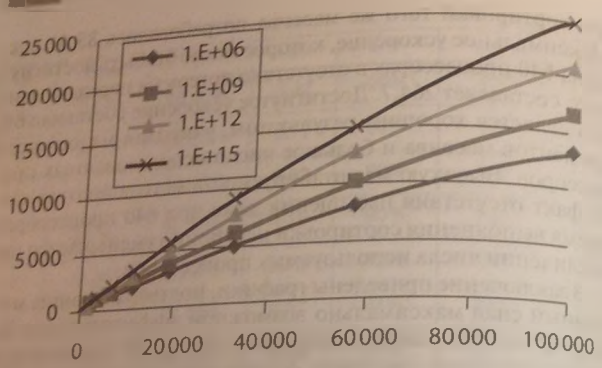


Рис. 51б. Зависимость ускорения от числа процессоров для разных размеров сортируемых массивов

Глава 6

Генерация псевдослучайных чисел

Последовательности случайных чисел (ПСЧ) находят широкое применение в вычислительной практике. Они 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. Как правило, методы, использующие случайные числа, предполагают 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. Привлекательной чертой описанной модели вычислений является то, что она обладает огромным запасом внутреннего параллелизма. В простейшем случае, эксперименты независимы друг от друга, и могут быть выполнены одновременно, каждый на своем процессоре.

Однако, для организации одновременной обработки требуется указать метод формирования на каждом из процессоров уникального набора случайных чисел, с тем, что бы эксперименты, выполненные на разных процессорах, не дублировали друг друга, а привносили новую информацию, уточняющую ответ. Рассмотрению методов, обеспечивающих формирование на каждом из процессоров взаимонезависимых последовательностей случайных или псевдослучайных чисел, и посвящена настоящая глава.

6.1. Требования к генераторам псевдослучайных чисел для МВС

Укажем два вида генераторов последовательностей чисел для решения задач стохастическими (недетерминированными, использующими случайные числа или их имитацию) методами:

– генераторы случайных чисел, основанные на некоторых физических принципах;

– генераторы псевдослучайных чисел, основанные на детерминированных алгоритмах.

Отметим кратко некоторые существенные особенности генераторов указанных видов.

Генераторы случайных чисел, основанные на физических принципах:

– обеспечивают формирование произвольно длинных последовательностей случайных чисел, что является их значительным преимуществом;

– обеспечивают формирование на различных процессорах различных последовательностей, что так же является их преимуществом;

– не обеспечивают вопроизводимость результатов. Является данный факт недостатком или преимуществом зависит от решаемой с помощью генератора задачи. С точки зрения криптографических приложений это скорее преимущество, но, с точки зрения удобства отладки вычислительных приложений, — недостатком. Невозможность многократного воспроизведения одной и той же последовательности случайных чисел, сформированных однажды с помощью физического генератора, существенно затрудняет отладку приложений, не только параллельных, но и последовательных;

– свойства формируемых последовательностей могут меняться в зависимости от таких параметров окружающей среды, как температура, влажность, давление и прочие. Например, класс физических генераторов, основанных на комбинации простых

осцилляторов, чувствителен к наличию внешнего высокочастотного электромагнитного поля [83]. Оно приводит к возникновению резонанса между осцилляторами, в результате чего формируемая последовательность перестает удовлетворять подавляющему большинству тестов, и становится фактически непригодной для проведения вычислительных экспериментов;

– не поддаются априорному тестированию. Этот недостаток является прямым следствием предыдущих двух. Предварительное тестирование генератора не гарантирует высокое качество формируемой в момент решения задачи последовательности. Невозможно заранее оттестировать именно тот фрагмент последовательности, который будет использован при решении задачи. Этот фрагмент уникален и не может быть сформирован дважды — при тестировании и при решении задачи. Свойства генератора, в силу предыдущего пункта, могут быть совершенно разными в момент тестирования и в момент решения задачи, что снижает ценность результатов предварительного тестирования;

– 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, что является их недостатком, но вполне преодолимым, поскольку известны методы формирования последовательностей, обладающих произвольно большим периодом;

– обеспечивают формирование на различных процессорах различных последовательностей, что является их преимуществом;

– ряд методов алгоритмической генерации обеспечивает возможность согласованного формирования на множестве процессоров фрагментов единой последовательности, что также является существенным преимуществом с точки зрения создания масштабируемых параллельных алгоритмов;

— могут быть предварительно протестированы, как на стандартных тестах, так и на ряде задач требуемого класса.

Таким образом, определяя способ формирования последовательностей чисел для параллельных вычислений, предварительно необходимо ответить на два вопроса:

— должен ли ответ, получаемый в результате последовательных вычислений, быть одним и тем же при различных запусках программы?

— должен ли ответ, найденный в ходе параллельных вычислений на нескольких процессорах в точности совпадать с ответом последовательной версии программы?

Если воспроизводимость результатов не требуется, а качество последовательностей имеющихся физических генераторов (их должно быть много — по числу используемых процессов) сомнений не вызывает, то вполне оправдано использование именно их.

В противном случае, предпочтение следует отдать программным генераторам. Отметим, что воспроизводимость результатов от запуска к запуску последовательной программы существенно облегчает локализацию возможных ошибок в последовательных программах, поскольку позволяет многократно выполнить одни и те же вычисления в разных режимах контроля вычислительного процесса (например, с отладочной печатью или под управлением отладчика). Более того, в существующих в настоящее время средствах отладки и верификации параллельных программ методу сравнения заслуженно отводится важная роль. Например, этот метод используется в системе DVM [24–26], предназначенной для создания эффективных переносимых параллельных вычислительных приложений на языках C-DVM и Fortran-DVM (*Distributed Virtual Machine* или *Distributed Virtual Memory*).

Дальнейшее изложение посвящено именно программным методам формирования последовательностей псевдослучайных чисел, как наиболее универсальным и обеспечи-

вающим удобство отладки, допускающим априорное тестирование, и слабо зависящим от аппаратных особенностей вычислительных систем.

Рассмотрим возможные методы формирования последовательностей псевдослуайных чисел на произвольном числе процессоров:

1) написание уникального генератора для каждого из процессоров;

2) использование на процессоре с номером *rank* элементов последовательности $u[(rank+ip) \bmod T]$, где p – число процессоров, а T – период исходной последовательности u (leapfrog метод).

3) использование на процессоре с номером *rank* элементов последовательности $u[num(rank)]... u[num(rank+1)-1]$. Согласно этому методу на каждом процессоре используется большой непрерывный фрагмент одной и той же последовательности u (skip-ahead метод).

Очевидно, что первый метод может быть использован только при небольшом числе процессоров и фактически не пригоден при разработке масштабируемых алгоритмов.

Второй метод особенно удобен при формировании последовательностей для векторных вычислительных систем. Далее будет показано, как именно можно быстро вычислять каждый p-ый член последовательности на каждом из процессоров. Однако этот метод имеет, как минимум два существенных недостатка. Рассмотрим в качестве примера параллельный алгоритм формирования множества псевдослучайных точек, распределенных в некоторой области трехмерного пространства.

Во-первых, leapfrog метод не обеспечивает идентичности результатов, полученных на разном числе процессов, поскольку для каждого числа процессов 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 составят числа с четными номерами, а второго – с нечетными, поэтому будут сформированы другие точки:

У первого процесса: (0,2,4),(6,8,10),(12,14,16).

У второго процесса: (1,3,5),(7,9,11),(13,15,17).

Поскольку сформированные множества точек не совпадают, мы не сможем непосредственно сравнить результаты, полученные при решении задачи одним и двумя процессами.

Второй существенный недостаток связан с тем, что высокое качество последовательности $u[i]$, не гарантирует при произвольном числе процессов p качества последовательностей $u[(rank+ip) \bmod T]$.

Характерный пример приведен на рисунке 52. Исходная последовательность $u_k=(5u_{k-1}+1) \bmod 64$ имеет вполне случайный вид, однако, её подпоследовательность v_k, включающая каждый четвертый элемент исходной последовательности ($v_k=u_{4k}$), является линейной функцией (в пределах периода, рис. 52а), что никак не отвечает требованию случайности.

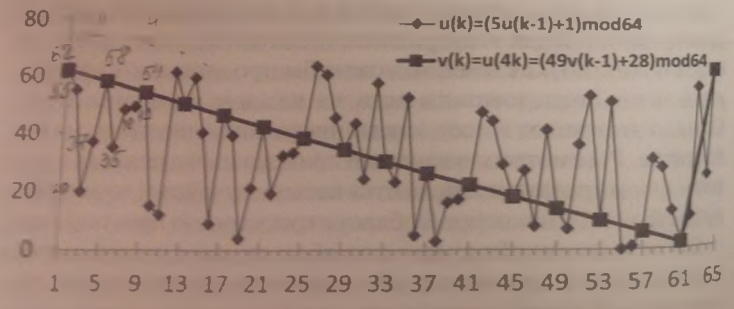


Рис. 52а. Метод leapfrog генерации последовательности псевдослучайных чисел

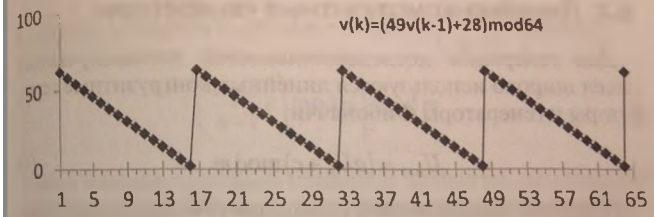


Рис. 52б. Метод leapfrog генерации последовательности псевдослучайных чисел

Таким образом, в общем случае, наиболее привлекательным выглядит именно третий метод – формирование на каждом из процессоров больших непересекающихся между собой фрагментов одной исходной последовательности (skip-ahead метод). Он позволяет, возвращаясь к примеру с точками в трёхмерном пространстве, в первом процессе использовать числа с номерами (0,...,8), а во втором — (9,...,17), формируя тем самым в точности тот же набор пространственных точек, что и при последовательной обработке. Подчеркнем, что, с точки зрения построения эффективных параллельных методов, необходимо обеспечить возможность формирования фрагментов последовательности, начинающихся в требуемых местах, без вычисления на каждом процессоре всех предшествующих фрагменту элементов исходной последовательности.

Подводя итог рассмотрению возможных вариантов формирования псевдослучайных чисел на множестве процессоров, сформулируем основные требования, предъявляемые к генераторам псевдослучайных чисел, предназначенных для использования на многопроцессоорых системах [20–23]:

1) возможность формирования последовательности достаточно большой длины;

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

5.2. Линейно-конгруэнтные генераторы

Для генерации последовательностей псевдослучайных чисел широко используются линейные конгруэнтные генераторы и генераторы Фибоначчи.

$$U_{n+1} = (aU_n + c) \bmod m. \tag{4}$$

Линейные конгруэнтные генераторы (4) позволяют с помощью соотношений (5)—(6) одновременно вычислять на векторном компьютере, оснащенном k процессорами, очередные k элементов последовательности, используя для вычисления каждого элемента порядка $O(1)$ операций [27]. Для этого достаточно один раз предварительно вычислить величины $a^k \bmod m$ и $a^k \bmod m$ и $\left(\frac{a^k-1}{a-1}c\right) \bmod m$.

$$U_n = \left(a^n U_0 + \frac{a^n-1}{a-1}c\right) \bmod m, \tag{5}$$

$$U_{k(n+1)} = \left(a^k U_{kn} + \frac{a^k-1}{a-1}c\right) \bmod m. \tag{6}$$

Однако, выражение (5) позволяет непосредственно вычислять элементы с произвольным номером n, используя порядка $O(\log n)$ операций (здесь и далее по тексту подразумевается использование логарифма по основанию 2). Чтобы подтвердить это, достаточно показать возможность понижения вдвое степени стоящих в скобках слагаемых. Используя тождество (7), можно преобразовать (5) к виду (8), где $n = k + t$, $k = \lfloor n/2 \rfloor$, а $\lfloor x \rfloor$ — наименьшее целое, не меньшее x. Рекурсивно повторяя процесс понижения степени, получим требуемый результат

$$a^{k+t} - 1 = a^t\left(a^k - 1\right) + a^t - 1, \tag{7}$$

$$U_n = \left[\left(a^n U_0 \right) \bmod m \cdot \left(a^n U_0 \right) \bmod m + a^n \bmod m \cdot \left(\frac{a^n - 1}{a - 1} c \right) \bmod m + \left(\frac{a^n - 1}{a - 1} c \right) \bmod m \right] \bmod m. \quad (8)$$

Еще меньше операций потребует алгоритм, основанный на использовании бинарного представления номера вычисляемого элемента n.

$$n = \sum_{i=0}^{\lfloor \log n \rfloor} \beta_i 2^i, \ \beta_i = \frac{n}{2^i} \bmod 2. \quad (9)$$

Рассмотрим функцию вычисления значения элемента последовательности, имеющего номер на 2^j больше, чем номер элемента, заданного в качестве аргумента (10).

$$F_j(U_j) = \beta_j \left[\left(a_j U_j + c_j \right) \bmod m \right] + (1 - \beta_j) U_j, \quad (10)$$

$$a_0 = a, \ a_{i+1} = a_i^2 \bmod m, \quad (11)$$

$$c_0 = c, \ c_{i+1} = \left[\left(a_i + 1 \right) c_i \right] \bmod m. \quad (12)$$

Пусть в качестве аргумента функции F_j задан элемент последовательности с номером j. Тогда при $\beta_i = 1$ значением функции является элемент последовательности с номером $j + 2^i$, а при $\beta_i = 0$ элемент с номером j. Коэффициенты a_i и c_i определяются рекуррентно, согласно (11), (12), и могут быть вычислены однократно в начале работы. Их количество равно $M = \lfloor \log N \rfloor$, где N — максимальный номер требуемого в ходе выполнения всех вычислений элемента последовательности.

Таким образом, зная элемент последовательности с нулевым номером, можно, используя (13), вычислить произвольный элемент последовательности за M обращений к функции F_j.

$$U_n = F_0\left(F_1\left(F_2\left(\ldots F_b(U_0)\ldots\right)\right)\right). \qquad (13)$$

Ряд свойств линейных конгруэнтных генераторов рассмотрен в [3], там же приведена библиография и содержатся примеры конкретных значений коэффициентов. К недостаткам генераторов этого типа 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. Вторая проблема, связанная с этими генераторами, описана в [28, 3]. Она заключается в том, что d-мерные точки, координаты которых сформированы с помощью (4), располагаются не более чем в $\sqrt[d]{d!m}$ гиперплоскостях размерности $(d-1)$. В ряде случаев этот факт значительно ограничивает применимость линейных конгруэнтных генераторов, например, в геометрических приложениях. В частности, при $a = 2^{16} + 3$, $c = 0$ и $m = 2^{31}$ (параметры, использованные в библиотеке IBM System 360/370) [27] в трехмерном случае ($d = 3$) таких плоскостей не более 16.

6.3. М-последовательности

При необходимости эффективной генерации ПСЧ с большими периодами используют генераторы Фибоначчи и одну из их форм M-последовательности [29], представляющие собой рекуррентные соотношения (14) порядка r.

$$U_n = a_{n-1}U_{n-1} + a_{n-2}U_{n-2} + \ldots + a_{n-r}U_{n-r} \bmod 2.$$
$$U_0 = 1, U_1, \ldots U_{r-1} = 0 \qquad (14)$$

Например, соотношение $Un = Un-20 + Un-33$ mod2, записанное для одноразрядных двоичных чисел, обеспечивает генерацию последовательности бит с периодом $2^{33}-1$ [30]. Обзор методов построения генератора Фибоначчи вида (15) приведен в [27].

$$U_n = U_{n-r} \Theta U_{n-s} \qquad (15)$$

где Θ — одна из операций: $+, -, *$ или сложение по модулю.

Для вычисления элемента M-последовательности с произвольным номером может быть использован метод, основанный на предварительном вычислении матриц смещения (*jump ahead*), рассмотренный, например, в [22, 21, 23, 31]. Этот метод применим к генераторам достаточно общего вида, однако его использование сопряжено с необходимостью хранения матриц размера $r \times r$ и выполнения $O(r^3 \log_2 v)$ операций для вычисления элемента, номер которого на v больше, чем номер уже известного элемента [22]. Ввиду большой трудоемкости формирования матрицы смещения, метод применим для вычисления элементов, расстояние между которыми фиксировано. Улучшенный вариант этого метода [22] требует не менее, чем $O(r^2 \log_2 v)$ операций для вычисления элементов при произвольном v. Рассматриваемый далее метод позволяет за счет сужения класса рассматриваемых генераторов, снизить оценку числа операций до величины $O(r \log r \log v)$ и оценку объема дополнительно хранимых данных до $O(r \log r)$.

Согласно [32], последовательности, формируемые с помощью рекуррентного соотношения (14), эквивалентны последовательностям, формируемым с помощью полиномов (16):

$$t_i = x^i \bmod G(x), \qquad (16)$$

где $G(x)$ — примитивный полином [33, 34] степени r,

$t_i = \sum_{j=0}^{r-1} c_{i+j} x^j$ — ненулевые полиномы степени ниже r.

Поясним это утверждение.

Докажем, что, если

$$G(x) = \sum_{i=0}^{r} g_i x^i, \qquad g_r = 1, \qquad x^k \bmod G(x) = \sum_{i=0}^{r-1} f_i^k x^i$$

то, для $\forall k \geq r$ выполняется

$$f_0^k = \sum_{i=0}^{r-1} g_i f_0^{k-r+i} \qquad (17)$$

Покажем, что $x^k \bmod G(x) = \sum_{i=0}^{r-1} \left(g_i \left(x^{k-r+i} \bmod G(x) \right) \right)$ (18)

Так как $g_r = 1$, то

$$g_r x^k \bmod G(x) + \sum_{i=0}^{r-1} \left(g_i \left(x^{k-r+i} \bmod G(x) \right) \right) = \sum_{i=0}^{r} \left(g_i \left(x^{k-r+i} \bmod G(x) \right) \right)$$

Таким образом, тождество (18) эквивалентно выражению

$$\sum_{i=0}^{r} \left(g_i \left(x^{k-r+i} \bmod G(x) \right) \right) = 0,$$

правильность которого непосредственно следует из следующего преобразования:

$$\sum_{i=0}^{r} \left((g_i x^{k-r+i}) \bmod G(x) \right) = \left(x^{k-r} \sum_{i=0}^{r} (g_i x^i) \right) \bmod G(x) = \left(x^{k-r} G(x) \right) \bmod G(x) = 0 \blacksquare$$

Выделив в (18) только свободные члены,

$$x^k \bmod G(x) \bmod x = \sum_{i=0}^{r-1} \left(g_i \left(x^{k-r+i} \bmod G(x) \right) \bmod x \right),$$

и полагая $a_i = g_i$, получим эквивалентность последовательности U_a (14) и последовательности свободных члены полиномов f_0^k (17).

В качестве примера рассмотрим формирование элементов последовательности порождаемой полиномом

$$G(x) = x^4 + x^1 + x^0,\ r = 4$$

и рекуррентной последовательностью

$$f_0^0 = 1, f_0^1 = 0, f_0^2 = 0, f_0^3 = 0$$

$$f_0^k = f_0^{k-4+3} + f_0^{k-4+0}$$

В таблице 6 непосредственно видна идентичность последовательностей (14) и $x^k \bmod G(x) \bmod x$.

Таблица 6

Эквивалентность М-последовательностей и полиномов

k	$f_0^k = f_0^{k-1} + f_0^{k-4}$ $x^k \bmod G(x) \bmod x$	$x^k \bmod G(x)$
0	1	1
1	0	x
2	0	x^2
3	0	x^3
4	1	$x^3 + 1$
5	1	$x^3 + x + 1$
6	1	$x^3 + x^2 + x + 1$
7	1	$x^2 + x + 1$
8	0	$x^3 + x^2 + x$
9	1	$x^2 + 1$
10	0	$x^3 + x$
11	1	$x^3 + x^2 + 1$
12	1	$x + 1$
13	0	$x^2 + x$
14	0	$x^3 + x^2$
15	1	1

С вычислительной точки зрения целесообразно использовать процедуру генерации, основанную именно на соотношении (16). Для определения z-разрядного псевдослучайного числа A используется соотношение $A = \sum_{i=0}^{z} 2^i (t_{s+i} \bmod x)$.

Пусть для хранения c_i используется бесконечная последовательность бит (на практике — кольцевой буфер длины r), первые r элементов которой заполнены коэффициентами полинома t_0. Из (16) непосредственно следует:

$$t_{i+1} = xt_i \bmod G(x) = \begin{cases} xt_i, & c_{i+r-1} = 0, \\ xt_i - G, & c_{i+r-1} = 1. \end{cases}$$

Пусть k – количество одночленов полинома $G(x)$. Тогда для определения коэффициентов каждого очередного полинома t_i, в худшем случае, следует выполнить инвертирование k разрядов, соответствующих ненулевым коэффициентам полинома $G(x)$ и установить $c_{i+r}=1$. В лучшем случае, достаточно установить $c_{i+r}=0$. Число операций, необходимых для поочередного получения значений одноразрядных элементов последовательности псевдослучайных чисел, фиксировано и невелико. Для генерации z-разрядного числа следует выполнить указанную процедуру z раз.

Рассмотрим действия, обеспечивающие формирование элемента последовательности с номером n. Положим $t_0=1$, тогда:

$$t_n = x^n \bmod G(x). \tag{19}$$

В соответствии с алгоритмом бинарного умножения [3], рассмотрим двоичное представление номера требуемого элемента:

$$q = \sum_{j=0}^{r-1} \beta_j 2^j, \quad \beta_j \in \{0,1\}, \text{ тогда} \tag{20}$$

$$x^q = \prod_{j=0}^{r-1} p_j^{\beta_j} \bmod G(x),$$

$$p_j = x^{2^j} \bmod G(x), \quad j \in [0, r-1]. \tag{21}$$

Поскольку число операций для определения каждого из произведений в (20) пропорционально числу одночленов в p_j, наиболее эффективны алгоритмы генерации, использующие полиномы G вида (22):

$$G(x) = \sum_{i=0}^{k} a_i x^{2^i - 1}, \quad \text{где} \quad k = \log(r+1). \tag{22}$$

При выполнении условия (22) полиномы вида $x^{2^j} \bmod G(x)$, необходимые для быстрого вычисления произвольного элемента U_n с помощью алгоритма бинарного умножения [3], содержат не более $\log_2(r+1)$ ненулевых членов. В качестве примера можно привести некоторые значения P_j для двух примитивных полиномов, первый из которых соответствует (22), а второй — нет (таблица 7).

Таким образом, с учетом (19), число действий, выполняемых, при вычислении бита с номером n, есть $O(r\log r\log n)$. Одновременно с определением бита с номером n правильно устанавливаются r разрядов — коэффициентов полинома $t_n = x^n \bmod G(x)$, необходимых для дальнейшего последовательного определения коэффициентов полиномов с номерами $n+1$, $n+2$... с помощью (14). Для формирования z-разрядного числа с номером Q следует вычислить коэффициенты полинома t_n, где $n=zQ$. Нулевой двоичный разряд искомого числа будет равен $t_n \bmod x$. Далее следует $z-1$ раз выполнить действия (14), устанавливая каждый раз i-й разряд искомого числа равным $t_{n+i} \bmod x$. Таким образом, при $z \le r$, формирование требуемого двоичного z-разрядного числа с произвольным номером Q можно выполнить за $O(r\log r\log n) + O(z) = O(r\log r\log n)$ операций. Объем данных, необходимых для хранения полиномов p_j оценивается как $O(r\log r)$, время для их первоначального однократного формирования оценивается как $O(r\log^2 r)$.

Таблица 7

Степени x по модулям разных полиномов

	$G(x)=x^{31}+x^3+1$	$G(x)=x^{31}+x^{28}+1$
i	x^{2^i}	x^{2^i}
0	x	x
1	x^2	x^2
2	x^4	x^4
3	x^8	x^8
4	x^{16}	x^{16}
5	x^3+x	$x^{29}+x$

Таблица 7 (продолжение)

6	x^6+x^5	$x^{26}+x^{25}+x^{24}+x^{23}+x^{22}+x^{21}+x^{20}+x^{19}+x^{18}+x^{17}+x^{16}+x^{15}+x^{14}+x^{13}+x^{12}+x^{11}+x^{10}+x^9+x^8+x^7+x^6+1$
7	$x^{16}+x^4$	$x^{26}+x^{25}+x^{24}+x^{23}+x^{22}+x^{21}+x^{20}+x^{19}+x^{18}+x^{17}+x^{16}+x^{15}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^6+x+1$
8	x^5+x^4+x	$x^{26}+x^{25}+x^{24}+x^{23}+x^{22}+x^{21}+x^{19}+x^{18}+x^{17}+x^{16}+x^{15}+x^{14}+x^{13}+x^8+x^7+x^6+x^5+x^4+x^3$
...	...	...
27	x^8+x^2+x	$x^{26}+x^{25}+x^{24}+x^{23}+x^{22}+x^{21}+x^{20}+x^{18}+x^{16}+x^{15}+x^{13}+x^{12}+x^9+x^8+x^2$
28	$x^{16}+x^4+x^2$	$x^{26}+x^{25}+x^{24}+x^{23}+x^{22}+x^{20}+x^{18}+x^{15}+x^{11}+x^8+x^4+x$
29	x^8+x	$x^{29}+x^{26}+x^{22}+x^{19}+x^{15}+x^{12}+x^8+x^5$
30	$x^{16}+x^2$	$x^{30}+x^{27}+x^{16}+x^{15}$
31	x	x

6.4. Проверка примитивности полиномов

Не для всех r существуют примитивные триномы вида (19). Кроме них, можно использовать примитивные полиномы, содержащие пять и более членов. На практике их можно быстро находить с помощью перебора на персональном компьютере. Для проверки примитивности полинома $G(x)$ [27, 32] достаточно убедиться, что выполняются условия (23)–(24):

$$x^{2^r} = x \bmod G(x), \tag{23}$$

$$x^{\frac{(2^r-1)}{h}} \neq 1 \bmod G(x), \tag{24}$$

где h_i — простые делители числа $2^r - 1 = \prod_i h_i$.

Например, полином (25) обеспечивает быстрое формирование последовательности длиной 10^{76} 32-битных целых чисел [35].

$$G(x) = x^{255} + x^{31} + x^7 + x^3 + 1. \tag{25}$$

Поскольку 255 не является Мерсеновской экспонентой, для проверки примитивности полинома (25) использовалась факторизация (разложение на простые множители) числа $2^{255} - 1$, приведенная в [36]:

$2^{521}-1 = 7 \cdot 31 \cdot 103 \cdot 151 \cdot 2143 \cdot 11119 \cdot 106591 \cdot 131071 \cdot 949111 \cdot$
$\cdot 85209728063337558431 \cdot 5702451577639775545838643151.$

Дополнительно укажем примитивные полиномы вида (19) обеспечивающие формирование последовательностей с длинами периодов 10^{155} и 10^{301} соответственно:

$$G(x) = x^{521} + x^{32} + 1,\ G(x) = x^{1023} + x^{7} + 1. \qquad (26)$$

6.5. Тестирование генераторов

Рассмотрим результаты проверки рассмотренных последовательностей с помощью разработанного George Marsaglia пакета для измерения качества последовательностей ПСЧ — тестов Diehard [37]. Они представляют собой набор из нескольких групп тестов, в том числе: расстояние между днями рождений, ранги двоичных матриц, словарные (обезьяньи) тесты, парковочный тест, монотонные последовательности, игра в кости. В общей сложности выполняется 319 серий тестов, результаты которых анализируются на соответствие аналитически определенным для каждого теста характеристикам. Например, парковочный тест сводится к анализу результатов проведения серии экспериментов, в каждом из которых выполняется 12 000 попыток случайного размещения в квадрате 100×100 единичных окружностей. Если очередная окружность пересекает уже существующую, она не размещается. Числа успешно «припаркованных» в результате выполнения экспериментов окружностей должны подчиняться нормальному распределению. В качестве меры соответствия ожидаемому распределению большинство тестов возвращает значение $P \in [0, 1]$ (в оригинале *p-value*). Последовательность расценивается как плохая, если в результате тестирования получено шесть или более P, равных 0 или 1.

Таблица 8 содержит результаты выполнения тестов Diehard для файлов, содержащих последовательности псевдослучайных чисел, сформированных рядом генераторов. Каж-

Таблица 8

Результаты тестов Diehard для разных генераторов ПСЧ

	$P = 0$	$0 < P < 0{,}00001$	$P < 0{,}001$	$P < 0{,}01$	$0.01 \le P \le 0{,}99$	$P > 0{,}99$	$P > 0{,}999$	$P > 0{,}99999$	$P = 1$
	a	b	c	d		D	C	B	A
LRND32 0	0	0	1	3	309	5	1	0	0
123456	0	0	1	4	312	2	0	0	0
1234567	0	0	0	6	309	4	0	0	0
12345678	0	0	1	5	312	1	0	0	0
123456789	0	0	0	2	311	6	0	0	0
SWBMWC	0	0	0	4	312	3	0	0	0
MWC	0	1	0	5	307	6	0	0	0
MIXRNDX	12	8	8	5	263	3	9	11	0
MIXRNDXY	12	10	5	5	262	4	7	14	0
BRND	74	5	2	13	187	2	11	18	7
MARD	146	117	0	0	32	2	2	2	18

дый файл, в соответствии с требованиями тестов Diehard, содержит порядка 11 Мбайт. Первая колонка таблицы соответствуют количеству тестов, P-значение которых попадает в интервал, указанный в первой строке. Например, в колонке, помеченной символом «a», указано, в результате выполнения скольких тестов получено значение $P = 0$. Тест считается непройденным, если сумма значений в колонках «a» и «A» больше или равна шести.

В строках таблицы 8 приведены результаты тестирования следующих генераторов:

LRND32 — M-последовательность на основе полинома (25). Числа в первой колонке соответствуют номеру первого элемента тестируемого фрагмента последовательности, период 10^{76} [35];

SWBMWC — комбинированный генератор на основе методов умножения с переносом MWC и Фибоначчи с запаздыванием SWBG [3], период $4 \cdot 10^{364}$;

MWC — генератор на основе метода умножения с переносом MWC [3], период $4 \cdot 10^{18}$;

MIXRNDX — рандомизация перемешиванием, алгоритм B [3, стр. 54];

MIXRNDXY — рандомизация перемешиванием, алгоритм M [3, стр. 54];

BRND — линейная конгруэнтная последовательность $x_{n+1} = 31415926534\,x_n + 2718281829 \bmod 2^{35}$, $x_0 = 0$, [3, стр. 69];

RAND — генератор библиотеки (run time library) Visual C.

Полученные результаты показывают, что последние четыре генератора, основанные на линейных конгруэнтных последовательностях, не удовлетворяют условиям тестов DIEHARD. Для генераторов SWBMWC, MWC и для фрагментов M-последовательности на основе полинома (22) все тесты выполнены успешно, что косвенно подтверждает возможность использования рассмотренных генераторов при проведении вычислительных экспериментов.

Глава 7

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. Далее рассматриваются критерии декомпозиции — требования, предъявляемые к виду доменов, и основные методы расчета декомпозиции.

7.1. Пример двумерной сетки

Характерный пример двумерной расчетной сетки приведен на рисунке 53. На нем изображена часть неструктурированной треугольной сетки, описывающей пространство вокруг крыла и закрылка самолета. Узлы сетки сгущаются к мелким деталям моделируемой конструкции и к тем зонам, в которых газодинамические параметры претерпевают наиболее существенные изменения. Несмотря на визуальное различие в размерах разноцветных зон, каждая из них содержит примерно одинаковое число узлов и, при выпол-

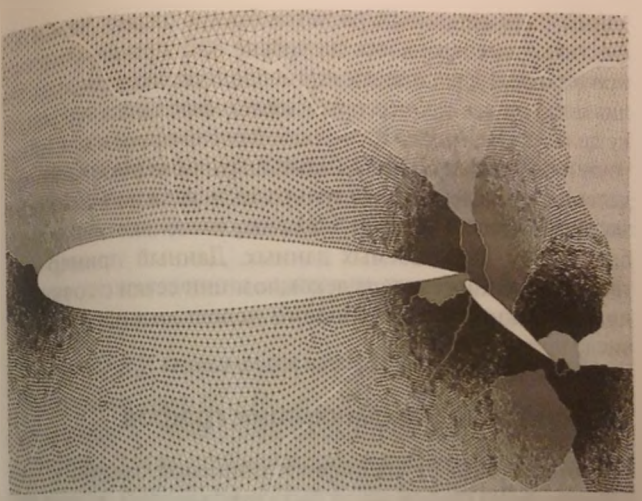


Рис. 53. Фрагмент нерегулярной сетки

нении численного моделирования, обрабатывается отдельным процессором. В ходе выполнения вычислений процессоры взаимодействуют между собой. Объемы передаваемых между процессорами данных определяются используемым численным методом, но для многих методов правомерно считать, что они пропорциональны длине границ доменов — числу ребер соединяющих вершины из разных доменов.

Ранее подробно рассматривались основные характеристики метода геометрического параллелизма — ускорение и эффективность. Было показано, что они существенно зависят от двух основных факторов: от равномерности распределения вычислительной нагрузки между процессорами и от объемов передаваемых данных. В первом приближении эффективность подобных методов описывается законом Амдаля. С ростом дисбаланса распределения вычислений между процессорами и с ростом объемов передаваемых

ных, эффективность расчетов быстро падает. Чтобы эффективность сохранялась на уровне 80% при двух тысячах процессоров, доля операций, не поддающихся распараллеливанию на все процессоры (последовательных операций) не должна превышать 10^{-4} — одной сотой процента. С такой точностью должна быть выдержана равномерность распределения вычислений и настолько мало должно быть время, затрачиваемое на обмены, а значит, настолько мал должен быть объем передаваемых данных. Данный пример подтверждает важность этапа декомпозиции сетки с точки зрения возможности эффективного использования большого числа процессоров.

7.2. Критерии декомпозиции графов

В зависимости от специфики решаемой задачи, к виду декомпозиции графов могут быть предъявлены разные требования (критерии). Хорошо известен классический критерий, поддерживаемый доступными программными пакетами. В его рамках предполагается выполнение следующих свойств:

- равномерное распределение по доменам суммарного веса узлов/ребер;
- минимизация максимального веса исходящих из домена ребер.

Выполнение каждого из них по отдельности не представляет сложности, но их одновременное удовлетворение является сложной задачей даже для сеток простой регулярной структуры. Формальное описание этого критерия дано в следующем параграфе (задача **(27)**—**(28)**). Критерий не накладывает ограничений ни на структуру формируемых доменов, ни на структуру связей между ними. Достаточно часто именно эти требования (равномерность распределения вершин по доменам при сокращении длины границ доменов) являются предпочтительными. Однако, для ряда задач

актуальны другие критерии, предъявляющие специфические требования к виду разбиения, соблюдение которых может способствовать упрощению параллельных алгоритмов и повышению их эффективности:

- выделение обособленных доменов;
- минимизация максимальной степени доменов;
- обеспечение связности доменов;
- обеспечение связности множества внутренних узлов доменов.

Далее перечисленные критерии рассматриваются подробнее.

7.2.1. Критерий 1: классический критерий декомпозиции графа

Пусть задан граф $G^0 = (V, E)$, $V\{v_i\}$, $|V| = n$. Пусть вершины v_i и ребра e_{ij} этого графа имеют веса $w(v_i)$ и $w(v_i, v_j)$ соответственно. Тогда, требуется найти такое разбиение $R(V) = (V_1, ..., V_p)$ вершин на заданное число доменов p, при котором J — взвешенная сумма весов вершин и разрезанных ребер — принимает минимальное значение (28):

$$V = \bigcup_{k=1}^{p} V_k,\ V_i \cap V_j = \varnothing,\ i \neq j. \tag{27}$$

$$\min_{R(V)} \left\{ J = \max_{k=1,\ldots,p} \sum_{v_i \in V_k} \left(w(v_i) + \alpha \sum_{v_j \notin V_k} w(v_i, v_j) \right) \right\}. \tag{28}$$

Множитель α обеспечивает согласование единиц измерения весов вершин и ребер, что позволяет одновременно учитывать оба требования. Во-первых, выравнивается вычислительная нагрузка, приходящаяся на каждый домен (она ассоциируется с весами вершин). Во-вторых, снижаются коммуникационные затраты, ассоциированные с числом разрезанных ребер. Задача декомпозиции сводится к минимизации J.

7.2.2. Критерий 2: выделение обособленных доменов

Рассмотрим декомпозицию сетки на домены двух видов. Домены первого вида (с нулевого по пятый на рисунке 54) отличаются тем, что среди них нет ни одной пары доменов, соединенных ребром. Все выходящие из доменов первого вида ребра замыкаются на узлы, принадлежащие доменам второго типа (домен 6 на рисунке 54).

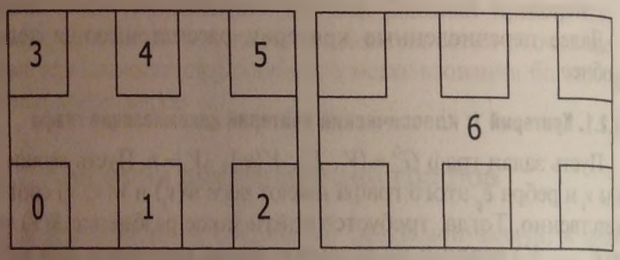


Рис. 54. Выделение обособленных доменов

Обработка единственным процессором домена 6 приводит к потерям эффективности. В системах с общей памятью эти потери компенсируются снижением числа конфликтов кэш-памяти. Обработка обособленных доменов разными процессорами позволяет уменьшить взаимное влияние, вызванное обращением разных процессоров к данным, отображаемым в одни и те же строки в кэш-памяти.

В системах с распределенной памятью описываемый подход позволяет использовать нетрадиционную стратегию организации работ, предполагающую конечность скорости распространения сигналов в природе [38]. В рамках этого подхода правомерно рассматривать обособленные домены (0—5) как невзаимодействующие на некотором небольшом интервале времени. В результате, задача распадается на множество несвязанных, каждая из которых может быть решена автономно на своем процессоре. Найденные таким образом

решения для доменов первого вида позволяют сформировать граничные условия для доменов второго вида, обработка которых выполняется на втором этапе.

В общем случае, r-дольная (двудольная, в рассмотренном примере) декомпозиция графа позволит выполнить расщепление задачи на r последовательно обрабатываемых групп независимых подзадач. Известно, что в двумерном случае при планарной сетке достаточно 4-х групп, что непосредственно следует из теоремы о 4-х красках. При таком подходе 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. Наибольший интерес эта технология представляет 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.

7.2.3. Критерий 3: минимизация максимальной степени домена

В дальнейшем нам потребуется понятие макрографа декомпозиции.

Макрограф

Макрограф $\hat{G}$ состоит из p вершин, каждая из которых соответствует одному домену V_i разбиения исходного графа:

$$\hat{G} = \left(\hat{V},\ \hat{E}\right),\ \hat{V} = \{\hat{v}_i\},\ \hat{E} = \{\hat{e}_{ij}\},\ \left|\hat{V}\right| = p. \qquad (29)$$

Вес $\hat{w}_i$ вершины $\hat{v}_i$ макрографа равен суммарному весу вершин, входящих в домен i:

$$\hat{w}_i = \sum_k w_k,\ v_k \in V_i. \qquad (30)$$

Вес $\hat{w}_{ij}$ ребра $\hat{e}_{ij}$, соединяющего вершины i, j макрографа, равен суммарному весу ребер, соединяющих вершины доменов i и j:

$$\widetilde{w}_{ij} = \sum_{k,l} w_{kl},\ v_k \in V_i,\ v_l \in V_j. \qquad (31)$$

Критерий минимизации максимальной степени домена (числа граничащих с ним доменов — максимальной степени вершины макрографа) обеспечивает снижение числа актов обмена данными между процессорами на каждом шаге по времени:

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

где K — максимально допустимая степень вершины макрографа, определяемая особенностями решаемой задачи.

В общем случае, это позволяет снизить коммуникационные расходы за счет уменьшения потерь, вызванных латентностью каналов передачи данных. Возможен и более существенный выигрыш. При использовании адаптивных сеток отсутствие треугольников сетки, вершины которых принадлежат трем разным доменам, позволяет упростить логику параллельного вычислительного алгоритма. Например, время расчета на 63 процессорах при использовании декомпозиции, показанной на рисунке 56, составило 4,16 секунды против 10,56 секунд при расчете, использующем классическую декомпозицию (рис. 55) [39]. Подчеркнем, что использовалась одна и та же расчетная программа, но узлы вычислительной сетки были распределены по доменам разными способами. Выигрыш более чем в два раза получен исключительно за счет удачной декомпозиции сетки.

7.2.4. Критерий 4: обеспечение связности графов каждого из доменов

Одним из существенных недостатков, присущих иерархическим методам декомпозиции графов, является формирование доменов, некоторые из которых состоят из разрозненных фрагментов сетки.

На рис. 57 показаны две части одного такого домена. Графы всех остальные доменов связные, а граф этого домена — фрагментирован. Формирование подобных разбиений характер-

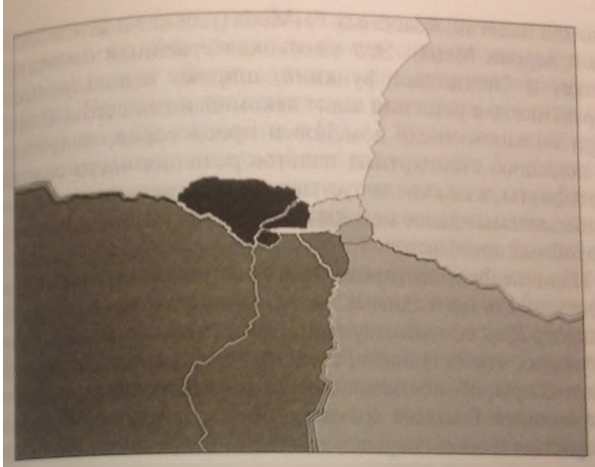


Рис. 55. Декомпозиция на 11 доменов в соответствии с классическим критерием декомпозиции [39]

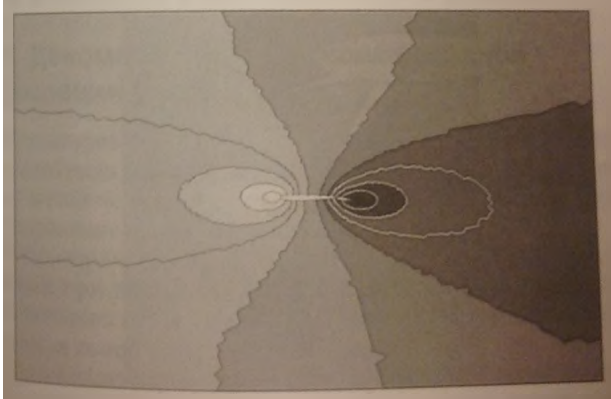


Рис 56. Минимизация степени вершины макрографа [39]

но для пакетов, подобных ParMetis (или его последовательной версии Metis). Это высококачественный бесплатный пакет и библиотека функций, широко используемые на практике для решения задач декомпозиции сеток. Однако, при большом числе доменов и процессоров, полученные с помощью стандартных пакетов решения часто содержат артефакты, в том числе пустые домены и несвязные домены. Описываемые далее методы направлены на снижение числа подобных артефактов.

Наличие фрагментированных доменов может снижать эффективность проводимых на их основе расчетов. Вершины макрографа, соответствующие несвязным доменам имеют большую, чем остальные, степень. Это приводит к тому, что процессоры, обрабатывающие фрагментированные домены, затрачивают большее время на обмен данными, поскольку возрастает и объем передаваемых данных, и число актов приема-передачи данных.

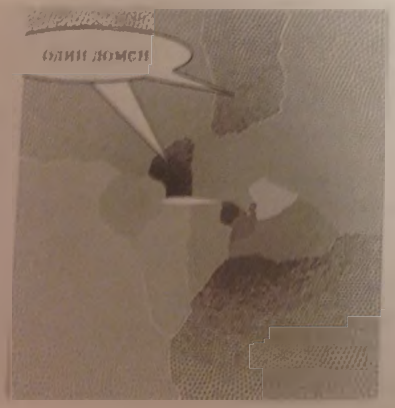


Рис. 57 Фрагментированный домен

Фрагментированность домена может приводить к существенному снижению эффективности компрессии сеточных функций (уменьшения объема данных, описывающих исходную функцию). Например, при сохранении результатов для визуализации больших объемов данных [40, 43,78] можно выполнять компрессию с контролируемой потерей точности. Коэффициент компрессии существенно зависит от того, насколько близки между собой значения функции в узлах фрагмента сетки. Если все узлы принадлежат одному компактному фрагменту сетки, то значения функции в этих узлах будут близкими с большей вероятностью, чем если они принадлежат двум фрагментам, расположенным в разных частях моделируемой области. Даже если локально функция изменяется плавно, с увеличением расстояния между узлами разница может оказаться существенной. При перемешивании в одном домене узлов, принадлежащих двум удаленным фрагментам сетки, алгоритму компрессии будут переданы узлы, содержащие существенно разные значения функции, что приведет к падению коэффициента компрессии. Подробнее этот вопрос обсуждается в главе 9, посвященной визуализации сеточных функций.

7.3. Декомпозиция на основе исходной нумерации узлов

Рассмотрев основные требования, предъявляемые к различного вида декомпозициям сеток, обратимся теперь к обзору методов, позволяющих получать декомпозиции, удовлетворяющие соответствующим критериям.

Рассмотрим тривиальный метод декомпозиции, используемый при первоначальном распределении по процессорам больших сеток с целью дальнейшего более аккуратного решения самой задачи декомпозиции.

Изначально нерегулярная сетка представляет собой множество последовательно пронумерованных узлов, между которыми определены некоторые связи, например, — ребра.

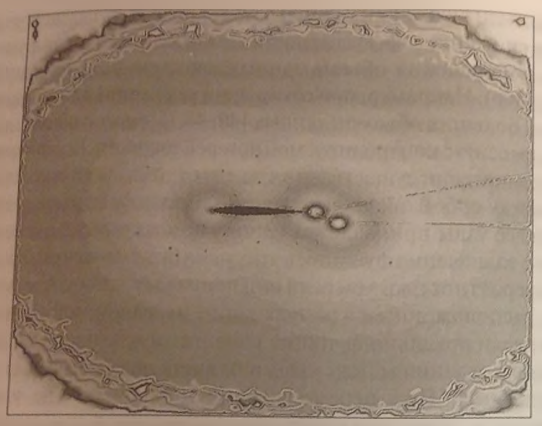


Рис. 58. Разбиение на 8 частей, основанное на естественной нумерации вершин

Простейший способ 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 $k\frac{n}{p}$ до $(k+1)\frac{n}{p}-1$. Такой подход гарантирует равномерность распределения узлов, но приводит к тому, что границы между доменами проходят по большому числу ребер (рис. 58).

К примеру, отдельные точки, отмеченные красным на рисунке 58, имеют соседние номера и будут назначены одному процессору. В результате, при расчете каждого шага по времени потребуется передача информации о всех узлах домена между процессором обрабатывающим этот домен и практически всеми остальными процессорами. В этом случае оценка времени выполнения алгоритма вместо (2) преобразуется к:

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

что практически совпадает с оценкой неудачного варианта конвейерного параллелизма и, как было показано, по эффективности значительно хуже решения, предполагающего компактную декомпозицию с малым объемом передаваемых данных. Таким образом, этот метод, в отличие от рассматриваемых далее, практически не пригоден для построения рациональных декомпозиций, но полезен для предварительного распределения узлов.

7.4. Рекурсивная бисекция

При решении задачи декомпозиции графов широко используется рекурсивная бисекция — некоторая общая стратегия разбиения графа на множество частей. В соответствии с ней, сетка последовательно, с помощью $k - 1$ шагов, делится на k частей. На первом шаге сетка делится на две части. Далее, на каждом шаге одна из полученных ранее частей снова делится на две. Конкретные алгоритмы деления фрагмента сетки на две части (алгоритмы бисекции) будут рассмотрены в последующих разделах. Независимо от того, какой именно алгоритм бисекции используется, следует определиться с процедурой вычисления желаемого числа вершин в каждой из формируемых частей графа (в промежуточных фрагментах и в окончательных доменах).

Для разбиения графа, содержащего n вершин, на произвольное число доменов k может быть использован алгоритм A23. На рис. 59 представлен пример разбиения с его помощью 100 вершин графа на 7 частей приблизительно равного размера. В каждой из вершин бинарного дерева (рис. 59) указано желаемое число вершин фрагмента сетки (n) и пропорция ($k1/k2$), в которой это число следует разбить на две части. Отметим слово «желаемое» в предыдущей фразе. Как будет показано в следующем примере,

при некоторых методах бисекции веса сформированных фрагментов сетки могут отличаться от оптимальных значений.

Алгоритм A23

Рекурсивная бисекция

```
RecursiveBisect(graf,n,k)
{
k1=(k+1)/2
k2=k-k1
n1=n*k1/k
n2=n-n1

// разбиение вершин графа на две части
// размером n1 и n2
Bisect(graf,n,subgraf1,n1,subgraf2,n2)

if(k1>1) RecursiveBisect(subgraf1,n1,k1)
if(k2>1) RecursiveBisect(subgraf2,n2,k2)
}
```

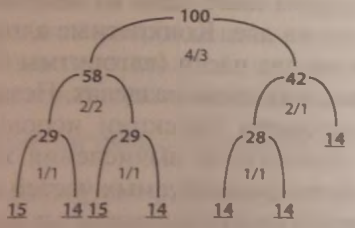


Рис. 59. Бинарное дерево разрезов

7.5. Декомпозиция регулярных графов

Задача декомпозиции сеток трудна не только в том случае, когда граф сетки имеет нерегулярную структуру. Разбиение даже «простейшей» прямоугольной решетки оказывается нетривиальной задачей.

В качестве примера рассмотрим разбиение прямоугольной сетки из 25 узлов (квадрата 5 на 5) на четыре домена. Воспользуемся простым алгоритмом индексной бисекции (алгоритм А24). В рамках алгоритма предполагается, что каждый из фрагментов решетки имеет прямоугольную форму.

Разделим сетку на две части между вторым и третьим столбцами узлов. Получившиеся части будут содержать разное число узлов — в одной части 10 узлов, в другой 15 (рис. 60а). Веса сформированных фрагментов (15 и 10) отличаются от оптимальных (13 и 12) значений. В результате второго разбиения (между вторым и третьим рядами) декомпозиция выполнена, но получен существенный дисбаланс нагрузки процессоров. Одному из процессоров будет назначено на обработку в 2,25 раза больше узлов, чем другому: девять против четырех (рис. 60б). Идеальная декомпозиция может выглядеть следующим образом (рис. 60в): три домена содержат по 6 узлов и один — 7.

Тривиальность алгоритма индексной бисекции оборачивается плохим качеством найденного разбиения. Потенциальный выигрыш во времени выполнения расчетов с помощью идеальной декомпозиции (6, 6, 6, 7), по сравнению с простой декомпозицией (4, 6, 6, 9) равен 9/7 — почти 30%.

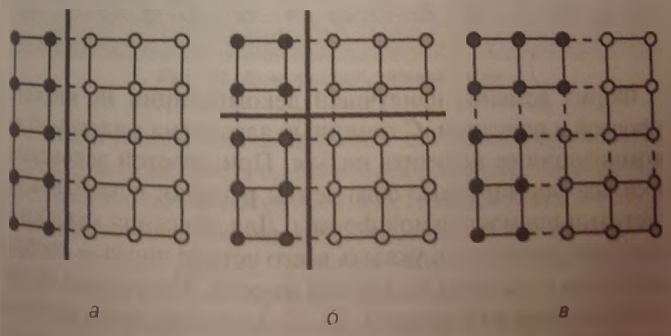


Рис. 60. Разбиение сетки на 4 домена

Алгоритм. A24
Рекурсивная индексная бисекция

```
BisectIndex( left_x, size_x, left_y, size_y, k)
{
  if(k==1)
  {
    // сформирован очередной домен
    // ( left_x, size_x, left_y, size_y)
    return;
  }

  k1=(k+1)/2
  k2=k-k1

  if(size_x>=size_y)
{
  n1=size_x*k1/k
  n2=size_x-n1
    BisectIndex( left_x, n1, left_y, size_y, k1)
    BisectIndex( left_x+n1, n2, left_y, size_y, k2)
}
else
{
  n1=size_y*k1/k
  n2=size_y-n1
    BisectIndex( left_x, size_x, left_y, n1, k1)
    BisectIndex( left_x, size_x, left_y+n1, n2, k2)
}
}
```

Форма доменов наилучшей декомпозиции не является простой в описании. С помощью алгоритма индексной декомпозиции ее получить нельзя. При простой декомпозиции все формируемые фрагменты решетки оставались решетками прямоугольной формы. Для описания каждого из них было достаточно указать всего четыре числа — координаты угла и размеры по каждой из осей. Наилучшая декомпозиция содержит домены, не обладающие таким удобным свойством. Некоторые из доменов могут не иметь прямо-

угольной формы. Для описания формы домена, содержаще-
го 7 узлов (рис. 60в), требуется указать существенно больше
данных, чем для описания прямоугольника, а значит, требу-
ется и другой, более сложный алгоритм декомпозиции.

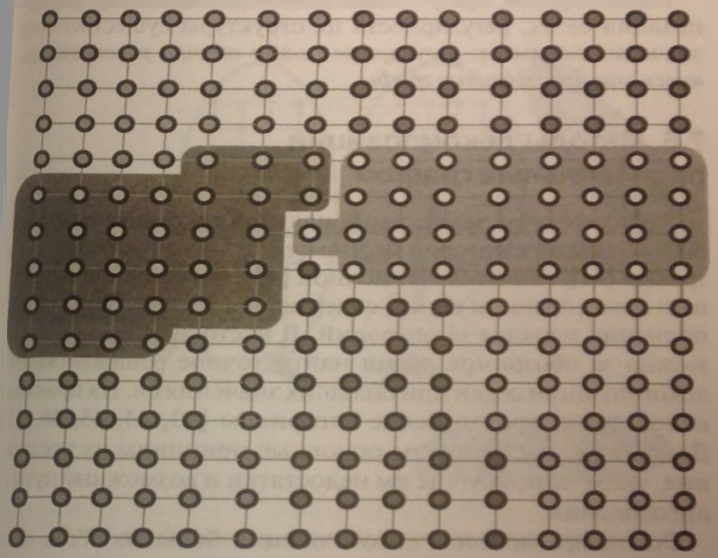


Рис. 61. Домены регулярной сетки

С увеличением числа доменов ситуация усугубляется,
преимущества простоты описания прямоугольной решет-
стремительно теряются (рис. 61). При делении многомер-
их сеток на большое число частей, получаемые фрагменты
рачивают регулярные свойства, потому что их границы
юбретают сложную форму. В связи с этим, даже с пря-
угольной решеткой целесообразно работать как с произ-
ьным графом. При декомпозиции ее следует описывать

множеством пронумерованных узлов и явно заданных между ними связей. Индексная бисекция должна быть заменена другим методом, например, рассматриваемыми далее методами координатной или спектральной бисекции.

Таким образом, при решении задачи рациональной декомпозиции сеток, регулярность их структуры существенного значения не имеет: *регулярные сетки обычно рассматриваются как произвольные графы*.

7.6. Методы декомпозиции произвольных графов

В общем случае задача рациональной декомпозиции, наиболее близкой к которой можно считать задачу о разрезании графов [41], является NP-полной [42]. Вычисление наилучшей декомпозиции сетки, содержащей n узлов, требует выполнения порядка $n!$ операций. В настоящее время невозможно за обозримое время найти точное решение задачи декомпозиции сетки при больших значениях n. На практике используют эвристические алгоритмы [43, 44, 45, 46, 47]. Далее будут рассмотрены основные принципы их построения, указаны присущие им недостатки и возможные пути их преодоления.

Основным методом декомпозиции больших сеток является иерархический подход [48, 49, 50], предполагающий огрубление сетки (построение приближенного образа сетки, содержащего меньшее число узлов), декомпозицию огрубленного образа и последующее уточнение решения на основе локальных критериев.

7.6.1. Иерархическая декомпозиция

Рассмотрим этапы эффективного иерархического подхода (рис. 62):

- огрубление графа: построение последовательности уменьшающихся в размере вложенных графов;

- начальная декомпозиция: вершины огрубленных графов распределяются по заданному числу доменов;
- восстановление графа и локальное уточнение: вершины всех графов сформированной последовательности перераспределяются по доменам.

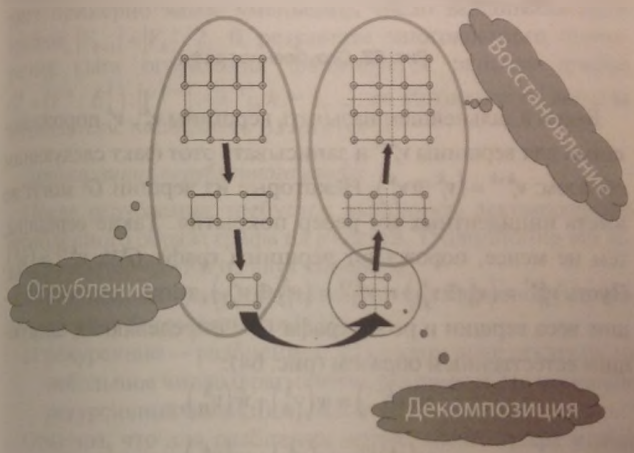


Рис. 62. Общая схема иерархических алгоритмов декомпозиции сеток

Огрубление графа

Алгоритм огрубления графа основан на идее итерационного стягивания ребер (рис. 63). На каждой итерации из всего множества ребер графа G^k выбирается подмножество *ребер покрытия* таким образом, чтобы никакая из вершин сетки не была инцидентна более чем одному из выбранных ребер. Далее каждая пара вершин (v_i^k, v_j^k), соединяемых ребром покрытия e_{ij}^k, стягивается в одну новую вершину v_i^{k+1}.

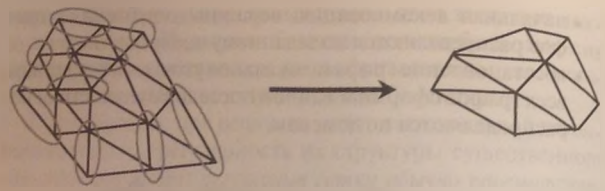


Рис. 63. Огрубление графа

Будем в дальнейшем называть вершины v_i^k, v_j^k порождающими для вершины v_i^{k+1} и записывать этот факт следующим образом: $v_i^{k+1} = \langle v_i^k \oplus v_j^k \rangle$. Некоторые из вершин G^k могут не иметь инцидентных им ребер покрытия. Такие вершины, тем не менее, порождают вершины графа G^{k+1}: $v_i^{k+1} = \langle v_i^k \rangle$. Пусть $v_{i1}^{k+1} = \langle v_{i1}^k \oplus v_{j1}^k \rangle$ и $v_{i2}^{k+1} = \langle v_{i2}^k \oplus v_{j2}^k \rangle$, тогда соответствующие веса вершин и ребер графа G^{k+1} определяются следующим естественным образом (рис. 64):

$$w\left(v_{i1}^{k+1}\right) = w\left(v_{i1}^k\right) + w\left(v_{j1}^k\right),$$

$$w\left(v_{i2}^{k+1}\right) = w\left(v_{i2}^k\right) + w\left(v_{j2}^k\right),$$

$$w\left(v_{i1}^{k+1}, v_{i2}^{k+1}\right) = w\left(v_{i1}^k, v_{i2}^k\right) + w\left(v_{i1}^k, v_{j2}^k\right) + w\left(v_{j1}^k, v_{i2}^k\right) + w\left(v_{j1}^k, v_{j2}^k\right).$$

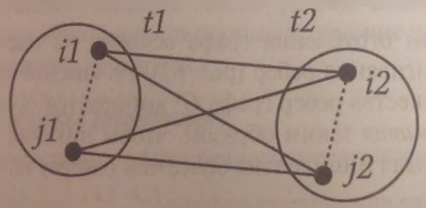


Рис. 64. Стягивание ребер (*i*1, *j*1) и (*i*2, *j*2).

При выборе множества ребер покрытия графа G^k следует отдавать предпочтение ребрам, инцидентным вершинам, имеющим наименьшие веса. Это позволяет уменьшить разброс весов вершин графа G^{k+1} и улучшить качество итогового разбиения.

Однократное выполнение описанной процедуры позволяет примерно вдвое уменьшить число вершин больших графов $|V_{k+1}| \approx |V_k|/2$. В результате многократного повторения шага огрубления формируется цепочка графов $G^k = (V^k, E^k) : |V^{k-1}| > |V^k|$, $k = 1, \ldots, m$. Разбиение на домены начинается с последнего из них (G^m).

Декомпозиция огрубленного графа

После огрубления требуется выполнить декомпозицию огрубленного образа графа на p частей. Традиционно эта задача решается одним из двух способов:

1) разбиением сразу на требуемое число доменов (многопутевая декомпозиция [51]);
2) рекурсивно — разбиением на каждом шаге рекурсии на небольшое число фрагментов, обычно на 2 (с помощью рекурсивной бисекции), 4 или 8.

Отметим, что для разбиения огрубленного графа можно использовать практически любой алгоритм декомпозиции. При достаточно сильном огрублении исходного графа (до небольшого, близкого к p числа вершин), можно использовать алгоритм полного перебора, дающий точное решение. Тем не менее, чем большего размера граф подвергается непосредственному разбиению (чем меньше огрублен исходный граф) и чем выше качество этого разбиения, тем выше качество окончательного результата. Один из наиболее привлекательных методов декомпозиции — спектральный [52], однако, этот метод является и наиболее требовательными с точки зрения вычислительных затрат, поскольку его использование сопряжено с необходимостью вычисления собственных векторов спектральной матрицы графа [50]. Среди

менее затратных методов можно указать метод координатной бисекции и метод заполняющих кривых [53, 54].

Восстановление графа

В результате выполнения этапов огрубления и рекурсивной бисекции формируется разбиение графа G^k на p доменов: $R(V^k)=(V_1^k, ..., V_p^k)$. В качестве начального приближения для разбиения вершин менее огрубленного графа G^{k-1} выберем такое $R(V^{k-1})=(V_1^{k-1}, ..., V_p^{k-1})$, в котором $v_i^{k-1} \in V_r^{k-1}$ тогда и только тогда, когда $v_q^k \in V_r^k$ и либо $v_q^k=(v_i^{k-1} \oplus v_j^{k-1})$, либо $v_q^k=(v_i^{k-1})$. Иными словами, порождающие вершины V^{k-1} распределяются в домены с теми же номерами, что и образованные ими вершины V^k. Полученное начальное разбиение можно существенно улучшить частичным перераспределением вершин между соседними доменами. Для этих целей можно использовать эффективный алгоритм локального уточнения границ доменов.

Локальное уточнение границ доменов

При разбиении огрубленного графа формируются домены несовпадающих весов, так как веса агрегированных (порождённых) вершин могут иметь значительный разброс. Локальное уточнение позволяет выровнять между собой веса доменов и уменьшить число ребер, пересекающих границы доменов (рис. 66).

Одним из эффективных методов локального уточнения является алгоритм Кернигана — Лина [55] (KL-алгоритм, the Kernighan — Lin algorithm). Он является эвристическим и позволяет избежать полного перебора вариантов размещения вершин графа по доменам.

Обозначим через $P(i)$ номер домена, которому принадлежит вершина v_i. С помощью KL-алгоритма можно проверить

выигрыш от перемещения каждой вершины v_i в каждый из доменов, соседних с доменом $P(i)$, и выполнить перемещения, минимизирующие J (28). Домены r и t называются соседними, если существует ребро, соединяющее вершины из этих доменов: $\exists e_{ij} \in E : v_i \in V_r$ и $v_j \in V_t$. Выигрыш от перемещения вершины определяется величиной уменьшения значения величины J. Каждая итерация алгоритма включает следующие действия:

- для каждой вершины v_i определяются величины выигрышей $g_i(k)$, достигаемых в результате перемещения вершины i в домен k:

$$\overline{W} = \frac{\sum_{v_i \in V} w_i}{|V|};$$

$$g_i^0(k) = \left|W_{P(i)} - \overline{W}\right| - \left|W_{P(i)} - \overline{W} - w_i\right| + \left|W_k - \overline{W}\right| - \left|W_k - \overline{W} + w_i\right|;$$

$$g_i^1(k) = \sum_{e_{ij} \in E} \begin{cases} w_{ij}, & P(j) = k, \\ -w_{ij}, & P(j) = P(i), \\ 0, & \text{иначе} \end{cases}$$

$$g_i(k) = g_i^0(k) + g_i^1(k);$$

- определяется некоторое множество пар (i, k), для которых величины $g_i^1(k)$ имеют большие значения;
- для выбранных пар проверяется условие $g_i(k) > 0$ и, если оно удовлетворяется, то вершина i включается во множество перемещаемых на данной итерации вершин H;
- выполняется перемещение вершин выбранного множества H;
- если величина J уменьшилась, то полученное разбиение принимается в качестве текущего и выполняется следующая итерация;
- если уменьшения не произошло, то сделанные перемещения отменяются и выбирается другое множество перемещаемых вершин или, если несколько попыток не

увенчались успехом, оптимальное разбиение считается найденным.

При определении величин $g(k)$ учитывается эффект от перемещения только одной вершины. Отметим, что сумма выигрышей, полученных от перемещения двух вершин по отдельности не равна выигрышу от их одновременного переноса. На рисунке 65a приведен пример графа, разделенного на два домена. Перенос из домена в домен любой из вершин v_2, v_3, v_4, v_5 уменьшает на единицу общее число ребер, пересекающих границу между доменами (рис. 65b). Однако, одновременный перенос всех четырех вершин приводит к увеличению общего числа таких ребер с 4-х до 6-ти, снижая таким образом качество разбиения (рис. 65c).

Возможен и обратный эффект: положительный выигрыш при одновременном переносе нескольких вершин, даже если перенос некоторых из них в отдельности ведет к проигрышу. В связи с этим, случайным образом, допускается перенос нескольких вершин, даже если в результате разбиение частично ухудшается.

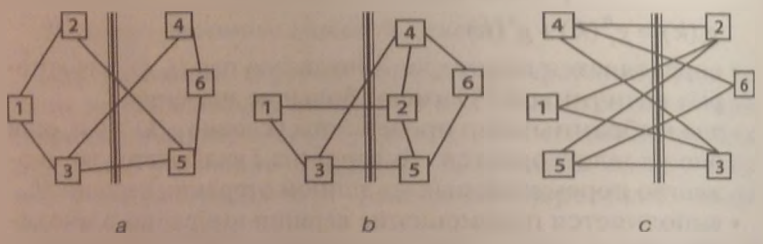


Рис. 65. Перенос вершин при локальном уточнении границ доменов

Перенос каждой вершины приводит к изменению выигрышей от последующих перемещения ее самой и некоторых других вершин. Прямой пересчет всех выигрышей занимает значительное время. На первой итерации соответствующие потери неизбежны, но на последующих итерациях полный

пересчет выигрышей заменяется корректировкой уже вычисленных величин. Основное время требуется для расчета величин $g_i^1(k)$, описывающих выигрыш от изменения числа ребер, соединяющих домены. Общее число необходимых для этого действий оценивается как $T^0 = O\left(p\overline{\rho(v)}|V|\right)$, где $\overline{\rho(v)}$ – средняя степень вершин (среднее число инцидентных вершинам ребер). В реальных расчетных сетках средняя степень вершин не зависит от их общего числа $\left(\overline{\rho(v)} = O(1)\right)$, таким образом, $T^0 = O\left(p|V|\right)$. Именно на основе величин $g_i^1(k)$ происходит предварительный отбор пар (i, k), определяющих H. Величины $g_i^0(k)$ и $g_i(k)$ для большинства вершин не вычисляются совсем. При переносе одной вершины v_j число ребер, пересекающих границы доменов, может измениться только за счет ребер, инцидентных v_j, следовательно, и обновить следует только элементы $g_i^1(k)$, $i \in AH$, соответствующие множеству вершин, соседних с вершинами H. Здесь A – матрица смежности вершин графа. Как правило, $|AH| \ll |V|$, поэтому время перерасчета выигрышей значительно меньше времени полного расчета всех элементов $g_i^1(k)$.

Для эффективной реализации изложенной идеи следует использовать такую дисциплину хранения $g_i^1(k)$, которая не только предоставит быстрый доступ к изменяемым элементам, но и даст возможность быстрого выбора претендентов на перемещение – вершин, обладающих максимальными выигрышами. Для хранения $g_i^1(k)$ можно использовать структуры данных типа «корзинка» B_s [56]. Их применение позволяет успешно использовать хеширование данных [6] – разбиение вершин на непересекающиеся классы по признаку равенства выигрышей от перемещения вершины из одного домена в другой. Максимально возможный для данного графа выигрыш от перемещения между доменами одной вершины g_{max} определяется структурой графа и может быть вычислен заранее (32).

$$g_{max} = \max\left(w(v_i) + \alpha \sum_{v_j \in E} w(v_i, v_j) \right). \quad (32)$$

Потребуется $p^2 - p$ корзинок — по числу возможных перемещений вершины из одного домена в другой. Каждая корзинка содержит $2g_{max} + 1$ двунаправленных списков с номерами вершин домена. В списке B_{ij}^k хранятся номера вершин, перенос которых из домена i в домен j даст выигрыш k. Для каждой вершины хранится информация только об одном перемещении, соответствующем максимальному выигрышу от перемещения этой вершины. Кроме списка корзинок, хранится список свободных вершин — вершин, выбранных для переноса из одного домена в другой. Описанная структура гарантирует выполнение всех основных операций за время $O(1)$:

- выбор очередной вершины, перенос которой приносит максимальный выигрыш;
- перемещение вершины между корзинками;
- определение позиции вершины в корзинке и обновление значений каждого из элементов $g_i^1(k)$.

Таким образом, время выполнения операций перемещения вершин и обновления величин $g_i^1(k)$, необходимое для перемещения всего множества вершин H, оценивается как $T^1 = O(|H|)$. Для планарных сеток величина T^1 не превышает $O(|V|)$, даже если перемещению подлежат все вершины, лежащие на границах доменов. В двумерном случае число вершин, лежащих на границе доменов, можно оценить как $|H| = O\left(p\sqrt{\frac{|V|}{p}}\right)$, поскольку предварительное разбиение к моменту применения алгоритма локального уточнения уже выполнено, и домены имеют не оптимальную, но приближенную к ней форму. Таким образом, $T^1 = O\left(\sqrt{p|V|}\right)$.

Общее время выполнения алгоритма можно оценить величиной

$$T = T^0 + T^1 = O\left(p|V| + \gamma\sqrt{p|V|}\right),$$

где γ — число итераций алгоритма локального уточнения.

Величина γ, значительно влияющая на общее время работы алгоритма, косвенно зависит от выбранной стратегии перемещения вершин и определяется рядом эмпирических правил и параметров, например:

- каждая вершина переносится из одного домена в другой не более одного раза;
- каждую вершину из домена i можно переносить только в те домены, в которых существуют вершины, связанные ребрами с вершинами домена i;
- на каждой итерации переносится не более k вершин, где k выбирается эмпирически, например $k = |V| / 10$.

Алгоритмы, подобные описанному методу локального уточнения, применяются для улучшения качества разбиений, полученных обсуждаемыми в последующих разделах методами.

7.6.2. Спектральная бисекция

Спектральная бисекция — это метод разбиения графа на два домена в заданном отношении весов, минимизирующий суммарный вес ребер, соединяющих вершины из разных доменов, основан на использовании спектральной матрицы графа (матрицы Лапласа) [52, 57, 58] и описывается этапами A–D. Поскольку метод является затратным с вычислительной точки зрения, как правило, он применяется для декомпозиции огрубленной, а не исходной сетки.

Формализуем задачу спектральной бисекции следующим образом. Определим на множестве вершин графа сеточную функцию q. Значение q_i, соответствующее вершине с номером i, может принимать только одно из двух значений: –1 или 1. Будем считать, что вершины, в которых q принимает

значение −1, принадлежат одному домену, а остальные — другому. Очевидно, что число разрезанных (соединяющих вершины из разных доменов) ребер будет равно

$$|E_c| = \frac{1}{4}\sum_{e_{ij}\in E,\, q_i \neq q_j} (q_i - q_j)^2. \qquad (33)$$

Здесь e_{ij} — ребро, соединяющее вершины i и j. Именно величину $|E_c|$ следует минимизировать, но при дополнительном условии, обеспечивающем одинаковое число вершин в обоих доменах:

$$\sum_{i=1}^{|V|} q_i = 0. \qquad (34)$$

Для замыкания системы добавим условие нормировки:

$$\sum_{i=1}^{|V|} q_i^2 = q^T q = |V|. \qquad (35)$$

Описываемый вариант метода спектральной бисекции обеспечивает рациональное разбиение графа на две части, что позволяет, используя алгоритм рекурсивной бисекции, разбить граф на произвольное число частей.

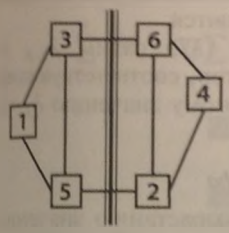
Этап А. Построение спектральной матрицы графа

На первом этапе формируется спектральная матрица l графа G расчетной сетки. Недиагональный элемент матрицы l_{ij}, $i \neq j$ равен −1, если между узлами i, j есть ребро. Диагональный элемент $l_{ii} = \rho_i$, где ρ_i — степень вершины i (36). Остальные элементы равны 0.

$$l_{ij} = \begin{cases} -1, & e_{ij} \in E,\ i \neq j, \\ \sum_{k \neq i} l_{ik}, & i = j, \\ 0, & \text{иначе} \end{cases} \qquad (36)$$

Ниже приведена спектральная матрица (37) графа, содержащего шесть вершин (рис. 66).

Этап В. Определение первого собственного вектора матрицы Лапласа



$$L=\begin{pmatrix} 2 & 0 & -1 & 0 & -1 & 0 \\ 0 & 3 & 0 & -1 & -1 & -1 \\ -1 & 0 & 3 & 0 & -1 & -1 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ -1 & -1 & -1 & 0 & 3 & 0 \\ 0 & -1 & -1 & -1 & 0 & 3 \end{pmatrix} \quad (37)$$

Рис. 66. Граф и соответствующая ему спектральная матрица L

Решение системы $Lq = \lambda q$ дает собственные значения спектральной матрицы. Поскольку матрица Лапласа вырождена, одно из них будет нулевым $\lambda_0 = 0$. Ему соответствует собственный вектор e, все компоненты которого равны 1, так как, с учетом (34), справедливо $Le = 0$.

Принимая во внимание (35), преобразуем минимизируемую квадратичную форму (33):

$$|E_c| = \frac{1}{4} q^T L q = \frac{1}{4} q^T \lambda\, q = \frac{1}{4} \lambda |V|. \quad (38)$$

Таким образом, для минимизации числа разрезанных ребер (38) $|E_c| = \frac{1}{4}\lambda|V|$ при ограничениях (34) и (35) следует найти собственный вектор ψ, соответствующий минимальному ненулевому собственному числу спектральной матрицы $L\psi = \lambda_1 \psi$. Этот вектор (вектор Фидлера [59]) удовлетворяет условию (34), поскольку он ортогонален нулевому собственному вектору e: $e\psi = 0$. Выбрав соответствующую нормировку, легко обеспечить выполнение и второго условия (35).

Все собственные значения, кроме нулевого, матриц Лапласа связных графов положительны [60]. Поставленная задача минимизации (38) при ограничениях (34) и (35) является задачей целочисленного программирования. При ее приближенном решении от требования целочисленности компонент вектора Фидлера отказываются.

Собственные значения матрицы (37) равны $\lambda_{1..6} = \{0, 1, 3, 3, 4, 5\}$. Собственный вектор, соответствующий минимальному ненулевому собственному значению $\lambda_2 = 1$, равен $\psi = (2, -1, 1, -2, 1, -1)$.

Этап C. Упорядочивание вершин графа

Упорядочим номера вершин по возрастанию значений соответствующих им элементов вектора ψ. В результате получим вектор, определяющий новый порядок вершин. В рассматриваемом примере порядок вершин будет следующим: $s = (4, 2, 6, 3, 5, 1)$. Наименьший элемент собственного вектора ψ равен -2 и является четвертым, поэтому в векторе s первым будет элемент номер 4 и т. д.

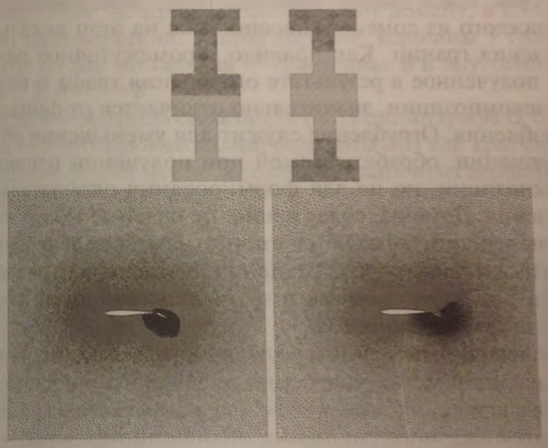
Этап D. Формирование двух доменов

Используя вектор s, можно распределить вершины графа на две части в требуемой пропорции $Q = \frac{n_1}{n_1 + n_2}$, где n_1, n_2 – число вершин, содержащихся в формируемых фрагментах сетки. Первая часть графа должна содержать вершины, имеющие меньшие значения компонент найденного собственного вектора, вторая — имеющие большие значения.

В рассматриваемом примере в первый домен попадают вершины (4, 2, 6) а во второй — (3, 5, 1), что в данном случае соответствует оптимальному разбиению графа (рис. 66) на две равные части. В общем случае, спектральное разбиение является достаточно хорошим приближением к оптимальному решению.

Поскольку вершины сетки обременены весами, в общем случае следует определить такой номер k, для которого выполняются условия (39), и включить в первый домен вершины, имеющие номера s_i, $i = 1, \ldots, k$, а во второй домен — остальные вершины.

$$\sum_{i=1}^{k} w\left(v_{s_i}\right) \le Q \sum_{i=1}^{|V|} w\left(v_i\right) \text{ и } \sum_{i=k+1}^{|V|} w\left(v_{s_i}\right) \le Q \sum_{i=1}^{|V|} w\left(v_i\right). \qquad (39)$$



Спектральная декомпозиция *Иерархическая декомпозиция (пакет Metis)*

Рис. 67. Примеры декомпозиции

На рисунке 67 представлены примеры декомпозиций, полученных пакетом Metis (справа) и спектральным методом (слева).

Аналогичную декомпозицию, но на четыре или восемь частей, можно выполнить с использованием двух или трех векторов Фидлера, соответствующих третьему и четвертому

наименьшим собственным значениям. Соответствующая декомпозиция оптимальна с точки зрения коммуникационных затрат в системе процессоров, соединенных каналами связи в гиперкуб [61].

7.6.3. Алгоритм инкрементного роста

Огрубление графа с помощью алгоритма стягивания ребер приводит к тому, что основная тяжесть принятия решения о том, какие именно вершины будут объединены в рамках каждого из доменов, переносится на этап локального уточнения границ. Как правило, промежуточное разбиение, полученное в результате огрубления графа и начальной декомпозиции, значительно отличается от финального разбиения. Огрубление служит для уменьшения объема информации, обрабатываемой при получении первичной декомпозиции, но не для формирования окончательного разбиения. Домены, содержащие не связные между собой группы вершин, образуются именно на этапах рекурсивной бисекции и локального уточнения, в связи с чем, в рамках рассматриваемого далее метода, направленного на формирование связных доменов:

- не используются этапы огрубления и бисекции; их заменяет иной метод получения начального приближения, повышающий вероятность получения связных доменов;
- этап локального уточнения модифицируется так, чтобы в ходе его выполнения связность доменов не утрачивалась.

Постановка задачи связной декомпозиции

Основная цель предлагаемого метода формирования начального приближения — получение доменов, вершины которых расположены «компактно», «близко» друг к другу. Для формализации этой аморфной формулировки введем понятие ядер графа.

Ядра графа

Анализируя графы сеток, описывающих геометрические расчетные области, можно выделить множество граничных вершин $B \subset V$. Учитывая, что на всем множестве V введено разбиение (28), будем считать вершину $v_i \in V_k$ граничной еще и в том случае, если $\exists e_{ij} : v_j \notin V_k$. Иными словами, вершина является граничной, если она принадлежит множеству граничных вершин всей сетки, или среди ее соседей есть вершины, не принадлежащие тому домену, в который входит сама вершина. Таким образом, вершины V графа G разбиты на два множества — граничных B и внутренних S: $V = B \cup S$, $B \cap S = \phi$.

Пусть A — матрица смежности [62] вершин графа G. Определим глубину произвольной вершины $d(i)$ как кратчайшее расстояние [63] от нее до множества граничных вершин графа: $d(i) = \min_k : A^k v \cap B \neq \phi$.

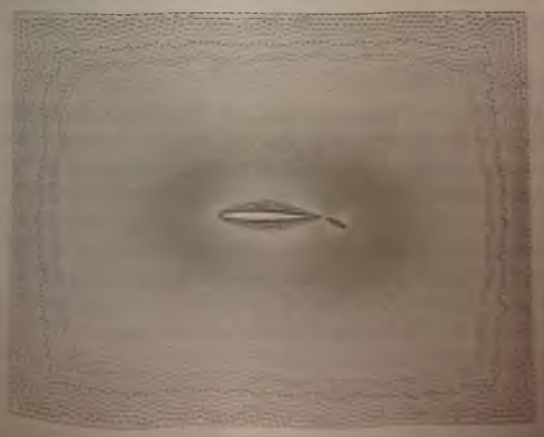


Рис. 88. Слои сетки

Ядро R_k определим как подграф G, образованный вершинами глубины не менее k и ребрами, инцидентными только этим вершинам. Определим слой T_k как подграф G, образованный множеством вершин глубины k и инцидентными только им ребрами. На рисунке 68 показаны слои вершин, определяющие ядра графа. Ядро является связным, если его граф состоит из одной компоненты связности.

Рассматриваемый далее алгоритм инкрементного роста доменов ориентирован на формирование доменов, ядра заданного уровня каждого из которых связны, что является более сильным требованием, чем требование связности каждого из доменов.

Обозначим через $V(G)$ множество вершин графа G. Рекурсивное порождение вершин слоев может быть описано следующим образом:

$$V(T_{k+1}) = \mathrm{A}V(T_k) \setminus V(T_k) \setminus V(T_{k-1}),\ V(T_{-1}) = \phi,\ V(T_0) = B.$$

Оно может быть реализовано с помощью экономичного алгоритма, требующего для порождения слоев с номерами 1, ..., k порядка $O\left(\sum_{j=0}^{k} |T_j|\right)$ действий. Из связности слоя T_k графа G следует связность ядра R_k. Обратное утверждение в общем случае неверно. Например, на рис. 68 слой уровня 0 несвязен (граница представлена тремя фрагментами — прямоугольная замкнутая цепь, окаймляющая всю область и два крыловых профиля внутри области), однако граф является связным. Таким образом, для установления факта связности ядра R_k целесообразно действовать следующим образом:

- сформировать слой T_k и проверить его связность, что может быть выполнено за $O(|T_k|)$ действий;
- в случае несвязности слоя T_k следует проверить наличие цепей, принадлежащих R_k и соединяющих компоненты связности T_k.

На рисунке 69 приведен пример, иллюстрирующий введенное понятие ядра.

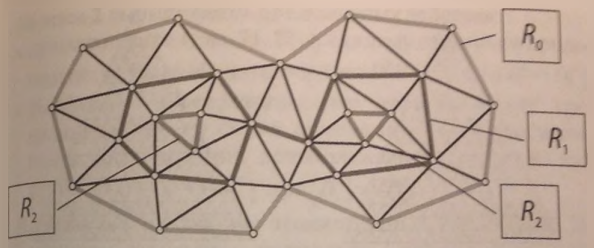


Рис. 69. Ядра R_0 и R_1 связны, ядро R_2 несвязно

Инкрементный алгоритм декомпозиции графа

Рассмотрим инкрементный метод рационального разбиения графов. Наиболее близким его аналогом является пузырьковый алгоритм Дейхмана [64, 65], основанный на идее минимизации радиуса формируемых доменов.

Итерационный инкрементный алгоритм разбиения графа включает в себя следующие этапы:

1) инициализация доменов — определение p случайно выбранных вершин (рис. 70);
2) распределение вершин по доменам методом инкрементного роста (рис. 71, 72);
3) локальное уточнение границ сформированных доменов (рис. 73);
4) анализ связности ядер сформированных доменов (рис. 74а) и окончание работы, если в каждом домене достигнута связность ядер заданного уровня;
5) перенос части закрепленных за доменами вершин в группу свободных вершин (рис. 74б). Возможные действия варьируются от отчуждения нескольких внешних слоев всех доменов до отчуждения практически всех вершин тех доменов, что не удовлетворяют заданным критериям качества. Переход к этапу 2.

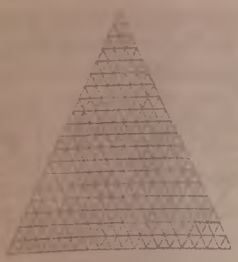


Рис. 70. Инициализация доменов

Рис. 71. Добавление первого слоя вершин в домены

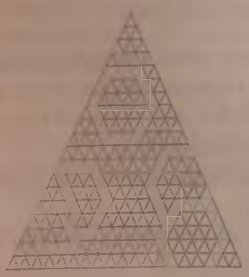


Рис. 72. Добавление второго слоя вершин в домены

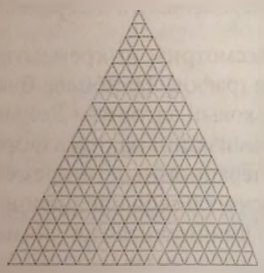


Рис. 73. Окончательное разбиение

Пусть D_i^k — множества вершин, принадлежащих доменам i ($i = 1...p$) на шаге k. Предполагается, что вначале все вершины являются свободными — принадлежат группе свободных вершин $D_0^0 = V$.

На этапе 1 случайно выбранные p вершин (рис. 70) изымаются из группы свободных вершин, и каждая из них включается в отдельный домен D_i. Тем самым, после инициализации каждый из доменов состоит ровно из одной вершины: $|D_i^1| = 1$. Домены могут располагаться произвольным образом, изменяясь в дальнейшем за счет изменения состава образующих их вершин.

На этапе 2 выполняются два основных действия:

- присоединение (рис. 71, 72) к каждому из доменов прилегающих к ним свободных вершин $D_i^{k+1} = D_i^k \cup \left(\mathrm{A} D_i^k \cap D_0^k\right)$;
- диффузное перераспределение вершин между граничащими между собой доменами с целью выравнивания числа вершин, принадлежащих каждому из них $D_i^{k+1} = D_i^k \cup \left(\mathrm{A} D_i^k \cap D_j^k\right)$, при $\left|D_j^k\right| > \left|D_i^k\right|$.

На этапе 3 выполняется традиционный KL-FM [48, 55, 56] алгоритм локального уточнения. Сходные результаты, при значительно меньших затратах времени и оперативной памяти дает применение модифицированного фрагментарного алгоритма локального уточнения. Его суть сводится к последовательному применению алгоритма KL-FM к частям исходного графа H_i. Каждая часть H_i образована объединением подграфов домена i и множества инцидентных ему доменов $H_i = G_i \cup \bigcup G_j,\ j : \exists v \in G_j,\ \mathrm{A} v \cap G_i \neq \phi$. Выигрыш во времени обработки достигается за счет уменьшения числа доменов, обрабатываемых при каждом выполнении процедуры локального уточнения.

Рис. 74. Этапы разбиения сетки на 25 доменов инкрементным алгоритмом

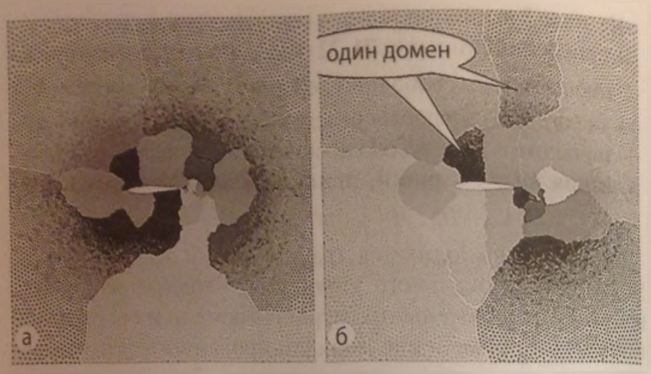


Рис. 75. Разбиение сетки на 25 доменов инкрементным алгоритмом и пакетом Metis

На этапе 5 выполняется частичное сокращение размеров доменов за счет переноса вершин нескольких внешних слоев во множество свободных вершин. Соответствующие действия выполняются, если проверка, выполненная на этапе 4, выявила несоответствие построенных доменов заданным критериям качества. Конкретные возможные действия по уменьшению размеров доменов варьируются от отчуждения от доменов нескольких внешних слоев всех доменов до отчуждения практически всех вершин тех доменов, что не удовлетворяют заданным критериям качества (связности ядер). К примеру, при наличии домена, состоящего более чем из одной компоненты связности (рис. 74а, 75б), выбирается оставляемое в домене подмножество вершин только одной из этих компонент (рис. 74б), все остальные вершины домена переносятся в группу свободных вершин.

Промежуточное разбиение сетки, полученное по окончании этапа 3, приведено на рис. 74а. Результат отчуждения семи внешних слоев всех доменов и одного из фрагментов отмеченного несвязного домена приведен на рис. 74б. Окончательное разбиение на 25 доменов приведено на рис. 75а. Разбиение на 25 доменов с помощью пакета Metis [46] при-

ведено на рис. 75б, отмечен образовавшийся при разбиении несвязный домен, состоящий из двух компонент.

Сравнивая результаты, можно сделать вывод о том, что инкрементный алгоритм, сохраняя высокое качество разбиения, позволяет усилить компактность доменов. Наименьший номер первого несвязного ядра доменов, полученных при разбиении сетки инкрементным методом, равен 11, тогда как в разбиении, полученном иерархическим методом, этот номер равен 0 (рис. 75б), поскольку есть несвязный домен. Полученное в данном примере сокращение общего числа разрезанных ребер (4558 и 4749 ребер соответственно) не является основной целью и не гарантируется в общем случае, хотя и наблюдается во многих экспериментах.

7.7. Декомпозиция больших сеток

В предыдущих параграфах рассматривались последовательные алгоритмы декомпозиции, не пригодные для обработки сеток большого размера. Здесь и далее под большой сеткой понимается сетка, объем описания которой превышает объем оперативной памяти одного процессорного узла. Для декомпозиции больших сеток применяют параллельные алгоритмы, предполагающие, что сетка уже распределена по процессорным узлам. Они эффективны при условии, что вершины распределены по процессорам более или менее компактными группами. Однако, к моменту начала работы алгоритмов декомпозиции, никакая декомпозиция сетки еще не найдена, поэтому требуется некоторый простой способ предварительного распределения графа по выполняющим декомпозицию процессорам. В этом качестве можно использовать рассмотренное ранее распределение согласно номерам вершин. Однако, как уже упоминалось, в результате подобного первичного равномерного распределения вершин на каждом из процессоров окажется множество разрозненных фрагментов сетки. Существенно уменьшить фрагментарность первичного, фактически случайного, распределения

вершин можно с помощью параллельного метода координатной рекурсивной бисекции. Этот метод уступает по качеству формируемой декомпозиции методу спектральной бисекции, но, благодаря высокой скорости работы, простоте и ориентированности на распределенную обработку, его целесообразно использовать на предварительном этапе.

7.7.1. Координатная рекурсивная бисекция

Координатный метод основан на рекурсивной бисекции (алгоритм А23). В качестве процедуры ***Bisect*** используется процедура координатного разбиения ***BisectCoord*** группы вершин на две части (алгоритм А25).

Алгоритм А25

Координатная бисекция

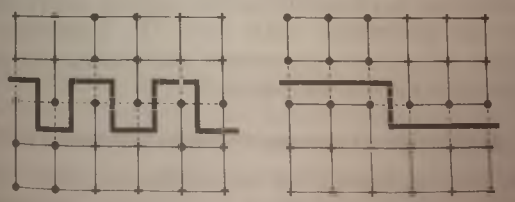
```
BisectCoord( istart, iend, p) // декомпозиция вершины
          // с номерами [istart,iend] на p доменов
{
// Выбрать направление вдоль которого следует
// упорядочить вершины домена
// Упорядичить вершины вдоль выбранного направления
// Вычислить номер первой вершины второго домена:
imiddle=(iend-istart+1)*((p+1)/2)/p
}
```

Предполагается, что все вершины графа пронумерованы единой нумерацией. Каждый из доменов содержит вершины, номера которых принадлежат непрерывному интервалу, поэтому для задания декомпозиции достаточно указать число доменов и номера первых вершин каждого из доменов. В начале работы алгоритма определен единственный домен, содержащий вершины с номерами [*istart,iend*]. В результате выполнения процедуры ***BisectCoord*** формируются два новых домена: с вершинами из диапазонов [*istart,imiddle-1*] и [*imiddle,iend*], размер которых определяется общим числом доменов окончательной декомпозиции.

На каждом шаге вершины сортируются по увеличению значения одной из координат, после чего в первую группу

относят вершины с меньшими значениями этой координаты, а во вторую — с большими. Каждый раз следует выполнить предварительное упорядочивание по каждой из трех координат и выбрать тот вариант, который приводит к меньшему числу разрезанных ребер. Эффективные процедуры параллельной сортировки больших, распределенных по процессорам, массивов рассматриваются в главе 8.

При упорядочивании вершин вдоль некоторого направления следует учитывать расположение вершин не только по этому направлению, но и по другим. В особенности это касается регулярных решеток, поскольку в них значения «поперечных» координат могут совпадать у многих вершин. В результате использования процедур неустойчивой сортировки такие вершины, если данная координата не учитывается, могут быть произвольно перемешаны в отсортированном массиве, что существенно снизит качество декомпозиции. Характерный пример приведен на рисунке 76. Критерии сортировки указаны под каждой из полученных декомпозиций. Левая декомпозиция получена при сортировке с учетом только координаты *y*, а правая — с учетом *y*, и, если *y* координаты совпадают, координаты *x*. Семь разрезанных во втором случае ребер меньше одиннадцати в первом, что подтверждает необходимость учета дополнительных координат при сортировке.



y[i-1]<=y[i]

(y[i-1]<y[i])
или
((y[i-1]==y[i]) и (x[i-1]<x[i]))

Рис. 76. Координатная бисекция

Таблица 9

Сравнение алгоритмов декомпозиции

	геометрическая декомпозиция		ParMETIS	
число доменов	80 000		20 000	
время	21 сек		10 сек	
число вершин в домене	2 612	2 613	2 328	10 932
min / max				
среднее число связей с соседними доменами	14		14	
число некомпактных доменов	229		364	

В качестве примера рассмотрим декомпозицию 200 миллионов узлов по 125 (таблица 9) процессорам с помощью рассмотренного метода координатной рекурсивной бисекции и с помощью пакета ParMetis [46]. Декомпозиция выполнялась на 125 процессорах суперкомпьютера «Чебышев» (МГУ им. М. В. Ломоносова). В первом случае удалось выполнить декомпозицию на большее число доменов (80 тысяч доменов против 20 тысяч) и обеспечить меньший дисбаланс распределения вершин между доменами.

Некоторые из сформированных блоков могут содержать несвязные фрагменты сетки (рис. 77). Процедура параллельного локального уточнения дополнительно улучшает предварительную декомпозицию.

7.7.2. Двухуровневая стратегия обработки и хранения сеток

При моделировании на многопроцессорных вычислительных системах характерно выполнение длительных вычислений с помощью большого количества сравнительно коротких сеансов. Число используемых процессоров может меняться от одного сеанса к другому, что вынуждает многократно решать задачу балансировки загрузки.

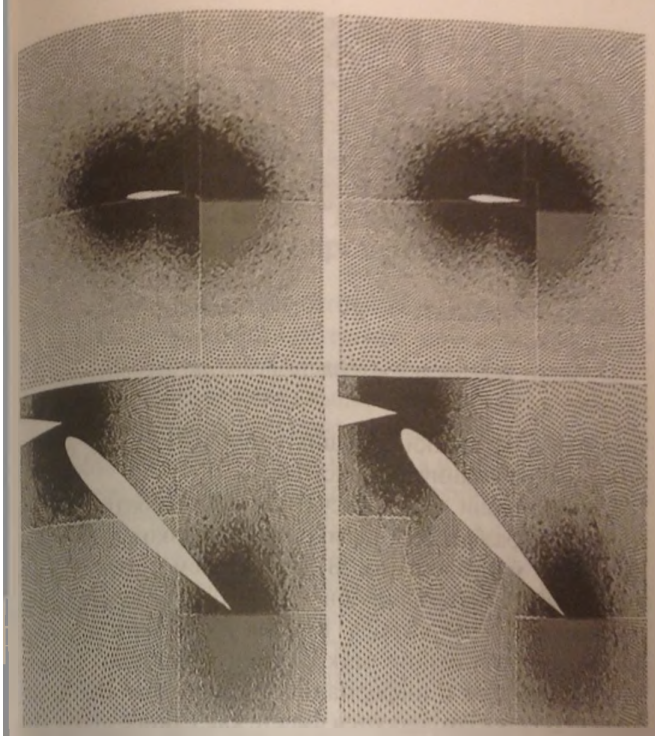


Рис. 77. Координатная бисекция и локальное уточнение

Аналогичная задача возникает при визуализации результатов. Большой объем данных предполагает использование множества процессоров для фильтрации изучаемых данных, следовательно, необходим метод их рационального распределения по выделенным для обработки процессорам. Несмотря на высокую эффективность иерархических алгоритмов, разбиение нерегулярных расчетных сеток большого размера занимает значительное время. Для уменьшения потерь целесообразно использовать *двухуровневую стратегию*

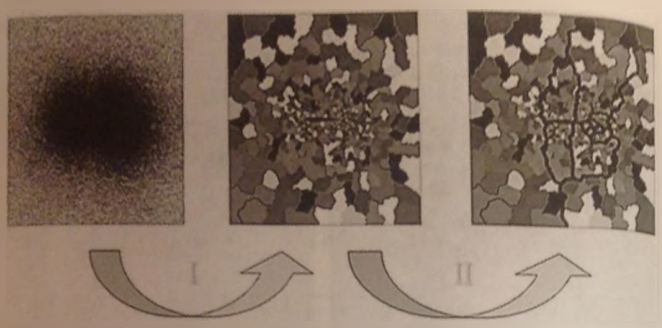


Рис. 78. Иерархическое представление двумерной сетки

хранения и обработки больших сеток [66, 67, 68, 69, 43], проиллюстрированную на рисунке 78.

В соответствии с ней расчетная сетка предварительно однократно разбивается на множество блоков небольшого размера — микродоменов. В дальнейшем сетка хранится в виде набора микродоменов и графа, определяющего их взаимосвязи — макрографа. Каждая из вершин макрографа соответствует одному микродомену. Вершины каждого из микродоменов и информация об определенных на них элементах сетки сохраняются вместе как единое целое. При каждом сеансе расчета производится разбиение макрографа, и каждому процессору назначается список микродоменов. Поскольку в макрографе значительно меньше вершин, чем в исходной сетке, его разбиение требует меньшего времени и может быть выполнено последовательными алгоритмами. Дополнительный выигрыш времени достигается за счет возможности распределенного хранения больших графов как совокупности микродоменов, что значительно уменьшает накладные расходы на чтение и запись сеток большим числом процессоров.

Глава 8

Динамическая балансировка загрузки процессоров

8.1. Стратегии балансировки загрузки

Как правило, значения сеточных функций вычисляются для множества последовательных моментов модельного времени. В общем случае, решение задачи декомпозиции может потребоваться перед выполнением каждого шага вычислений. Например, при моделировании эволюции многокомпонентных смесей в ходе развития моделируемого процесса может меняться состав представленных в узлах сетки веществ. Соответственно, будут изменяться и вычислительные затраты по обработке узлов, что приведет к росту дисбаланса вычислительной нагрузки на процессоры и повлечет за собой необходимость ее перераспределения.

Можно выделить три характерных случая, соответствующих различным стратегиям планирования использования процессорной мощности.

Веса вершин графа фиксированы и не меняются с течением модельного времени (при изменении номера шага по времени) $w_i(t) = \text{const}$. В этом случае допустимо использование алгоритмов статической балансировки загрузки процессоров.

Веса вершин изменяются медленно от шага к шагу по времени: $w_i(t+\Delta t) = \alpha_i(t) w_i(t+\Delta t)$, $\alpha_i(t) \approx 1$. В этом случае правомерно либо использовать рассматриваемые в следующем параграфе диффузионные методы балансировки загрузки, либо периодически выполнять полное перераспределение вершин с помощью алгоритмов декомпозиции сеток.

Веса вершин значительно изменяются от шага к шагу по времени и не могут быть определены заранее $w_i(t+\Delta t) \neq w_i(t+\Delta t)$.

Использование методов статической или диффузной балансировки загрузки не дает в этом случае положительного эффекта. Необходимо использовать методы, отличительной чертой которых является отсутствие априорного планирования распределения вычислений по процессорам. Один из таких методов — серверный параллелизм — рассматривается на примере моделирование горения метанового факела.

8.2. Метод диффузной балансировки

Рассмотрим широко применяемый метод динамической балансировки загрузки процессоров — метод диффузной балансировки загрузки. В его рамках перераспределение вычислительной нагрузки выполняется преимущественно между логически соседними процессорами. Метод эффективен при решении задач с квазистационарным (т. е. медленно меняющимся) распределением вычислительной нагрузки по узлам сетки и позволяет сохранить важное свойство «локальности» вычислительного алгоритма, не требуя непосредственного взаимодействия каждого из процессоров со всеми остальными.

Отталкиваясь от метода геометрического параллелизма, предположим, что некоторое распределение узлов вычислительной сетки по процессорам уже известно, но по каким-либо причинам, оно не является оптимальным. Например, на рисунке 21 в главе о геометрическом параллелизме средний из трех процессоров выполняет назначенный ему объем работ дольше, чем остальные процессоры. Естественным решением является частичное перераспределение заданий (кирпичей или узлов сетки) между этим и соседними с ним процессорами.

Дисбаланс может быть обусловлен неудачным начальным распределением работ, а может быть вызван изменением эффективной производительности процессоров или изменением трудоемкости обработки узлов уже в ходе выполнения работы

1. Не всегда есть возможность априори указать трудоемкость обработки узлов сетки. Например, граничные и внутренние узлы сетки требуют разных времен обработки, соотношение между которыми изначально неочевидно, что делает невозможным точное решение задачи первичной декомпозиции.
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.

Рассмотрим задачу вычисления нового распределения узлов сетки по процессорам при условии, что дано текущее распределение узлов и время, затраченное каждым из процессоров на обработку этих узлов.

Пусть расчетная сетка содержит n узлов, и на шаге j процессор i обработал n_i точек за время t_i. Предполагается, что время 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 n_i' узлов сетки по процессорам, например, следующим образом:

$$n_i' = n\frac{\frac{n_i}{t_i}}{\sum_{i=1}^{p}\frac{n_i}{t_i}}. \qquad (40)$$

Этот подход обеспечивает равномерное распределение работ в том случае, если трудоемкости обработки всех узлов сетки на каждом из шагов по времени равны между собой, а дисбаланс вычислительной нагрузки обусловлен разными производительностями процессоров (причины 2 и 4). Выражение (40) обеспечивает равномерное распределение оди-

наковых по трудоемкости заданий пропорционально производительностям процессоров.

В случае, если процессоры обладают одинаковой производительностью, а дисбаланс обусловлен различной трудоемкостью обработки узлов сетки (причина 1 и 3), подход к перераспределению узлов должен быть другим. Общая вычислительная нагрузка определяется как $T = \sum_{k=1}^{p} t_i$. Для достижения идеальной балансировки на каждый из процессоров следует назначить столько вершин, сколько на нем может быть обработано за время $T_{ideal} = \frac{T}{p}$. Предположим, что все узлы сетки пронумерованы единой нумерацией, причем на каждом процессоре обрабатывается непрерывный диапазон номеров узлов. Тогда на процессоре с номером i обрабатываются узлы с номерами из отрезка $[L_i, R_i]$, где $L_i = \sum_{k=0}^{i} n_k - n_i$, $R_i = L_{i+1} - 1$. Трудоемкость расчета каждой из точек, обработанных на процессоре i, оценивается[1] как $\tau_i = \frac{t_i}{n_i}$. Тогда, $\sigma_j = \tau_{proc(j)}$ — время обработки узла с номером j, где $proc(j)$ — номер процессора, на котором обрабатывалась точка с номером j.

Целью балансировки нагрузки при этих условиях является минимизация θ (41). Суммирование в (41) выполняется по всем узлам сетки, назначенным на процессор i согласно новому распределению.

$$\theta = \min_{i} \max \sum_{j \in proc_i} \sigma_j. \qquad (41)$$

[1] Отметим, что эта оценка справедлива в предположении, что все обрабатываемые одним процессором узлы сетки требуют для своей обработки одинакового времени. В общем случае это не так, поэтому может потребоваться более детализированная оценка времени обработки узлов сетки.

Для приближенного определения нового распределения n_i'' можно воспользоваться алгоритмом А26. В предположении, что изменение трудоемкости обработки узлов сетки происходит медленно, разница между старым и новым распределением будет невелика. Это позволяет ограничиться передачей малых объемов данных между ограниченным числом пар процессоров.

Алгоритм А26

Равномерное распределение заданий разной трудоемкости по процессорам одинаковой производительности

```
m=0; t=0;
for(j=0;j<n;j++)
{
 t=t+σ_j
 if((t≥T_min) {n_i''=j-m; m=j; t=0; }
}
```

Разницу между результатами, полученными при двух разных предположениях (40) и (41) можно оценить, обратившись к таблице 10. В первых двух строках указано исходное распределение узлов сетки по процессорам и время, затраченное каждым из четырех процессоров на обработку. В третьей строке указаны величины n_i', соответствующие новому распределению сетки согласно (40). В четвертой строке указаны величины n_i'', соответствующие новому распределению сетки согласно алгоритму А26. Итак, получены совершенно разные решения, что говорит о важности выяснения на этапе расчета причин возникновения дисбаланса. Знание этих причин позволит обоснованно выбрать стратегию перераспределения вычислений между процессорами, согласно выражению (40), алгоритму А26 или их комбинации.

Таблица 10

Перераспределение узлов сетки

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

С практической точки зрения можно рекомендовать периодическое выполнение на каждом из процессоров, помимо основного расчета, одной и той же короткой вычислительной процедуры с фиксированным числом действий. Сравнение времен выполнения этой процедуры на разных процессорах даст возможность определить фактическое соотношение текущих эффективных производительностей процессоров и обосновано принять решение о методе вычисления нового распределения узлов сетки: (40), (41) или, не рассматриваемое здесь, соотношение более общего вида, учитывающее и разную производительность процессоров, и разную трудоемкость обработки узлов.

На рисунке 79 схематично показано, как могут меняться области, обрабатываемые каждым из процессоров, от шага к шагу по времени (от слоя к слою кирпичей).

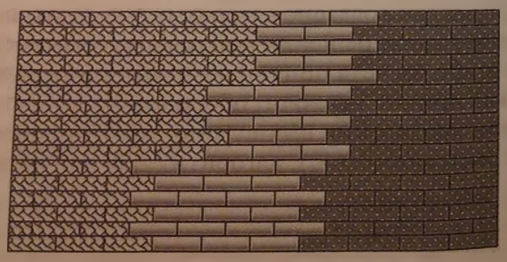


Рис. 79. Диффузная балансировка

Диффузная балансировка загрузки, с одной стороны, способствует сокращению длительности расчета, благодаря выравниванию вычислительной нагрузки приходящейся на каждый из процессоров, а с другой стороны — требует дополнительных затрат времени на сбор статистики о выполненном расчете, на определение нового плана распределения узлов сетки и на само перераспределение узлов. В связи с этим, описанную процедуру диффузной балансировки следует проводить не на каждом шаге расчета, а только тогда, когда накопившийся за множество шагов расчета дисбаланс вызывает снижение эффективности расчета ниже допустимого уровня.

Обратим внимание на то, что, несмотря на локальный характер перераспределения узлов расчетной сетки между процессорами, решение о том, сколько узлов сетки следует оставить на каждом из процессоров, как правило, следует принимать централизовано. Соответствующие затраты времени могут быть оценены как $O(p)$, поскольку требуют вычисления p новых значений n_i'. Использование локальных процедур вычисления величин n_i', основанных на знании сведений о параметрах расчета только на процессорах с номерами $(i-1), (i), (i+1)$ возможно, но точность такого решения ниже.

8.3. Моделирование горения метанового факела

Ранее были рассмотрены используемые для решения сеточных задач на многопроцессорных системах методы геометрического параллелизма и коллективного решения. Идеи, положенные в основу этих методов, отличает простота и элегантность. Не будет преувеличением утверждение о том, что подавляющее большинство решаемых в настоящее время с помощью методов конечных разностей или конечных элементов задач газовой динамики, микроэлектроники, экологии и многих других эффективно решаются именно методом геометрического параллелизма. Метод геометрического параллелизма подразумевает статическую балансировку за-

грузки процессоров. В его рамках заранее определяется обрабатываемая каждым процессором часть пространственной сетки. Статическая балансировка эффективна при условии, что априорной информации достаточно для предварительного равномерного распределения общей вычислительной нагрузки между процессорами. Другую крайность представляет метод коллективного решения, обеспечивающий динамическую балансировку загрузки процессоров. В его рамках заранее неизвестно, какие именно узлы сетки будут обработаны тем или иным процессором. Процессоры получают задания динамически, по мере выполнения уже полученных, что обеспечивает равномерную загрузку процессорных узлов при наличии большого набора независимых заданий.

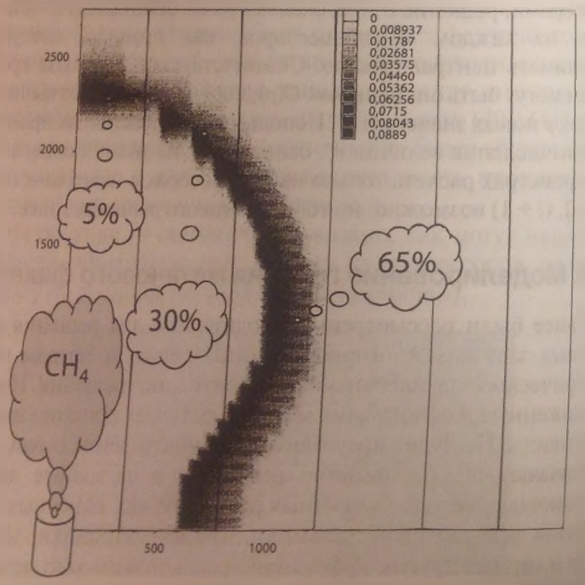


Рис. 80. Разбиение расчетной области по процессорам в соответствии с принципом геометрического параллелизма

Однако, существуют классы задач, решение которых с помощью упомянутых принципов неэффективно. В качестве наглядного примера рассмотрим моделирование с помощью сеточных методов горения поступающей из скважины в атмосферу метановой струи (рис. 80). Будем считать, что моделируется горение в двумерной области, покрытой четырехугольной решеткой расчетных узлов. Рассматриваемая модель химически реагирующих течений предполагает совместное решение уравнений газовой динамики и химической кинетики, описывающих 63 реакции с участием 19 веществ: CH_4, CH_3, CH_3O, O_2, H_2O, H, O, OH, HO_2, CO, CH_2O, CHO, H_2, CO_2, N_2, N, NO, NO_2, N_2O.

В работах [70] на основе методов суммарной аппроксимации [71] и расщепления по физическим процессам [72] разработана методика расщепления системы уравнений, описывающих газодинамические процессы с неравновесной кинетикой, на систему уравнений в частных производных и систему жестких обыкновенных дифференциальных уравнений (ОДУ).

Расщепление (рис. 81) выполнено на блок, описывающий газодинамические процессы (блок уравнений ГД), и блок, описывающий процессы химической кинетики — горения (блок уравнений ХК). Уравнения газодинамического блока описывают диффузию и перенос реагирующих компонент. Уравнения блока химической кинетики описывают кинетику горения — динамику развития в каждом узле расчетной сетки химических процессов.

Блоки ГД и ХК выполняются последовательно, поэтому далее будем говорить об одноименных этапах. Системы обыкновенных дифференциальных уравнений (ОДУ), описывающих кинетику горения в разных узлах сетки, взаимно независимы. Это позволяет выполнять вычисления на этапе ХК во всех узлах одновременно — никакие обмены данными между узлами на этом этапе не нужны. Таким образом, потенциально этап ХК обладает огромным запасом внут-

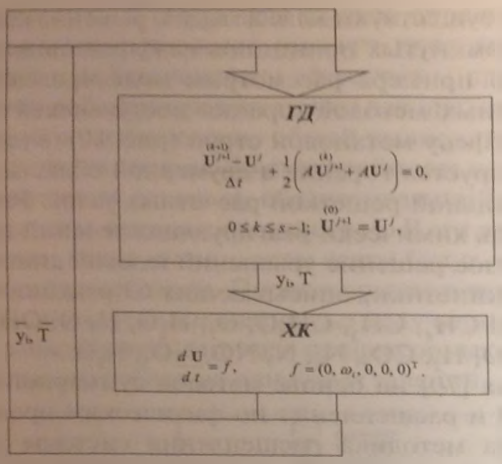


Рис. 81. Блок-схема алгоритма моделирования задач горения

реннего параллелизма и допускает использование стольких процессоров, сколько узлов содержит расчетная сетка. Несмотря на высокий запас внутреннего параллелизма, построение эффективного параллельного алгоритма сопряжено со значительными трудностями. Особенностью рассматриваемых систем ОДУ является значительная разница во времени решения ОДУ, описывающих точки, в которых происходят интенсивные химическое реакции (требуется большое время для определения решения), и тех ОДУ, которые соответствуют точкам, расположенным далеко от фронта пламени. Например, в точках с низкой температурой интенсивные химические реакции не протекают, и для решения требуется близкое к нулю время.

Параллельный алгоритм моделирования ГД блока построен на основе метода геометрического параллелизма и, при равномерном распределении точек по процессорам, обеспечивает их равномерную загрузку на этом этапе. Но

распараллеливание блока ХК на основе метода геометрического параллелизма крайне неэффективно по нижеследующим причинам.

1. Узлы расчетной сетки, в которых протекают интенсивные химические реакции («горячие» точки), локализованы в сравнительно узкой зоне — на фронте пламени и непосредственно за ним (рис. 80). Как правило, точки, описывающие фронт пламени, сосредоточены на сравнительно небольшом числе процессоров. В результате, время обработки блока химической кинетики значительно больше для процессоров, содержащих «горячие» точки (рис. 82), по сравнению со временем, затрачиваемым процессорами, моделирующими удаленные от горящего факела области.
2. Положение фронта пламени меняется с течением времени, и «горячие» точки на разных шагах расчета могут располагаться на разных процессорах.

Рис. 82. Распределение нагрузки по расчету «горячих» точек по процессорам

3. Поскольку скорости реакций зависят от температуры и величины концентраций веществ, время решения ОДУ тоже сильно меняется как от точки к точке в пространстве, так и в каждой точке на разных шагах по времени. Время решения системы ОДУ в каждой из «горячих» точек фактически не поддается априорной оценке. Можно лишь утверждать, что при низких температурах время решения мало. В дальнейшем этот факт будет использован для исключения заведомо лишних операций перераспределения заданий.

Перечисленные факторы делают неприменимыми для распараллеливания блока ХК и метод геометрического параллелизма, и другие методы статической балансировки загрузки.

Известный метод динамической балансировки загрузки — метод коллективного решения — предполагает наличие некоторого управляющего процессора, содержащего все необходимые исходные данные (концентрации веществ, температуру и т. д.) для всех расчетных точек. Метод коллективного решения неэффективен для распараллеливания блока ХК в силу высокой скорости миграции «горячих» точек. Непосредственное использование метода коллективного решения сводится к выполнению на каждом шаге по времени следующей последовательности действий:

- передача всех данных, описывающих задания, от всех процессоров на управляющий процессор;
- обработка всех заданий в соответствии с принципами коллективного решения;
- обратное распределение вычисленных результатов по всем процессорам для выполнения моделирования этапа газовой динамики.

Перечисленные действия снижают эффективность распараллеливания до неприемлемого уровня. Основная причина этого кроется в наличии «узкого горла» — единственного управляющего процессора, что требует передачи большого объема данных на каждом шаге и ограничивает возможности обрабатывающих процессоров всей системы.

Альтернативой является рассматриваемый далее алгоритм серверного параллелизма. Он основан на принципе коллективного решения, но в значительной мере свободен от его недостатков, благодаря наделению каждого из процессоров управляющими функциями. Алгоритм удовлетворяет следующим основным требованиям: каждым процессором осуществляется преимущественная обработка локальных «горячих» точек; обмен данными выполняется одновременно с обработкой доступных «горячих» точек; обеспечена возможность вторичного перераспределения точек, что позволяет передавать их большими группами, при необходимости динамически разбиваемыми на более мелкие части.

8.3.1. Постановка задачи динамической балансировки

Требуется разработать алгоритм, обеспечивающий обработку за наименьшее время совокупности заданий, при следующих условиях:

- в начале вычислений блока ХК на каждом из процессоров хранится некоторое количество заданий n_i;
- время, требуемое для выполнения каждого из заданий (в нашем случае — решения одной системы ОДУ, относящейся к одному узлу расчетной сетки) t_i^j, $j = 0, ..., n_i - 1$, заранее не известно;
- по завершении вычислений результаты расчета каждого из заданий должны быть размещены на том процессоре, на котором задание располагалось в начале вычислений.

Несложно получить минимальную и максимальную оценку времени обработки заданий на p процессорах.

Минимальное время достигается при равномерном распределении всей вычислительной нагрузки по доступным процессорам (рис. 82):

$$T_{\min} = \frac{\sum_{i=0}^{p-1}\sum_{j=0}^{n_i-1} t_i^j}{p}.$$

Максимальное время достигается при отсутствии перераспределения заданий между процессорами в ходе вычислений. В этом случае на каждом процессоре будут обработаны ОДУ, связанные с узлами сетки, обрабатываемыми на процессоре на этапе ГД:

$$T_{\max} = \max_i \sum_{j=0}^{n_i-1} t_i^j .$$

Таким образом, максимальный выигрыш (42) во времени выполнения, при большом дисбалансе распределения по процессорам трудоемких элементарных заданий, может достигать значительной величины.

$$\frac{T_{\max}}{T_{\min}} = p \frac{\max_i \sum_{j=0}^{n_i-1} t_i^j}{\sum_{i=o}^{p-1} \sum_{j=0}^{n_i-1} t_i^j}. \qquad (42)$$

8.3.2. Алгоритм серверного параллелизма

Изложим положения, послужившие основой для создания алгоритма решения подобных задач на многопроцессорных системах с распределенной памятью. Рассматриваемый метод серверного параллелизма является развитием метода коллективного решения, но не требует наличия управляющего процессора, позволяя избежать:

- потерь времени на пересылку всех данных управляющему процессору и обратно;
- последовательного характера распределения заданий единственным процессором.

Сформулируем ряд дополнительных требований.

- Следует осуществлять преимущественную обработку каждым из процессоров локальных «горячих» точек (точек, хранящихся на данном процессоре). Точки с других

процессоров запрашиваются, только если все локальные обработаны или переданы на обработку другим процессорам.

- При наличии на процессоре необработанных «горячих» точек не следует инициировать поиск свободных процессоров для их обработки, хотя бы потому, что свободных процессоров может не оказаться.
- Опрашивать процессоры в поисках необработанных точек следует только в том случае, если на процессоре нет готовых к обработке точек.
- Поиск и передача необработанных точек и прием обработанных точек должны выполнятся одновременно с их обработкой. Эти два процесса (обработки и перераспределения точек) по возможности не должны мешать друг другу.
- Должна быть реализована возможность вторичного перераспределения точек, полученных для обработки одним процессором от другого, что позволит передавать точки большими группами, не опасаясь, что точки, требующие большого времени обработки, будут сконцентрированы только на небольшом числе процессоров, в то время как остальные процессоры будут простаивать.
- Поскольку допускается вторичное перераспределение точек, потребуем, чтобы точки возвращались не тому процессору, который непосредственно предоставил точки для обработки, а тому, на котором они были размещены изначально — процессору — «владельцу» точек.

К примеру, пусть шесть из восьми точек были переданы с первого процессора на третий (рис. 83), а затем четыре из них — с третьего процессора на пятый. Тогда после окончания вычислений все точки должны быть переданы непосредственно на первый процессор: с третьего на первый и с пятого на первый, минуя третий процессор.

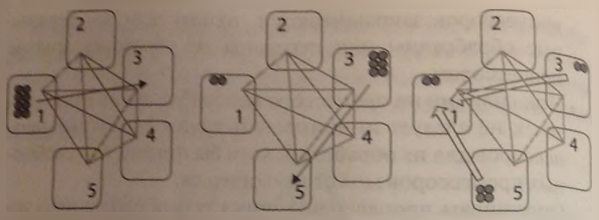


Рис. 83. Перераспределение точек между процессорами

Структура параллельного алгоритма

Структура алгоритма при использовании 5-ти процессоров приведена на рис. 84. На каждом из процессоров одновременно выполняются два параллельных процесса – управляющий и обрабатывающий.

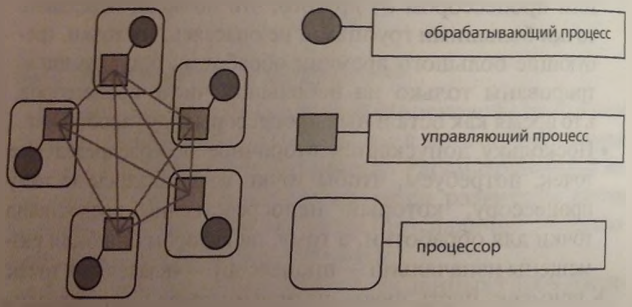


Рис. 84. Взаимодействие процессов

Необходимость запуска на каждом процессоре не только обрабатывающего, но и управляющего процесса, обусловлена необходимостью одновременного выполнения двух существенно разных задач:

- выполнение вычислений;
- перераспределение «горячих» точек от тех процессоров, на которых их избыток, к тем, на которых их нет.

Проблема заключается в том, что процессор, на котором есть необработанные «горячие» точки, должен иметь возможность ответить на поступившие запросы от других процессоров, одновременно выполняя вычисления. Можно было бы, оставаясь в рамках одного процесса, «периодически» проверять факт поступления запроса. Но на этом пути возникает ряд трудно разрешимых проблем. При высокой частоте таких проверок непроизводительно тратится вычислительный ресурс процессора, даже когда запросов нет. При редких проверках запрашивающие процессоры будут непроизводительно простаивать, длительное время находясь в состоянии ожидания. Однако, еще важнее третье обстоятельство. Как правило, для решения систем жестких ОДУ используются сторонние солверы — библиотеки программ, исходные тексты которых могут быть недоступны. В результате, проверки можно будет делать только между обращениями к солверу, следовательно, время отклика будет равно времени решения очередной системы ОДУ, что приведет к большим простоям ожидающих процессоров. Более того, даже если исходные тексты доступны, потребуется определить в сложных кодах именно те места, в которых следует выполнять проверку поступления запросов. Отметим, что подобные коды реализуют сложную логику адаптивной процедуры выбора шага интегрирования. Найденные места должны отличаться тем, что размещенные в них проверки будут выполняться с предсказуемой регулярностью, что на практике труднодостижимо.

Значительно проще использовать предложенный подход, основанный на разделении обрабатывающего и коммуникационного процессов.

Итак, на процессоре с номером i запущены два процесса: управляющий M_i и обрабатывающий S_i. Основной цикл

действий, выполняемых обрабатывающим процессом S_i, выглядит следующим образом:

- ожидание сообщения от управляющего процесса M_i;
- если сообщение содержит указание о необходимости завершить работу, то обрабатывающий процесс завершает работу;
- если сообщение содержит данные, описывающие очередное задание на решение системы ОДУ, то обрабатывающий процесс выполняет численное интегрирование соответствующей системы ОДУ и направляет управляющему процессу M_i сообщение, содержащее результат расчета.

Таким образом, обрабатывающий процесс взаимодействует только с процессом M_i и не взаимодействует с остальными процессами.

В отличие от обрабатывающего, управляющий процесс не производит расчетов. Он связан с каждым из управляющих процессов, расположенных на других процессорах, и со своим обрабатывающим процессом. Перераспределение точек по процессорам выполняется именно на уровне управляющих процессов. Между собой управляющие процессы полностью эквивалентны и выполняют один и тот же алгоритм, основой которого является цикл взаимодействия процессов.

Управляющий алгоритм балансировки загрузки

Будем придерживаться следующей терминологии.

Точка — набор данных, описывающих одну «горячую» точку.

Необработанная точка — точка, которая в данный момент не передана обрабатывающему процессу для обработки и не передана никакому другому процессору. Обратим внимание, что согласно этой терминологии, точка перестает быть необработанной с момента ее предоставления обрабатыва-

ющему процессу. Реальные вычисления по решению системы ОДУ могут быть завершены значительно позже.

Обрабатываемая точка — точка, переданная для обработки обрабатывающему процессу. Точка перестает быть обрабатываемой и становится *обработанной* после того, как решение системы ОДУ уже завершено и управляющий процесс получил от обрабатывающего процесса результат вычислений.

Локальная точка — точка называется локальной по отношению к какому-либо процессору, если она была расположена на нем в начале выполнения алгоритма (на каждом временном шаге). Будем называть процессор *локальным* по отношению к точке, если соответствующая точка локальна по отношению к этому процессору.

Внешняя точка — точка называется внешней по отношению к какому-либо процессору, если она не является по отношению к нему локальной.

Переданная точка — локальная точка, предоставленная для обработки другому процессору. Переданная точка, как правило, не обрабатывается на процессоре, для которого она является локальной. Исключение составляет случай, когда в результате вторичного перераспределения точек, некоторые из переданных точек вернулись на локальный процессор в качестве необработанных.

Обрабатывающий процесс может находиться в одном из следующих состояний:

занят — если установлен соответствующий флаг. Этот флаг устанавливается перед передачей обрабатывающему процессу необработанной точки (неважно локальной или внешней) и сбрасывается после того, как точка уже обработана, и управляющий процесс получил от обрабатывающего процесса результат;

свободен — если не занят, т.е. готов к получению очередной свободной точки.

В основе алгоритма лежит бесконечный цикл, в котором каждый из управляющих процессов выполняет следующие действия:

1) *если*
   - есть необработанные точки (неважно локальные или внешние) *и*
   - обрабатывающий процесс свободен,

   то
   - установить *флаг обрабатываемой точки*,
   - одна из необработанных точек передается на обработку обрабатывающему процессу.

2) *если*
   - нет локальных необработанных точек *и*
   - нет внешних точек *и*
   - нет обрабатываемых точек *и*
   - *флаг запроса на получение необработанных точек* не установлен *и*
   - есть процессоры, которые еще не ответили, что не могут предоставить точки для обработки (соответствующий *флаг запрета обменов* не установлен),

   то
   - послать запрос на получение необработанных точек одному из таких процессоров *и*
   - установить *флаг запроса на получение необработанных точек*,

   иначе

3) *если*
   - все переданные точки получены обратно обработанными *и*
   - от всех процессоров получено сообщение о том, что точки для обработки предоставлены быть не могут *и*
   - всем процессорам послано сообщение о том, что точки для обработки предоставлены быть не могут,

   то

   завершение работы

4) получить очередное сообщение от любого процессора или от своего обрабатывающего процесса;
5) обработать полученное сообщение;
6) перейти к началу цикла (п. 1).

Таким образом, каждый из процессоров заканчивает выполнение алгоритма при выполнении *всех* следующих условий:

a) нет локальных необработанных точек;
b) нет внешних точек;
c) нет обрабатываемых точек;
d) всем процессорам был послан запрос на получение необработанных точек;
e) всем процессорам было послано сообщение о том, что необработанные точки предоставлены быть не могут;
f) от всех процессоров получено сообщение о том, что необработанные точки предоставлены быть не могут;
g) все локальные точки обработаны и получены результаты обработки всех переданных точек.

Рассмотрим подробнее обработку сообщений внутри цикла (п. 5).

Сообщение от обрабатывающего процесса:

сообщение «очередная точка обработана»

сбросить *флаг обрабатываемой точки*
если обработана локальная точка,
то
  записать результат обработки в массив локальных точек,
иначе (обработана внешняя точка)
  записать результат обработки в массив внешних точек
  если внешних необработанных точек больше нет,
  то вернуть точки их локальному процессору

Сообщения от управляющего процесса процессора P

сообщение «запрос на получение необработанных точек»

если есть необработанные точки,
то
передать часть необработанных точек процессору *P*,
иначе
послать процессору *P* сообщение о том, что точки предоставлены быть не могут

сообщение «возврат обработанных точек»

получить точки
записать их в массив локальных точек

сообщение «точки для обработки»

сбросить *флаг запроса на получение необработанных точек*
получить точки для обработки

сообщение «точки для обработки не могут быть предоставлены»

установить *флаг запрета обменов* с процессором *P*.

Изложенный алгоритм не предусматривает ни централизованного сбора данных управляющим процессором, как при коллективном решении, ни централизованного управления путем сбора информации о наличии «горячих» точек и текущей загрузке процессоров. Компенсируется это созданием свободного «рынка» для перераспределения загрузки, где занятые счетом процессоры выступают как поставщики, свободные — как потребители, а товаром являются данные, требующие обработки.

Взаимодействие в основном цикле выглядит следующим образом.

Каждый из процессоров либо начинает обработку одной из необработанных точек, либо, если необработанных точек

нет, посылает запрос одному из процессоров на получение необработанных точек.

Далее процессоры переходят к ожиданию получения какого-либо сообщения. Сообщение обязательно поступит — либо от обрабатывающего процессора в момент окончания обработки точки, либо от процессора, которому ранее был послан запрос, либо от процессора, не имеющего необработанных точек. Даже если два или более процессора послали запросы друг другу, запросы будут получены адресатами, поскольку передача сообщений осуществляется асинхронно. Прием сообщений осуществляется синхронно, по одному сообщению на каждом витке цикла.

При построении алгоритма важно гарантировать отсутствие никем не принятых сообщений. Поэтому на каждый запрос предусмотрено ответное сообщение, что существенно снижает вероятность возникновения тупиковых состояний.

Контроль возникновения тупиков

Отсутствие тупиков обеспечивается согласованностью актов передачи и приема данных для сообщений каждого вида. Однако, поскольку формального доказательства невозможности возникновения тупикового состояния не представлено, а выполненное тестирование, как уже отмечалось, никаких гарантий не дает, целесообразно отдельно рассмотреть метод, позволяющий идентифицировать возникновение тупика как на этапе отладки, так и на этапе эксплуатации разработанного приложения.

Эффективен подход, предусматривающий запуск на каждом процессоре дополнительного контролирующего легковесного процесса, — нити, имеющей доступ к глобальным переменным управляющего процесса (рис. 85). Наличие доступа к глобальной памяти управляющего процесса является существенным требованием. Разместим в глобальной памяти такие ключевые параметры, как число обработанных точек, число принятых и отправленных запросов и им

Сообщения от управляющего процесса процессора P

сообщение «запрос на получение необработанных точек»

если есть необработанные точки,
то
передать часть необработанных точек процессору P_i
иначе
послать процессору *P* сообщение о том, что точки предоставлены быть не могут

сообщение «возврат обработанных точек»

получить точки
записать их в массив локальных точек

сообщение «точки для обработки»

сбросить *флаг запроса на получение необработанных точек*
получить точки для обработки

сообщение «точки для обработки не могут быть предоставлены»

установить *флаг запрета обменов* с процессором *P*.

Изложенный алгоритм не предусматривает ни централизованного сбора данных управляющим процессором, как при коллективном решении, ни централизованного управления путем сбора информации о наличии «горячих» точек и текущей загрузке процессоров. Компенсируется это созданием свободного «рынка» для перераспределения загрузки, где занятые счетом процессоры выступают как поставщики, свободные — как потребители, а товаром являются данные, требующие обработки.

Взаимодействие в основном цикле выглядит следующим образом.

Каждый из процессоров либо начинает обработку одной из необработанных точек, либо, если необработанных точек

нет, посылает запрос одному из процессоров на получение необработанных точек.

Далее процессоры переходят к ожиданию получения какого-либо сообщения. Сообщение обязательно поступит — либо от обрабатывающего процессора в момент окончания обработки точки, либо от процессора, которому ранее был послан запрос, либо от процессора, не имеющего необработанных точек. Даже если два или более процессора послали запросы друг другу, запросы будут получены адресатами, поскольку передача сообщений осуществляется асинхронно. Прием сообщений осуществляется синхронно, по одному сообщению на каждом витке цикла.

При построении алгоритма важно гарантировать отсутствие никем не принятых сообщений. Поэтому на каждый запрос предусмотрено ответное сообщение, что существенно снижает вероятность возникновения тупиковых состояний.

Контроль возникновения тупиков

Отсутствие тупиков обеспечивается согласованностью актов передачи и приема данных для сообщений каждого вида. Однако, поскольку формального доказательства невозможности возникновения тупикового состояния не представлено, а выполненное тестирование, как уже отмечалось, никаких гарантий не дает, целесообразно отдельно рассмотреть метод, позволяющий идентифицировать возникновение тупика как на этапе отладки, так и на этапе эксплуатации разработанного приложения.

Эффективен подход, предусматривающий запуск на каждом процессоре дополнительного контролирующего легковесного процесса, — нити, имеющей доступ к глобальным переменным управляющего процесса (рис. 85). Наличие доступа к глобальной памяти управляющего процесса является существенным требованием. Разместим в глобальной памяти такие ключевые параметры, как число обработанных точек, число принятых и отправленных запросов и им

подобные. Основная задача контролирующего процесса – периодический опрос и анализ изменения состояния этих переменных. В промежутках между опросами процесс переводится средствами операционной системы в спящее состояние. Интервал между проверками переменных может быть выбран достаточно произвольно, например, он может быть равен нескольким секундам. Важно, что контролирующий процесс не модифицирует ключевых параметров, поэтому синхронизация в момент выполнения проверки не требуется.

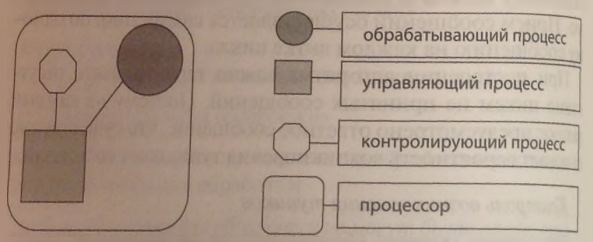


Рис. 85. Контролирующий процесс

В общем случае обсуждаемый метод позволяет идентифицировать два варианта нежелательного поведения программы:

- возникновение тупика;
- возникновение бесконечного цикла, не выполняющего полезных вычислений.

И в том, и в другом случае перестанет меняться как минимум один из параметров — число обработанных точек. При обнаружении подобной ситуации, контролирующий процесс должен вывести для последующего анализа значения и частичную историю изменения контролируемых переменных, после чего принудительно завершить выполнение программы. Отсутствие синхронизации между контролирующим и коммуникационным процессами гарантирует практически полное отсутствие замедления расчета, вызванное запуском

дополнительного контролирующего процесса. Это обстоятельство позволяет использовать данный метод не только на стадии отладки, но и на стадии эксплуатации программы.

Эффективность метода

В заключение раздела приведем график (рисунок 86) эффективности расчетов, выполненных на первой Российской терафлопной системе МВС-1000М (МСЦ РАН). Немонотонность кривой можно объяснить недетерминированным характером распределения вычислительной нагрузки, в результате которого некоторым процессорам передаются более трудоемкие пакеты заданий.

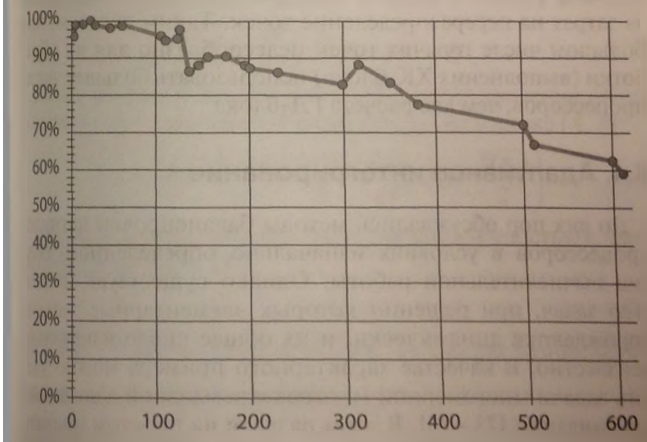


Рис. 86. Эффективность, сетка 1000×1000 узлов

При расчете газодинамических процессов с увеличением числа процессоров относительное влияние затрат на передачу данных возрастает, и эффективность расчета снижается. Начиная с некоторого значения числа процессоров, его

дальнейшее увеличение не приводит к сокращению времени решения задачи. Наблюдается эффект «насыщения», — выхода кривой ускорения на предельный уровень, после чего следует спад производительности.

При расчете ОДУ химической кинетики ситуация принципиально иная. Расчет проводится в каждой точке локально, и необходимость обмена данными между процессорами отсутствует. Поэтому причин для ограничения производительности, обусловленных обменами между обрабатывающими процессорами, нет. Эффективность расчета может уменьшаться, только из-за простоев процессоров, связанных с их неравномерной загрузкой, вызванной различием времени расчета в разных пространственных точках и из-за затрат на перераспределение точек. Таким образом, при большом числе горячих точек целесообразно для их обработки (выполнения ХК-блока) использовать большее число процессоров, чем для расчета ГД-блока.

8.4. Адаптивное интегрирование

До сих пор обсуждались методы балансировки нагрузки процессоров в условиях изначально определенного объема вычислительной работы. Однако существует множество задач, при решении которых элементарные задания порождаются динамически, и их общее число изначально неизвестно. В качестве характерного примера можно указать задачи многомерной многоэкстремальной адаптивной оптимизации [73—75]. В этом разделе на простом примере обсуждается метод динамической балансировки загрузки процессоров, применимый именно в подобном случае при динамическом порождении заданий непосредственно в ходе выполнения вычислений. Отличие метода от ранее рассматриваемых заключается в полном отсутствии управляющих процессов, обеспечивающих распределение работ. Балансировка осуществляется за счет равноправного досту-

та обрабатывающих процессов к общему, динамически пополняемому пулу заданий.

На примере задачи вычисления с точностью ε определенного интеграла (43) рассмотрим алгоритмы, минимизирующие время вычислений на многопроцессорной системе с общей памятью. Их использование обеспечивает снижение времени решения задачи относительно «наилучшего» доступного последовательного алгоритма:

$$J(A,B)=\int_A^B f(x)dx. \qquad (43)$$

Пусть на отрезке $[A, B]$ задана равномерная сетка, содержащая $n+1$ узел:

$$x_i = A+\frac{B-A}{n}i,\ i=0,\ \dots,\ n. \qquad (44)$$

Тогда, согласно методу трапеций, можно численно найти определенный интеграл от функции $f(x)$ на отрезке $[A, B]$:

$$J_n(A, B)=\frac{B-A}{n}\left(\frac{f(x_0)}{2}+\sum_{i=1}^{n-1}f(x_i)+\frac{f(x_n)}{2}\right). \qquad (45)$$

Будем полагать значение J найденным с точностью ε, если выполнено условие (46):

$$|J_{n_1}-J_{n_2}|\le\varepsilon|J_{n_2}|,\ n_1>n_2, \qquad (46)$$

где n_1 и n_2 — количество узлов, образующих две разные сетки.

8.4.1. Последовательные алгоритмы

Метод трапеций

Рассмотрим алгоритм, реализующий метод трапеций:

Алгоритм А27

Последовательный алгоритм «метод трапеций»

```
IntTrap(A,B)
{
...
}
```

```
J2n=(f(A)+f(B))(B-A)/2

do
{
Jn= J2n

n=2n

s=f(A)+f(B)

for(i=1;i<n;i++)
s+=2f(A+(B-A)i/n);

J2n=s(B-A)/n;
}
while(|J2n- Jn|≥ ε J2n)

return J2n
}
```

Алгоритм A27 неэффективен в силу ряда причин, среди которых выделим две:

- в некоторых точках значение подынтегральной функции вычисляется более одного раза. Например, в точке A функция будет вычислена k раз, где k — число выполнений цикла *do — while*;
- на всем интервале интегрирования используется равномерная сетка, тогда как число узлов сетки на единицу длины на разных участках интервала интегрирования, необходимое для достижения заданной точности, зависит от вида функции $f(x)$. Например, число точек, необходимых для достижения заданной точности при определении интеграла (48) возрастает при $A \to 0$, несмотря на уменьшения длины отрезка интегрирования (рис. 87, табл. 10).

Модифицируем алгоритм так, чтобы избежать указанных недостатков.

Метод рекурсивного деления

В дальнейшем будем рассматривать интегрирование функции (47), вид которой приведен на рисунке 87. Точное значение интеграла (43) от этой функции выражением (48)

$$f(x)=\frac{1}{x^2}\sin^2\left(\frac{1}{x}\right),\ 0<A<<1. \qquad (47)$$

$$I(A,B)=\int_A^B \frac{1}{x^2}\sin^2\left(\frac{1}{x}\right)dx=\frac{1}{4}\left(2\frac{B-A}{AB}+\sin\left(\frac{2}{B}\right)-\sin\left(\frac{2}{A}\right)\right)\Bigg|_A^B. \qquad (48)$$

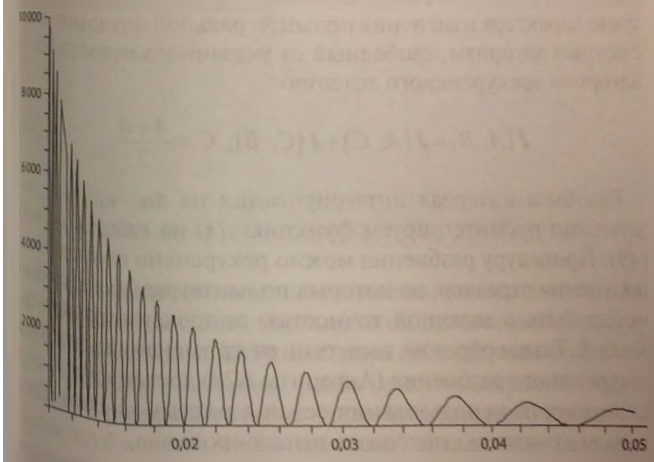


Рис. 87. График f(x)

Таблица 11

Результаты вычисления интеграла на разных отрезках

A	B	Npoints	eps real	time, c
0,00001	0,0001	1 553 568 289	-2,77E-11	434,55
0,0001	0,001	1 726 123 903	1,90E-10	470,99
0,001	0,01	360 075 831	2,05E-11	74,12

Окончание табл. 11

A	B	Npoints	eps real	time, c
0,01	0,1	79 973 845	-2,22E-12	16,44
0,1	1	105 108 653	8,67E-11	21,42
1	10	396 149	-6,00E-11	0,094
10	100	412 331	-6,30E-11	0,096

Использование для вычислений непосредственно метода трапеций неэффективно, поскольку предполагает равномерное сгущение сетки на всем отрезке интегрирования без учета характера изменения подынтегральной функции. Рассмотрим алгоритм, свободный от указанного недостатка – алгоритм «рекурсивного деления»:

$$J(A, B) = J(A, C) + J(C, B),\ C = \frac{A+B}{2}. \qquad (49)$$

Разобьем интервал интегрирования на две части и независимо проинтегрируем функцию *f(x)* на каждой из них (49). Процедуру разбиения можно рекурсивно повторять до получения отрезков, на которых подынтегральная функция может быть с заданной точностью аппроксимирована отрезком. Таким образом, выигрыш от применения алгоритма рекурсивного разбиения (Алгоритм А28) достигается за счет возможности использования сеток с разным числом узлов на разных участках интервала интегрирования.

Алгоритм А28

Последовательный алгоритм «рекурсивного деления»

```
main()
{
J= IntRecursive(A, B, f(A), f(B))
}

IntRecursive (A,B,fA,fB)
{
J=0
```

```
C=(A+B)/2

fC=f(C)

sAB=(fA+fB)*(B-A)/2

sAC=(fA+fC)*(C-A)/2
sCB=(fC+fB)*(B-C)/2

sACB=sAC+sCB

if(|sAB-sACB|≥ ε|sACB|)

J = IntRecursive (A,C,fA,fC)+ IntRecursive (C,B,fC,fB)

else

J = sACB

return J
}
```

При наличии достаточного количества процессоров и нулевого времени порождения параллельных процессов можно было бы непосредственно запускать пары подпрограмм *IntRecursive* для обработки половинок разбиваемого интервала. Но, поскольку в реальной системе число процессоров ограничено и время, необходимое для порождения параллельных процессов, существенно больше нуля, такой подход неэффективен. Запуск процессов, число которых значительно превышает число физических процессов системы, приводит к тому, что параллельная программа работает медленнее последовательной.

Следует разработать метод, позволяющий распределить вычислительную работу между ограниченным числом процессов. Несколько ветвей программы могли бы одновременно обрабатывать разные части интервала интегрирования, но координаты концов этих интервалов хранятся

в программном стеке процесса (они попадают туда в момент вызова процедуры *IniRecursDyn*) и фактически недоступны программисту. Если бы они были расположены в некотором явно описанном массиве, можно было бы передать их по частям для обработки разным нитям программы.

Метод локального стека

Рассмотрим соответствующий алгоритм — метод локального стека. Алгоритм использует массив данных, организованный по принципу стека — первым удаляется последний из добавленных элементов. Соответствующие процедуры приведены в листинге А30. В процедурах А30 для сокращения текста программ опущен код, обеспечивающий контроль переполнения и исчерпания стека.

Алгоритм А29

Метод локального стека

```
IntLocalStack(A,B)
{
J=0

fA=f(A)
fB=f(B)

sAB=(fA+fB)*(B-A)/2

while(1)
{
C=(A+B)/2

fC=f(C)

sAC=(fA+fC)*(C-A)/2
sCB=(fC+fB)*(B-C)/2
sACB=sAC+sCB

if(|sAB-sACB|≥ ε|sACB|)
{
```

```
PUT_INTO_STACK( A, C, fA, fC, sAC)
A=C
fA=fC
sAB=sCB
}
else
{
J+=sACB
if(STACK_IS_NOT_FREE)
break

GET_FROM_STACK( A, B, fA, fB, sAB)
}
}

return J
}
```

Алгоритм А30

Процедуры доступа к локальному стеку

```
// данные, описывающие стек

sp=0//указатель вершины стека
struct
  {
  A,B,fA,fB,s
  }
  stk[1000]  // массив структур в которых
// хранятся отложенные задания

// макроопределения доступа к данным стека

#define STACK_IS_NOT_FREE (sp>0)
#define PUT_INTO_STACK(A,B,fA,fB,s)
  {
  stk[sp].A=A  stk[sp].B=B
  stk[sp].fA=fA  stk[sp].fB=fB  stk[sp].s=s
sp++
}
#define GET_FROM_STACK(A,B,fA,fB,s)
```

```
  }
  sp--
  A=stk[sp].A  B=stk[sp].B
  fA=stk[sp].fA  fB=stk[sp].fB  s=stk[sp].s
}
```

Времена выполнения алгоритмов A28 и A29 отличаются примерно на 5% в пользу последнего, таким образом, алгоритм «локального стека» будет в дальнейшем использоваться в качестве наилучшего из имеющихся в нашем распоряжении последовательных методов. Именно относительно алгоритма A29 будет оцениваться эффективность создаваемых параллельных алгоритмов.

8.4.2. Параллельные алгоритмы

Относительно несложно разработать параллельные алгоритмы решения обсуждаемой задачи на основе метода геометрического параллелизма и метода коллективного решения, но они будут обеспечивать низкое ускорение.

Метод геометрического параллелизма

Согласно методу геометрического параллелизма, отрезок интегрирования разбивается на p частей, где p — число используемых процессоров. Все части отрезка распределяются по процессорам. Каждый процессор вычисляет частичную сумму на переданном ему отрезке, после чего все частичные суммы складываются (алгоритм A31).

Алгоритм A31

Параллельный алгоритм: метод геометрического параллелизма

```
for(i=0;i<p;i++)
StartParallelProcess(IntLocalStack, A+(B-A)*i/p,
                     A+(B-A)*(i+1)/p, &(s[i]))
 WaitAllParallelProcess
 J=0
 for(i=0;i<p;i++)
J+=s[i]
```

Функция *StartParallelProcess* в алгоритме A31 обеспечивает запуск процедуры *IntLocalStack* с параметрами, указанными в скобках. Ключевое слово *WaitAllParallelProcess* предполагает, что стоящие после него инструкции программы не будут выполняться до тех пор, пока все запущенные параллельные процессы не выполнятся полностью, и частичные суммы не будут записаны в соответствующие ячейки массива *s*. Дальнейшее вычисление интеграла сводится к нахождению суммы элементов массива *s* и выполняется последовательно одним процессом.

Алгоритм A31 эффективен при условии равномерного распределения всего объема вычислений по отрезку интегрирования. Однако существует множество функций, при интегрировании которых указанное условие не соблюдается. В их числе рассматриваемая нами на отрезке $[10^{-5}, 1]$ функция 47. При разбиении интервала $[10^{-5}, 1]$ на p равных частей, практически весь объем вычислений, необходимых для определения интеграла с точностью $\varepsilon = 10^{-5}$, сосредоточен в первой части:

$$\left[10^{-5}, 10^{-5} + \frac{1 - 10^{-5}}{p}\right].$$

Таблица 12

Параметры расчета интеграла на разных отрезках

p	Интервал 1	Интервал 2	Время 1, с	Время 2, с
10	[1e–5, 0,10000900000]	[0,10000900000, 1]	37,679	0,004
100	[1e–5, 0,01000990000]	[0,01000990000, 1]	37,274	0,037
1 000	[1e–5, 0,00100999000]	[0,00100999000, 1]	36,989	0,369
10 000	[1e–5, 0,00010999900]	[0,00010999900, 1]	34,064	3,364
100 000	[1e–5, 0,00001999990]	[0,00001999990, 1]	18,869	18,822

Из таблицы 12 следует полная непригодность метода геометрического параллелизма для решения поставленной задачи даже на системе из двух процессоров. При разбиении

на две *равные* части, на первую из них приходится практически вся вычислительная нагрузка. Для обеспечения равной вычислительной нагрузки на два процессора следует разбить интервал интегрирования на две *неравные* части, причем одна из них должна быть в 10^5 раз меньше другой.

Метод коллективного решения

Из таблиц 11, 12 следует также неприменимость и метода коллективного решения. Оценим его эффективность. Время выполнения последовательного алгоритма:

$$T_1 = \sum_{i=1}^{n} \tau_i, \tag{50}$$

где n — число частей, на которое разбили отрезок интегрирования, τ_i — время интегрирования части с номером i.

Время выполнения параллельного алгоритма:

$$T_p = \frac{1}{p} \sum_{i=1}^{n} (\tau_i + 2t_s), \tag{51}$$

где t_s — время передачи координат $[a, b]$ или время возврата результата вычислений s. Поскольку объем передаваемых данных и в первом, и во втором случае мал, правомерно считать времена их передачи одинаковыми и равными латентности.

Соотношение (51) предполагает, что время интегрирования на каждом из n интервалов одинаково: $\tau_i = \tau_0$. В этом случае эффективность составит:

$$E_p = \frac{n\tau_0}{p\frac{1}{p}n(\tau_0 + 2t_s)} = \frac{\tau_0}{\tau_0 + 2t_s} = \frac{1}{1 + \frac{2t_s}{\tau_0}}. \tag{52}$$

Согласно (52), эффективность близка к 100% при соблюдении условия $\tau_0 \gg 2t_s$ и при равенстве времен обработки разных фрагментов интервала интегрирования. Данные ва-

блица 10 свидетельствует об обратном: при равной длине фрагментов отрезка интегрирования требуется разное время для обработки каждого из них. В связи с этим соотношение (51) преобразуется следующем образом:

$$T_p = \max_p \sum_{i \in I_p} (\tau_i + 2t_s),$$

где I_p — множество номеров отрезков, обработанных процессом p.

Для того, чтобы все процессы были загружены равномерно, следует разбить исходный интервал на значительное число равных отрезков. Их количество на несколько порядков превысит общее число интервалов, необходимых при решении задачи с помощью последовательного алгоритма. Таким образом:

- параллельный алгоритм выполнит в целом большие операций, чем последовательный;
- соотношение $\tau_i \gg 2t_s$ не выполняется для большинства из этих отрезков, поскольку среди отрезков есть такие, на которых интеграл можно с достаточной точностью найти без измельчения сетки, что требует времени значительно меньше, чем необходимо для синхронизации (передачи данных).

В результате эффективность метода коллективного решения крайне низка. Его использование ведет к увеличению времени решения задачи. Таким образом, методы геометрического параллелизма и коллективного решения практически непригодны для решения поставленной задачи. Рассмотрим другой подход к решению поставленной задачи – метод глобального стека.

Метод глобального стека

Будем рассматривать системы с общей памятью. В качестве параллельных процессов будем использовать треды (threads) — легковесные процессы, нити. Для защиты разделяемых переменных будем использовать семафоры.

Метод глобального стека не требует наличия управляющего процесса. Процесс, выполняющий запуск расчета, никак не участвует ни в самой процедуре расчета, ни в принятии решения об окончании вычислений. В этом принципиальное отличие метода глобального стека от метода коллективного решения: порожденные процессы совершенно равноправны в своих возможностях, и каждый из них в равной степени выполняет вычислительные и управляющие функции. Укрупненная блок-схема алгоритма приведена на рис. 88.

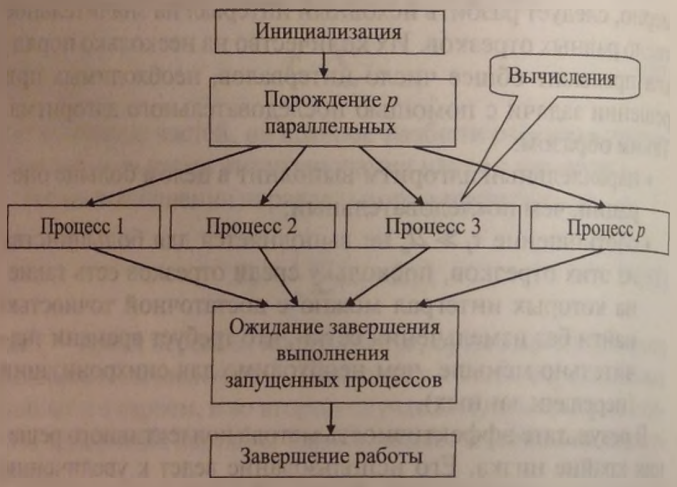


Рис. 88. Структура программы

Основные вычисления выполняются параллельными процессами: Процесс 1 ... Процесс *p*, каждый из которых выполняет один и тот же алгоритм.

Пусть в нашем распоряжении есть доступный всем параллельным процессам список отрезков интегрирования, организованный в виде стека. Назовем его глобальным стеком. Перед запуском параллельных процессов в глобальный стек помещается единственная запись — координаты концов от-

резка интегрирования, значения функции в концах отрезка и площадь соответствующей трапеции — в дальнейшем «отрезок». Предлагается следующая схема алгоритма, выполняемого каждым из параллельных процессов.

Пока в глобальном стеке есть отрезки:

1) взять один отрезок из глобального стека;
2) выполнить алгоритм локального стека (алгоритм A29), выполняя при записи в локальный стек следующие дополнительные действия:
   - если в локальном стеке есть несколько отрезков, а в глобальном стеке отрезки отсутствуют, то переместить часть отрезков из локального стека в глобальный стек;
3) по исчерпании локального стека добавить полученную частичную сумму к общему значению интеграла и вернуться к выполнению шага 1).

Данный несложный алгоритм вызывает ряд вопросов, например:

- должен ли процесс закончить работу, если в глобальном стеке отрезков нет?
- какую часть отрезков следует перемещать из локального стека в глобальный стек?
- как обеспечить корректную запись отрезков разными процессами в глобальный стек?
- в какой момент следует завершить работу параллельных процессов?

Легко увидеть, что отсутствие отрезков в глобальном стеке не является достаточным основанием для завершения работы процесса, обнаружившего этот факт. В самом деле, перед запуском интегрирующих процессов в глобальный стек помещается только один отрезок. В начале работы он будет взят из стека одним их процессов, например, Процессом1, после чего остальные процессы могут обнаружить, что

стек пуст. Могут, но не обязательно обнаружат, поскольку нет никаких весомых оснований утверждать, что какой-либо процесс, например, Процесс 2, обратится к глобальному стеку раньше, чем Процесс 1 разместит в нем некоторое количество новых отрезков. Обе ситуации возможны, поэтому следует предусмотреть адекватную реакцию, по возможности, не снижающую производительность алгоритма.

В запускающем процессе (алгоритм А32) после инициализации переменных инициируется выполнение *пр* интегрирующих процессов. Затем запускающий процесс переходит в состояние ожидания окончания работы запущенных процессов и в вычислениях участия не принимает.

Алгоритм А32

Запускающий процесс

```
// sdat. - глобальные переменные, разделяемые
//     всеми процессами slave_thr
//     и запускающим процессом main

main()
{
 // запускающая программа

 sdat.ntask=0 // число отрезков в глобальном стеке
              // интервалов
 sdat.nactive=0 // число процессов, обрабатывающих в
 // данный момент один из интервалов

 Sem_init(sdat.sem_sum, 1) // доступ к глобальной
                           // сумме открыт
 Sem_init(sdat.sem_list, 1) // доступ к глобальному
                            // стеку открыт
 Sem_init(sdat.sem_task_present, 0) // отрезков в
// глобальном стеке нет

 sdat.s_all=0 // Глобальная сума - переменная,
 // накапливающая частичные суммы

 PUT_INTO_GLOBAL_STACK(sdat.ntask)
```

```
(A,B,fun(A),fun(B),(fun(A)+fun(B))*(B-A)/2.)

sdat.ntask++ // в глобальный стек занесен первый
      // из отрезков

// откроем семафор, свидетельствующий о том,
// что в глобальном стеке интервалов есть отрезки
V(&sdat.sem_task_present)

// запуск пр параллельных процессов,
// выполняющих функции slave_thr

StartThrTask(p, slave_thr);
WaitThrTask();

// Параллельное выполнение функций slave_thr завершено

J=sdat.s_all; // значение интеграла
}
```

Уточним структуру интегрирующего процесса (алгоритм A33).

Алгоритм A33

Параллельный алгоритм: метод глобального стека (версия 1)

```
slave_thr()
{

// начало цикла обработки стека интервалов
while(sdat.ntask>0)
 {
 // чтение одного интервала из списка интервалов

 sdat.ntask-- // указатель глобального стека
 GET_FROM_GLOBAL_STACK(sdat.ntask)(a,b,fa,fb,sab)

 // начало цикла интегрирования одного интервала
 while(1)
   {
```

```
      c=(a+b)/2
      fc=fun(c)

      sac=(fa+fc)*(c-a)/2
      scb=(fc+fb)*(b-c)/2
      sacb=sac+scb

      if(!BreakCond(sacb,sab))
        {
        s+=sacb
        if(!sp) // локальный стек пуст
          break // выход из цикла интегрирования
// одного интервала
        sp--
        GET_FROM_LOCAL_STACK[sp]( a, b, fa, fb, sab)
        }
      else
        {
        PUT_INTO_LOCAL_STACK[sp]( a, c, fa, fc, sac)
        sp++
        a=c
        fa=fc
        sab=scb
        }

      // перемещение части локального стека
      // в общий список интервалов

      if((sp>SPK) && (!sdat.ntask))
        {
        while((sp>1) && (sdat.ntask<sdat.maxtask))
          {
          sp--
          GET_FROM_LOCAL_STACK[sp](a,b,fa,fb,sab)
          PUT_INTO_GLOBAL_STACK[sdat.ntask]
                  (a,b,fa,fb,sab)
          sdat.ntask++
          }
        }
      }
// конец цикла интегрирования одного интервала
```

```
  }
 // конец цикла обработки стека интервалов

// критическая секция вычисления общей суммы
P(sdat.sem_sum)
 sdat.s_all = sdat.s_all + s
 V(sdat.sem_sum)
}
```

Алгоритм мало отличается от метода локального стека (алгоритм А29). Отличия сосредоточены в его последней части:

- в конце цикла интегрирования одного интервала присутствует фрагмент переноса отрезков из локального стека в глобальный стек;
- по окончании цикла интегрирования выполняется суммирование частичных сумм, полученных всеми процессами.

Данный алгоритм неработоспособен, поскольку, несмотря на проверку в начале цикла обработки стека интервалов (*sdat.ntask>0*), может возникнуть ситуация, в которой одна и та же запись будет извлечена более чем одним процессом. Рассмотрим последовательность действий, иллюстрирующую возникновение этой ошибки:

№ шага	Состояние переменных	Процесс 1	Процесс 2
0	sdat.ntask=1	while(sdat.ntask>0)	while(sdat.ntask>0)
2		sdat.ntask--	
3	sdat.ntask=0		sdat.ntask--
4	sdat.ntask=-1	GET_OF_GLOBAL_STACK(sdat.ntask) – чтение записи с номером -1	GET_OF_GLOBAL_STACK(sdat.ntask) – чтение записи с номером -1

Два процесса успешно выполнили проверку наличия в глобальном стеке записей, затем первый процесс уменьшил указатель стека, ранее указывающий на первую свободную ячейку. После шага 2 указатель стека указывает ячейку, хранящую отрезок, что пока еще правильно. На шаге 3 второй процесс выполнил такое же действие. В результате указатель стека указывает на ячейку –1. Возникают сразу две проблемы: во-первых, на шаге 4 оба процесса прочтут одну и ту же запись; во-вторых, такой записи нет — она за пределами стека. Попытка чтения несуществующей записи приведет, в лучшем случае, к аварийной остановке программы, в худшем — к непредсказуемым неверным результатам.

Ошибка вызвана отсутствием мер, предотвращающих одновременную модификацию разделяемых переменных несколькими процессами.

Ограничение доступа к глобальному стеку отрезков интегрирования (алгоритм А34) с помощью критической секции приведет к ряду негативных последствий.

Алгоритм А34

Критическая секция накопления глобальной суммы

```
main()
.
Sem_init(sdat.sem_list, 1) //доступ к глобальному стеку
                           // открыт

slave_thr()
{
.while(1)
  {
// Начало критической секции чтения из глобального
// стека очередного интервала интегрирования
  //
  P(sdat.sem_list)

  if(sdat.ntask≤0)
{
```

```
    V(sdat.sem_list) // разрешить другим процессам
        // доступ к глобальному стеку
    break
    }

  sdat.ntask-- // указатель глобального стека
  GET_FROM_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)

  V(sdat.sem_list)
  //
  // Конец критической секции чтения из глобального
  // стека очередного интервала интегрирования

  }
```

При выполнении алгоритма А34 возможна следующая последовательность действий:

№ шага	состояние переменных	Процесс 1	Процесс 2
0	sdat.ntask=1 sdat.sem_list=1	P(sdat.sem_list)	
2	sdat.ntask=1 sdat.sem_list=0	sdat.ntask--	
3	sdat.ntask=0 sdat.sem_list=0	GET_OF_GLOBAL_STACK[0]	
4	sdat.ntask=0 sdat.sem_list=1	V(sdat.sem_list)	
5	sdat.ntask=0 sdat.sem_list=1		P(sdat.sem_list)
6	sdat.ntask=0 sdat.sem_list=0		if(sdat.ntask)
7	sdat.ntask=0 sdat.sem_list=1		процесс завершил работу

В результате, Процесс 2 заканчивает работу раньше, чем Процесс 1 полностью вычислит интеграл на заданном отрезке интегрирования. Таким образом, может оказаться, что Процесс 1 после завершения Процесса 2 породит некоторое количество отрезков и запишет их в глобальный стек. Процесс 2 мог бы принять участие в их обработке, но не примет, поскольку уже завершится к этому моменту. Результат выполнения программы будет правильным, так как Процесс 1 обработает все отрезки, но время выполнения программы будет больше, чем было бы при совместной работе двух процессов. Условие выхода из цикла обработки стека интервалов выбрано неудачно. Не следует прекращать работу интегрирующих процессов до тех пор, пока все отрезки интервала интегрирования не будут полностью обработаны.

Отрезок интегрирования может находиться в нескольких состояниях:

- храниться в глобальном стеке интервалов;
- обрабатываться некоторым интегрирующим процессом;
- находиться в локальном стеке интервалов некоторого процесса;
- быть полностью обработанным: значение интеграла на этом отрезке вычислено и прибавлено к локальной частичной сумме соответствующего процесса.

«Время жизни» отрезка после того, как некоторый процесс начал его обработку, относительно невелико — отрезок разбивается на две части и перестает существовать, породив два новых отрезка. Таким образом, требование *«все отрезки интервала интегрирования полностью обработаны»* означает, что:

- функция проинтегрирована на всех отрезках, покрывающих исходный интервал интегрирования;
- полученные на отрезках интегрирования значения интегралов добавлены к частичным суммам соответствующих процессов.

Таким образом, потребуются следующие основные общие данные:

	семафоры доступа:
sdat.sem_list	семафор доступа к глобальному стеку отрезков
sdat.s_all	семафор доступа к значению интеграла
	семафоры состояния:
sdat.sem_task_present	семафор наличия записей в глобальном стеке отрезков
	переменные:
sdat.list_of_tasks	глобальный стек отрезков
sdat.ntask	число записей в глобальном стеке отрезков — указатель глобального стека отрезков
sdat.nactive	число активных процессов
sdat.s_all	значение интеграла

Окончательная версия алгоритма, свободная от рассмотренных выше недостатков, приведена в листинге А35. Процедуры доступа к глобальному стеку аналогичны процедурам листинга А30.

Окончательный алгоритм.

Алгоритм А35

Параллельный алгоритм: метод глобального стека

```
int slave_thr()
{
 // все переменные, начинающиеся с sdat.
 // являются глобальными,
 // к ним имеет доступ каждый из запущенных процессов
```

```
// slave_thr
// и запускающая программа main

// sp, s - локальные переменные процессов slave_thr

sp=0 // указатель локального стека - локальный
     // стек пуст

s=0 // частичная сумма интегралов, вычисленных
    // на отрезках,
// обработанных данной копией процесса

// начало цикла обработки стека интервалов
while(1)
 {
 // ожидание появления в глобальном стеке интервалов
 // для обработки

 P(sdat.sem_task_present)

// чтение одного интервала из списка интервалов

// Начало критической секции чтения из глобального
// стека очередного интервала интегрирования
 //
 P(sdat.sem_list)

 sdat.ntask-- // указатель глобального стека
 GET_OF_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)

 if(sdat.ntask)
   V(&sdat.sem_task_present)

 if(a<=b) // очередной отрезок не является
          // терминальным
   sdat.nactive++ // увеличить число процессов,
                  // имеющих интервал для интегрирования

 V(sdat.sem_list)
 //
// Конец критической секции чтения из глобального
```

```
// стека очередного интервала интегрирования

  if(a>b) // отрезок является терминальным
    break // выйти из цикла обработки стека интервалов

// начало цикла интегрирования одного интервала
  while(1)
    {
    c=(a+b)/2
    fc=fun(c)

    sac=(fa+fc)*(c-a)/2
    scb=(fc+fb)*(b-c)/2
    sacb=sac+scb

    if(!BreakCond(sacb,sab))
      {
      s+=sacb
      if(!sp) // локальный стек пуст
       break // выход из цикла интегрирования
 // одного интервала
      sp--
      GET_FROM_LOCAL_STACK[sp]( a, b, fa, fb, sab)
      }
    else
      {
      PUT_TO_LOCAL_STACK[sp]( a, c, fa, fc, sac)
      sp++
      a=c
      fa=fc
      sab=scb
      }

    // перемещение части локального стека
    // в общий список интервалов

    if((sp>SPK) && (!sdat.ntask))
      {
      // Начало критической секции заполнения
      // глобального стека отрезками интегрирования
      //
```

```
        P(sdat.sem_list)

        if(!sdat.ntask)
          {
  // установить семафор наличия
  // записей в глобальном стеке

          V(sdat.sem_task_present)
          }

        while((sp>1) && (sdat.ntask<sdat.maxtask))
          {
          sp--
        GET_FROM_LOCAL_STACK[sp](a,b,fa,fb,sab)
      PUT_TO_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab)
          sdat.ntask++
          }

        V(sdat.sem_list)
 //
 // Конец критической секции заполнения глобального
 // стека отрезками интегрирования
       }
     }
 // конец цикла интегрирования одного интервала

 // Начало критической секции заполнения глобального
 // стека терминальными отрезками (a>b)
   //
   P(&sdat.sem_list)
   sdat.nactive--

if( (!sdat.nactive) && (!sdat.ntask) )
    {
    // запись в глобальный стек списка
    // терминальных отрезков
    for(i=0;i<nproc;i++)
      {
      PUT_TO_GLOBAL_STACK[sdat.ntask](2,1,-,-,-)
      sdat.ntask++;
      }
```

```
// в глобальном стеке есть записи
    V(sdat.sem_task_present)
    }

  V(sdat.sem_list)
  //
  // Конец критической секции заполнения глобального
// стека терминальными отрезками
  }
  // конец цикла обработки стека интервалов

 // Начало критической секции сложения частичных сумм
 //
 P(&(sdat.sem_sum))
 sdat.s_all+=s
 V(&(sdat.sem_sum))
 //
 // Конец критической секции сложения частичных сумм
}
```

Инициализация глобальных переменных выполняется в запускающей программе A32 перед запуском интегрирующих процессов ***slave_thr***. В начале работы процессы ***slave_thr*** инициализируют свои локальные стеки, устанавливая в них нулевое количество записей, они также устанавливают нулевые значения локальных сумм.

В дальнейшем основная работа выполняется внутри цикла обработки стека интервалов. В листинге A35 жирным шрифтом выделены строки, описывающие критические интервалы и доступ к ним. Каждый из процессов ожидает появления записей в глобальном стеке:

```
P(sdat.sem_task_present)
```

Использование данного семафора предотвращает доступ к заведомо пустому стеку, однако не отменяет необходимости использования семафора доступа ***sdat.sem_list***, гаран-

тирующего корректную модификацию глобальных переменных несколькими процессами. Таким образом, чтение записи выполняется в критической секции, ограниченной строками:

```
P(sdat.sem_list)
V(sdat.sem_list)
```

Внутри критической секции выполняется чтение и изъятие из глобального стека одного из отрезков. Далее открывается семафор наличия в глобальном стеке отрезков, если они есть:

```
if(sdat.ntask)
V(&sdat.sem_task_present)
```

Затем выполняется анализ типа отрезка. Если правый конец меньше или равен левому, то отрезок интерпретируется как сигнал окончания работы («терминальный» отрезок). В противном случае увеличивается счетчик активных процессов — процессов, получивших интервал интегрирования:

```
if(a<=b)
  sdat.nactive++
```

После выхода из критической секции осуществляется либо обработка очередного интервала в цикле интегрирования одного интервала, либо выход из цикла обработки стека интервалов, если получен терминальный отрезок.

Внутри цикла интегрирования одного интервала осуществляется контроль числа отрезков, размещенных в локальном стеке процесса. Если оно превышает наперед заданную величину *SPK*, и в глобальном стеке нет отрезков, то осуществляется перенос части отрезков из локального стека в глобальный. Перенос осуществляется в пределах крити-

ческой секции. Следует обратить внимание на то, что проверка

```
if((sp>SPK) && (!sdat.ntask))
```

выполняется вне критической секции, следовательно, возможна ситуация, в которой в момент доступа процесса внутрь критического интервала переменная *sdat.ntask* уже не будет иметь нулевое значение — то есть в глобальном стеке другим процессом уже будут размещены некоторые отрезки. Именно поэтому семафор наличия в стеке отрезков устанавливается только в том случае, если стек действительно пуст:

```
if(!sdat.ntask)
  V(sdat.sem_task_present)
```

По окончании цикла обработки очередного отрезка выполняется проверка наличия заданий в глобальном стеке отрезков и наличия активных процессов:

```
if( (!sdat.nactive) && (!sdat.ntask) )
```

Эти действия выполняются внутри критической секции, таким образом, если обе проверки дали отрицательный результат, можно быть уверенным в том, что все фрагменты заданного интервала интегрирования уже обработаны, можно переходить к суммированию частичных сумм. Однако только один процесс «владеет» информацией о том, что интегрирование фрагментов полностью завершено, следует сообщить об этом остальным процессам, напомним, что они находятся в состоянии ожидания открытия «семафора наличия отрезков в глобальном стеке». Таким образом, единственным легальным способом завершить их работу, является размещение в глобальном стеке терминальных отрезков и открытие соответствующего семафора:

```
for(i=0;i<nproc;i++)
{
PUT_TO_GLOBAL_STACK[sdat.ntask](2,1,-,-,-)
sdat.ntask++;
}

V(sdat.sem_task_present)
```

Процесс, получивший терминальный отрезок, добавляет внутри критической секции найденную им частичную сумму к общему значению интеграла и заканчивает свою работу.

```
P(&(sdat.sem_sum))
sdat.s_all+=s
V(&(sdat.sem_sum))
```

Рассмотренный алгоритм демонстрирует достаточно высокую эффективность при выполнении на 2, 3 и 4-х процессорах, что подтверждают приведенные в таблице 13 результаты определения интеграла $J = \int_{10^{-5}}^{1} \frac{1}{x^2} \sin^2\left(\frac{1}{x}\right)$ с точностью $\varepsilon = 10^{-5}$.

Таблица 13

Результаты тестирования программы вычисления определенного интеграла

N_p	1	2	3	4
Время выполнения	31,39	15,61	10,29	7,83
Ускорение	1	2,01	3,05	4,01
Эффективность	100%	101%	102%	100%

В листинге A32 запуск параллельных процессов и ожидание окончания их работы упрощенно описаны строками:

```
StartThrTask(np, slave_thr);
WaitThrTask();
```

В листинге А36 приведен подробный пример, иллюстрирующий последовательность запуска параллельных процессов, каждый из которых выполняет процедуру, указанную параметром tsub. В примере А32 запускалась процедура slave_thr.

Алгоритм А36

Процедуры запуска и завершения параллельных нитей

```
typedef int ThrSub(void);

pthread_t *tid_procs;

StartThrTask(int np, ThrSub *tsub)
{
int i;

// --- распределить массив дескрипторов запускаемых
// процессов

tid_procs = (pthread_t *)malloc(np*sizeof(pthread_t));

for (i = 0; i < np; i++) // -- запустить np процессов --
pthread_create(&(tid_procs[i]), NULL, tsub, NULL);

#ifdef HAVE_THR_SETCONCURRENCY_PROTO
thr_setconcurrency(np);
#endif
}

WaitThrTask(void)
{
int i;
for(i = 0; i < np; i++)
pthread_join(tid_procs[i], NULL);
}
```

Глава 9

Визуализация сеточных данных

С ростом числа процессоров, используемых для проведения вычислительных экспериментов, сложность задачи преобразования больших объемов обрабатываемых ими сеточных данных к виду, пригодному для их изучения, в том числе для их наглядного отображения, качественно возрастает. Будем говорить, что данные имеют большой объем, если для их обработки за желаемое время недостаточно ресурсов последовательных вычислительных систем.

При интерактивной визуализации «желаемое время» измеряется секундами и долями секунды. За это время требуется передать данные системе визуализации, преобразовать их к виду, удобному для визуального восприятия, и отобразить на экране. Если все эти операции можно выполнить с помощью одного из стандартных пакетов визуализации научных данных, таких как Origin или Tecplot, то необходимости привлечения для обработки многопроцессорной системы нет. Однако, если число узлов расчетной сетки превышает некоторый порог, для визуализации уже недостаточно ресурсов компьютеров рабочих мест пользователей. Характерная производительность компьютера, установленного на рабочем месте пользователя, — десятки Gflops, а производительность доступных для выполнения расчетов суперкомпьютеров — десятки и сотни Tflops. Даже без учета суперкомпьютеров, расположенных на первых позициях списка top500, разница в производительности составляет десятки тысяч раз. Требуемая для визуализации результатов вычислительная мощность обычно меньше, чем мощность, необходимая для проведения моделирования из-

учаемых процессов, тем не менее, алгоритмы визуализации олжны быть ориентированы именно на параллельную обработку данных. Недостаточная вычислительная мощность персонального компьютера — причина, по которой персонального компьютера недостаточно для полноценного изучения результатов проводимых на суперкомпьютерах вычислительных экспериментов. Укажем еще ряд причин, по которым для анализа больших объемов данных необходимо привлекать параллельные вычислительные системы и специальные, ориентированные на удаленную визуализацию решения:

- 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.

Характерные значения параметров каналов связи, объемов оперативной памяти и дисковой файловой системы рабочего места на порядки меньше требуемых, поэтому сценарий, согласно которому расчет выполняется на суперкомпьютере, а обработка и визуализации результатов — на компьютере рабочего места пользователя, на практике не реализуем. Можно сделать основной вывод: существенная часть операций, обеспечивающих визуализацию больших объемов данных должна выполняться на параллельных вычислительных мощностях многопроцессорного сервера визуализации.

9.1. Клиент-серверная технология

Наиболее естественным решением проблемы является использование технологии клиент-сервер (рис. 89), в соответствии с которой:

- серверная часть системы визуализации, выполняемая на многопроцессорной системе (будем называть ее *сервером визуализации*), обеспечивает обработку большого объема данных;
- клиентская часть системы визуализации (будем называть ее *клиентом визуализации*) выполняется на компьютере рабочего места пользователя. Она обеспечивает интерфейс взаимодействия с пользователем и выполняет непосредственное отображение данных, предварительно подготовленных сервером.

Рис. 89. Структура системы визуализации

На клиентской части системы визуализации возможно использование аппаратных возможностей персонального компьютера как для построения наглядных визуальных образов (с помощью графических ускорителей, стерео устройств), так и для удобного управления виртуальной сценой с помощью многомерных манипулято-

ров. На серверной части системы визуализации возможно привлечение достаточного числа вычислительных узлов, обеспечивающих в совокупности необходимую производительность, объем оперативной памяти, емкость и пропускную способность файлового хранилища.

9.2. Online или Offline-визуализация: плюсы и минусы

Визуализация результатов вычислительных экспериментов подразумевает взаимодействие как минимум двух компонент:

- модуля численного моделирования;
- модуля визуализации.

В связи с этим возникает первый принципиальный вопрос: как именно им следует взаимодействовать? Должны ли они быть объединены в одну программу, или это должны быть две разные программы. Следует ли выполнять визуализацию данных непосредственно при проведении расчета, используя данные, расположенные в оперативной памяти (online-визуализация), или следует периодически записывать эти данные на диск и только потом выполнять их визуализацию с помощью отдельной программы (offline-визуализация)?

9.2.1. Online-визуализация

Отметим два плюса первого подхода:

- экономия дискового пространства и экономия времени на чтение и запись результатов;
- потенциальная возможность активного влияния на ход выполнения вычислений.

Наблюдение за динамикой протекания моделируемого процесса непосредственно во время расчета позволяет корректировать параметры расчета, либо совсем остановить

его, если текущие результаты перестают отвечать ожиданиям. Однако, поддержка режима интерактивного управления расчетом далеко выходит за рамки задачи визуализации и требует отдельного обсуждения. Что касается возможности оценки промежуточных результатов и возможности при необходимости остановить расчет, то и в рамках второго подхода (offline-визуализации) и то, и другое вполне реализуемо. Периодическая запись результатов на диск позволяет, не дожидаясь окончания всего расчета, выполнять изучение промежуточных результатов. Запись признака досрочного завершения в файл, периодически опрашиваемый расчетной программой, тривиально решает проблему простейшей обратной связи.

Отметив плюсы, перечислим характерные проблемы, связанные с online-визуализацией.

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

б. Затруднены возможности повторного изучения единожды визуализированных данных. Поскольку основное преимущество интеграции модулей визуализации в расчетную программу — исключение этапа записи данных на диск, все данные должны храниться в оперативной памяти узлов. Расчетная программа осуществляет обработку массивов данных, доступ к которым могут иметь и модули визуализации. Предположим, что это именно так, и отдельной копии данных для визуализации не создается, что позволяет экономить не только дисковую память, но и оперативную. Как правило, пользователь сначала знакомится с общим видом объекта, а уже потом, выявив потенциально интересные зоны, переходит к изучению соответствующих деталей. Коллизия заключается в том, что между моментом визуализации общего вида и моментом более детализированной визуализа-

ции фрагмента объекта, расчет продолжается. Приостановка расчета на время изучения визуального образа сопряжена с простоем большой вычислительной мощности, что вряд ли будет одобрено руководством вычислительного комплекса. Поскольку расчет продолжается, состояние моделируемого объекта меняется. Ориентируясь на состояние объекта в некоторый момент модельного времени, Вы заказали детализированный просмотр части объекта. Система визуализации отобразит запрошенный фрагмент, но его состояние будет соответствовать уже *другому* моменту модельного времени. Вполне возможно, что место, в котором сконцентрированы наиболее интересные детали изучаемого процесса сместилось за пределы запрошенной области. Это принципиально ограничивает возможности детального изучения объекта и затрудняет его адекватное восприятие.

- Ограничены возможности визуализации динамики процесса. Малый шаг модельного времени и большое время расчета каждого шага не позволит, без запоминания промежуточных результатов, просмотреть в ускоренном темпе состояние объекта на множестве ряда отстоящих друг от друга моментов. Таким образом, для запоминания промежуточных результатов потребуется дополнительная оперативная память, объем которой не безграничен.
- Ограничены возможности визуализации заданных моментов времени, поскольку темп изучения пользователем визуальных образов не согласован с темпом проведения расчета.
- Неконтролируемо ухудшается балансировка загрузки процессоров. За счет того, что системой визуализации будет затребована дополнительная оперативная память, возможно, замедлится основной расчет, причем не на проценты, а на порядки. Если оперативной памя-

ти окажется недостаточно, то данные будут вытеснены на жесткий диск, что как правило приводит к колоссальному замедлению. 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. Фактически в одну задачу объединяются не просто две разные задачи, но две задачи, требующие разработки алгоритмов разных классов (первого и второго из рассмотренных в разделе 1.2), к которым предъявляются взаимно противоположные требования. Оптимизация согласованного использования системы этими двумя ядрами (вычислительным и визуализации) — существенно более сложная задача, чем раздельная оптимизация под каждую из этих задач. Для функционирования этих двух ядер требуется разное число процессоров. Однако, выделение под визуализацию отдельного набора процессоров потребует передачи всего объема данных от процессоров вычислительного ядра к процессорам визуализации, что, по сути, будет мало отличаться от обсуждаемой далее offline-визуализации, а по реализации будет существенно сложнее.

е. Возникают организационные трудности, обусловленные режимом коллективного доступа к мощностям суперкомпьютерных центров. На суперкомпьютерах центров коллективного пользования поддерживаются очереди заданий. Вы не *запускаете* задачу — вы *размещаете заявку* на запуск задачи — ставите задачу в очередь. Крупные вычислительные системы работают круглосуточно, и вполне возможно, что Ваша задача будет выполняться тогда, когда Вас не будет на рабочем месте, и воспользоваться результатами визуализации не удастся.

Offline-визуализация

Рассмотрим второй подход (его представляют такие пакеты визуализации, как ParaView, Visit, EnSight, OpenDX, RemoteViewer), предполагающий раздельную реализацию модулей численного моделирования и визуализации — выделение их в отдельные программы, взаимодействующие через файловую систему. В его рамках видится один существенный минус: необходимость периодической записи данных на диск промежуточных результатов. Это требует повышенных ресурсов дисковой памяти и приводит к дополнительным потерям времени. Запись может занимать существенное время, но, как будет показано, при записи с целью дальнейшей визуализации (а не с целью продолжения расчета с некоторого промежуточного состояния), объемы записываемых данных могут быть радикально сокращены, что сократит и время их ввода-вывода. Таким образом, в дальнейшем, описывая методы визуализации мы будем ориентироваться на подход, предполагающий offline визуализацию.

9.3. Этапы визуализации

Будем исходить из того, что исходные данные для визуализации представлены множеством наборов сеточных данных. Число таких наборов определяется числом моментов модельного времени, для которых выполнено сохранение промежуточных данных. Каждый набор сеточных данных представлен расчетной сеткой и сеточными функциями. Расчетная сетка задается набором вершин (их координатами) и элементами сетки, такими как ребра, грани, пирамиды, призмы и другие. В частном случае, при выполнении расчета на неадаптивной сетке, необходимость хранения отдельного экземпляра сетки для каждого момента модельного времени отсутствует, — достаточно хранить одну копию сетки для всех наборов сеточных функций, что дополнительно сокращает объем хранимых данных.

Скалярные сеточные функции представляются массивами, в каждом элементе которых хранятся значения изучаемых величин, таких как давление, температура или плотность, соответствующие узлам или другим элементам сетки. Векторные или иные величины могут быть представлены наборами скалярных величин.

Уточним задачу визуализации, определив операции, выполняемые расчетной программой (на этапе моделирования), сервером визуализации и клиентом визуализации (на этапе визуализации). Рассмотрим наиболее затратные с точки зрения времени выполнения этапы подробнее.

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

Декомпозиция сетки (пункты а и е) в рассматриваемом контексте необходима для эффективной организации параллельной обработки данных на нескольких процессорах сервера визуализации. Подробно задача декомпозиции рассмотрена в главе «Декомпозиция сеточных графов», в разделе 7.7. «7.7. Декомпозиция больших сеток».

Ввод-вывод фрагментов сетки и сеточных функций (пункты в, г, ж, з). К способу хранения самой сетки и заданных на ней сеточных функций предъявляются различные требования. Конечно-разностная или конечно-элементная сетка опи-

сывается информацией двух видов: геометрической и топологической. Геометрическая информация (такая как координаты узлов сетки) может рассматриваться как частный случай сеточной функции, но топологическая информация имеет совсем другую природу. Фактически это множество целочисленных нерегулярных данных, описывающих связи между элементами сетки. Поскольку при визуальном изучении результатов может потребоваться детальная визуализация любого фрагмента сетки, необходимо хранить сетку без каких-либо потерь топологической информации. Как будет показано при оценке вариантов компрессии данных, 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. Подробнее метод записи с частичной потерей точности скалярных сеточных функций рассматривается в разделе 9.5.3. «9.5.3. Огрубление и сжатие скалярных сеточных функций».

Дополнительный способ значительного сокращения времени декомпозиции, записи и чтения данных основан на их распределенном иерархическом хранении, рассматриваемом в разделе 9.5.1. «9.5.1. Соотношение времени чтения данных и времени их обработки».

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

На этом этапе решаются три взаимно противоречивые 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.

Сложность представляет именно совместное решение этих трёх задач. По отдельности каждая из них решается тривиально. Основным методом, позволяющим решить за заданное время все три задачи, является огрубление изучаемого объекта с контролируемой точностью — соответствующий пример рассматривается в разделе 9.4. «9.4. Визуализация изоповерхностей».

Визуализация трехмерных объектов доступными сегодня компьютерными методами всегда связана с потерями в точности их представления. Например, информация искажается при отображении трехмерных результатов на двумерные устройства визуализации. Отображение большого числа примитивов на дисплеях, имеющих ограниченное разрешение (число пикселей экрана), также приводит к большим потерям информации, поскольку в один пиксель экрана может отображаться множество узлов расчетной сетки.

Следовательно, целесообразно частично огрубить объект на параллельном сервере визуализации, до того как данные будут переданы на клиентскую часть системы визуализации и выведены на экран. Отметим, графические ускорители поддерживают возможность непосредственного отображения лишь ограниченного числа трехмерных

примитивов, что также говорит в пользу предварительного огрубления объекта — сокращения числа описывающих его примитивов.

Таким образом, основная задача этого пункта заключается в таком сокращении числа описывающих объект примитивов (узлов, ребер, граней и т. п.), при котором формируемый на их основе образ визуально близок исходному неогрубленному объекту. Подчеркнем, что требуется не сократить число описывающих объект примитивов в *некоторое число раз*, а сократить общий объем данных, описывающих огрубленный объект *за заданное время*, до заданной величины. Очевидно, что такая постановка задачи смещает акцент с точности описания объекта на время его обработки, что не является критичным, поскольку при интерактивной визуализации сохраняется возможность детального изучения частей объекта с меньшей потерей точности.

Операции пунктов (л, м) не являются затратными с точки зрения времени выполнения, а возможности по сокращению времени выполнения пунктов (б, д) обсуждаются в конце настоящей главы. Следует отметить, что возможность частичного манипулирования трехмерным объектом с использованием средств отображения виртуальной реальности существенно повышает удобство работы и уровень восприятия изучаемого объекта. Такие простые операции, как плавное вращение объекта, частичное приближение и увеличение его фрагментов *без* обращения к серверу визуализации, повышают управляемость объектом и создают ощущение погружения в виртуальную сцену, что является одним из основных преимуществ системы визуализации.

На этом краткий обзор этапов визуализации сеточных данных большого объема завершен. Далее, на примере визуализации изоповерхностей, рассматривается возможная реализация изложенных принципов.

9.4. Визуализация изоповерхностей

Одним из наглядных методов визуализации трехмерных скалярных данных является визуализация изоповерхностей.

Изоповерхность — это множество точек пространства, в которых скалярная функция принимает заданное значение.

В приближении сеточного подхода изоповерхность описывается набором треугольников, опирающихся вершинами на множество точек трехмерного пространства. Современные видеоускорители аппаратно поддерживают отображение массивов треугольников, что значительно повышает наглядность визуализации как за счет высокой скорости вывода данных на экран, так и за счет возможности автоматического формирования стереоизображений. Однако, число описывающих изоповерхность треугольников может многократно превышать число треугольников, непосредственно отображаемых графическим ускорителем. В связи с этим ядро системы визуализации представлено рядом алгоритмов фильтрации и огрубления первичных данных (изоповерхностей). Эти алгоритмы обеспечивают аппроксимацию исходной триангуляции новой триангуляцией, описываемой ограниченным объемом данных, достаточным, однако, для восстановления изучаемого образа с заданным уровнем качества (рис. 90).

За короткое время необходимо огрубить данные и передать результат огрубления на компьютер пользователя. Фактически, к алгоритму огрубления предъявляются два требования, проистекающие из желания обеспечить интерактивный режим работы (высокую скорость предоставления пользователю визуального образа). Во-первых, алгоритм огрубления должен работать быстро. Во-вторых, необходимо выполнить огрубление данных *до заданного объема*, диктуемого временем, отведенным на передачу данных на компьютер пользователя.

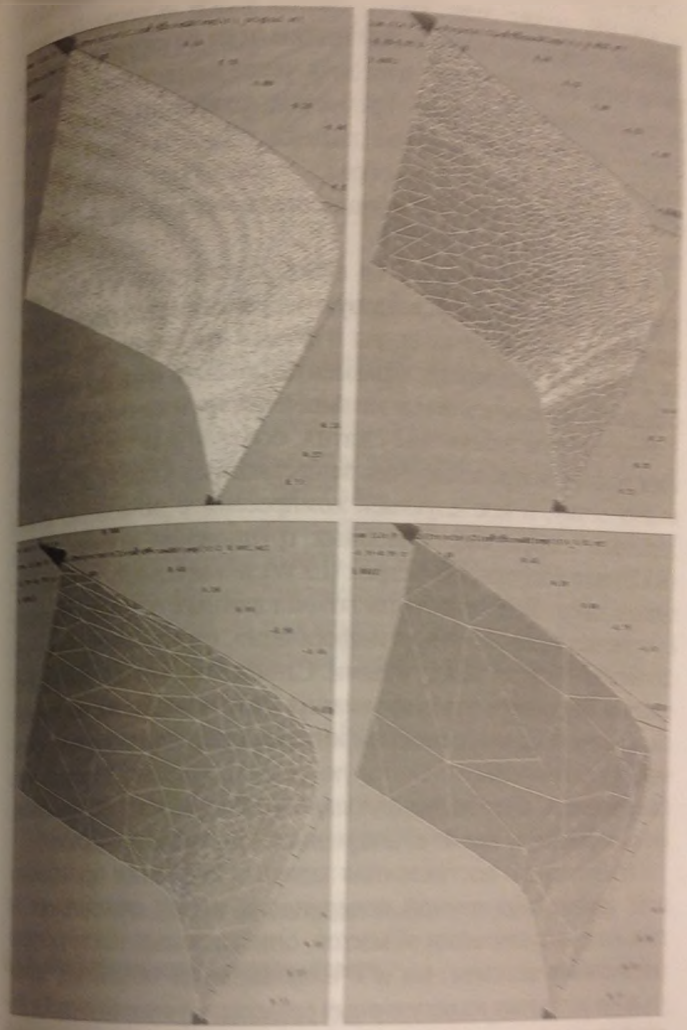


Рис. 90. Пример огрубления поверхности

Число описывающих изоповерхность узлов по порядку величины может совпадать с числом узлов исходной трехмерной сетки, поэтому и необходим этап ее огрубления до размеров, допускающих передачу через медленные каналы связи за короткое время. 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 (рис. 90).

При изучении объекта «в целом» деталями можно пожертвовать, поскольку, в первую очередь визуально воспринимаются основные контуры и формы трехмерного объекта. При необходимости, фрагменты объекта можно изучить с большим увеличением и меньшей потерей точности.

Простейшие алгоритмы сжатия, основанные на уменьшении точности представления вещественных чисел, описывающих сетку, и последующей компрессии с помощью стандартных алгоритмов, подобных групповому кодированию (RLE) или кодированию строк (LZW) значительного выигрыша не дают. Большую часть объема данных о триангуляции составляет целочисленная информация, описывающая ее топологию — связи между узлами. Стандартными алгоритмами сжатия без потерь эта информация практически не сжимается. Таким образом, основной интерес представляют алгоритмы, формирующие некоторую новую триангулированную поверхность, аппроксимирующую исходную изоповерхность, но содержащую значительно меньшее количество узлов.

Отметим, что поставленная 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. Поясним этот тезис на характерном примере сечения куба изоповерхностью показан на рисунке 91. В приведенном примере это се-

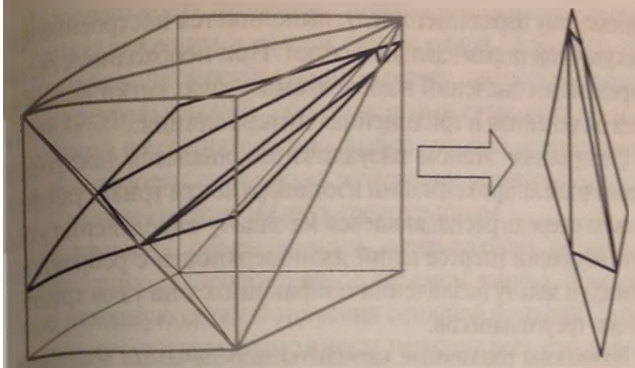


Рис. 91. Сечение регулярной 3D-сетки плоскостью

чение является четырехугольником. Поскольку мы условились описывать изоповерхность триангуляцией — множеством треугольников, следует разделить этот четырехугольник на треугольники. Даже в приведенном на рисунке 92 простейшем случае можно предложить два различных способа разбиения на треугольники. Для уменьшения подобного произвола куб предварительно разбивается на пирамиды и уже по точкам пересечения с ребрами пирамид проводят сечение.

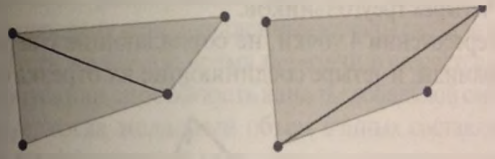


Рис. 92. Неоднозначность триангуляции четырехугольника

9.4.1. Аппроксимация изоповерхности

В дальнейшем предполагается, что выполняется обработка скалярной функции, определенной в узлах тетраэдральной сетки. В случае необходимости визуализации данных, заданных на трехмерной решетке, топологически эквивалентной

индексному параллелепипеду, выполняется построение промежуточной пирамидальной сетки. При необходимости, для определения значений в добавляемых узлах сетки используется билинейная и трилинейная интерполяция.

Следующим этапом визуализации является построение первичной аппроксимации изоповерхности триангуляцией, в свою очередь распадающееся на задачу определения узлов триангуляции (пересечений изоповерхности с ребрами пирамид) и задачу назначения опирающихся на узлы триангуляции треугольников.

Возможны различные варианты пересечения изоповерхности с ребрами пирамиды. В большинстве из них сечением пирамиды является точка, ребро или один треугольник. Рассмотрим два случая, порождающих большее число треугольников.

1. В пересечении 6 ребер — пирамида полностью принадлежит изоповерхности. Этот случай соответствует положению пирамиды в трехмерной области, в которой функция принимает одно и то же значение, совпадающее с уровнем проводимой изоповерхности. Необходимо добавить в изоповерхность все четыре треугольника, образующие грани пирамиды. Сечение состоит из четырех треугольников.
2. В пересечении 4 точки, не совпадающие с вершинами пирамиды, и четыре соединяющие их отрезка (рис. 93).

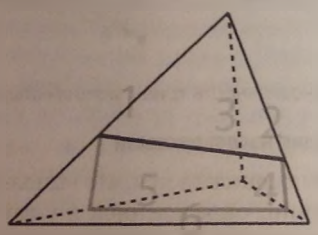


Рис. 93. Изоповерхность проходит через четыре точки

Изоповерхность пересекает пирамиду по четырем точкам, не принадлежащим в общем случае одной плоскости. Получившийся четырехугольник следует аппроксимировать двумя треугольниками, образованными сторонами четырехугольника и его диагональю, выбор которой осуществляется в зависимости от порядка нумерации вершин пирамиды [76]. Сечение состоит из двух треугольников.

Как правило, объем данных, описывающих первичную триангуляцию $|S_0|$, сопоставим с объемом данных, описывающих исходную сетку и значения сеточной функции. Таким образом, отсутствует возможность передачи всех этих данных за заданное время от серверной части системы визуализации к клиентской. Необходим этап, выполняющий подготовку данных меньшего объема $|S_s|$, описывающих поверхность S_s. При этом необходимо, чтобы поверхность S_s аппроксимировала первичную изоповерхность S_0 с приемлемой точностью [76]. Отметим, что «приемлемость» точности определяется субъективно. Важно, чтобы основные черты визуального образа, построенного на основе сокращенного набора данных, мало отличались от основных черт аналогичного образа, построенного на основе исходных данных.

Фактически, речь идет о разработке алгоритмов сжатия, описывающих триангуляцию данных до размеров, допускающих их передачу через медленные каналы связи за короткое время. Пусть желаемое время передачи измеряется секундами, а пропускная способность канала глобальной связи равна 1 Мбайт/с, тогда желаемый объем данных составляет мегабайты, а коэффициент сжатия данных измеряется тысячами. Данные, описывающие первичную триангуляцию, сжимаются незначительно стандартными методами сжатия без потерь. Таким образом, следует рассматривать алгоритмы сжатия данных с потерей точности, часто значительной. Следующие положения подтверждают приемлемость такого подхода:

- число узлов, описывающих первичную триангулированную изоповерхность S_0, может значительно превышать

число пикселей стандартного монитора пользователя, таким образом, при выводе на монитор первичной триангуляции часть информации будет неизбежно утеряна;

- при численном моделировании широко используются сгущающиеся сетки, причем разница между размерами самого длинного и самого короткого ребра сетки может составлять несколько порядков. В результате большое число узлов, описывающих зону сгущения сетки, проецируется в малое число пикселей экрана — визуально они сливаются, что также приводит к значительным потерям информации;
- в первую очередь визуально воспринимаются основные контуры и формы трехмерного объекта, спроецированного на двумерный экран. Таким образом, при изучении объекта «в целом» деталями можно пожертвовать. При необходимости, интересующий фрагмент объекта можно рассмотреть с большим увеличением и, соответственно, с меньшей потерей точности.

9.4.2. Виды данных, описывающих триангуляцию

Триангуляция описывается множеством пронумерованных узлов, каждому из которых поставлены в соответствие координаты трехмерного пространства (будем называть эту информацию координатной). Далее, на множестве узлов определяется множество треугольников, каждый из которых задается тройкой целых чисел — номеров узлов, являющихся его вершинами (эту информацию будем называть топологической).

- Рассмотрим две возможности сокращения объема данных, описывающих триангуляцию.
- Раздельное сжатие координатной и топологической информации.

Совместное сжатие координатной и топологической информации — формирование новой аппроксимирующей поверхности.

Можно предположить, что второй вариант предпочтительнее, поскольку позволяет учесть дополнительные знания о структуре сжимаемого объекта. Тем не менее, оценим объемы данных, соответствующих координатным и топологическим данным на примере пространственной триангуляции.

Пусть n — число узлов триангуляции, а m — число треугольников. Тогда, объем памяти, необходимой для хранения координат в трехмерном пространстве, в предположении задания координат вещественными числами одинарной точности (4 байта на число):

$$V_c(S_0) = 12n.$$

Объем данных, необходимых для хранения номеров вершин треугольников, в предположении, что общее число вершин в поверхности не превышает $2^{32} \approx 4 \cdot 10^9$ и для их адресации достаточно 4-байтовых целых чисел:

$$V_t(S_0) = 4 \cdot 3 \cdot m = 12m.$$

В случае заданной на множестве точек триангуляции, топологически эквивалентной планарной триангуляции, число треугольников не более чем вдвое превышает число точек. Таким образом, $V_t \sim 24n = 2V_c$. Следовательно, общий объем данных:

$$|S_0| = V_c(S_0) + V_t(S_0) = 36n.$$

Таким образом, $\frac{V_c(S_0)}{|S_0|} = \frac{2}{3}$ общего объема данных составляет целочисленная информация, описывающая связи между узлами.

Ситуация усугубляется при дальнейшем увеличении числа точек сетки. Для адресации большего числа вершин следует использовать 8-байтовые числа, соответственно, $V_t \sim 4V_c$, и доля топологической информации составляет уже $\frac{4}{5}$. Более того, триангуляция, определенная на точках трехмерно-

го пространства, не обязательно топологически эквивалентна планарной. В общем случае она может быть представлена произвольным набором непересекающихся треугольников, причем отношение числа этих треугольников к общему числу точек может быть сколь угодно велико. Примером служит конструкция, состоящая из $k \geq 3$ точек, расположенных на окружности, и $p \geq 2$ точек, расположенных на прямой, проходящей через центр окружности перпендикулярно ее плоскости. Каждая из лежащих на окружности точек соединена со всеми точками, лежащими на прямой, и наоборот (рис. 94). Таким образом, любые две соседние точки окружности и две соседние точки оси являются вершинами некоторой пирамиды.

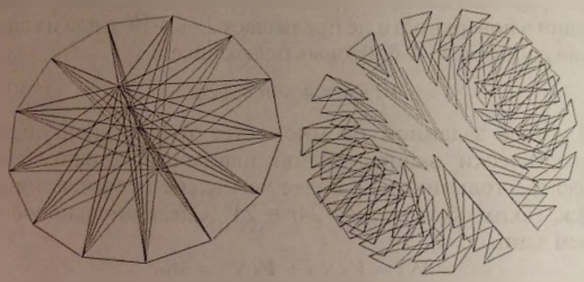


Рис. 94. Заполнение пространства пирамидами

В такой конструкции $m = k(2p - 1)$ различных треугольников.

При $k = p = \frac{n}{2}$:

$$m = \frac{n(n-1)}{2} = O(n^2).$$

Таким образом, отношение объема топологической информации $V_t(S_0) = 8 \cdot m = 4n(n - 1)$ к объему координатной

информации оказывается пропорционально общему числу точек $24\frac{n(n-1)}{2}/12n \approx n$ и с ростом числа вершин может быть сколь угодно велико. Следовательно, разработка алгоритмов сжатия координатной информации в отсутствие эффективных алгоритмов сжатия топологической информации нецелесообразна.

В настоящее время нет общепринятых алгоритмов, обеспечивающих многократное сжатие без потерь целочисленных данных, описывающих топологию сеток. Существующие специальные методы сжатия триангуляций, среди которых наиболее эффективен метод шелушения [77], существенно ориентированы на планарность сжимаемых графов и не обобщаются непосредственно на случай триангуляций, описывающих объекты трехмерного пространства.

Таким образом, основной интерес представляют алгоритмы, выполняющие совместное сжатие с потерей точности топологической и координатной информации. Результатом их работы должна быть некоторая триангулированная поверхность, аппроксимирующая исходную триангуляцию изоповерхности, но содержащая значительно меньшее, чем в исходной триангуляции, количество узлов.

9.4.3. Метод редукции

Известен ряд методов огрубления триангуляций, в том числе методы кластеризации, детализации, синтеза, редукции и другие. Метод редукции представляет наибольший интерес. Основная идея метода заключается в итерационном удалении из исходной поверхности некоторого количества узлов таким образом, чтобы триангуляция, определенная на оставшихся точках, аппроксимировала исходную поверхность с требуемой точностью.

Редукция исходной поверхности выполняется с помощью нескольких шагов просмотра [78, 79]. На каждом шаге исключение узлов происходит последовательно. Для каждого узла

поверхности проверяется возможность его удаления без нарушения заданной точности аппроксимации. При удалении узла и опирающихся на него треугольников соседние с ним вершины соединяются друг с другом так, чтобы образовались треугольники, аппроксимирующие поверхность наилучшим образом. Для контроля точности аппроксимации на каждом шаге сохраняется некоторая информация об удаленных узлах и треугольниках. В простейшем случае удаленные узлы накапливаются в списках, ассоциированных с порожденными треугольниками. Для повышения качества аппроксимации удаление узлов происходит итерационно, при этом на каждом шаге узлы удаляются равномерно по всей поверхности. Не удаляются узлы, смежные с уже удаленными на этом шаге узлами. Просмотр повторяется, если при выполнении последнего шага был удален хотя бы один узел.

Рис. 95. Сжатие редукцией, ошибка аппроксимации 2%

Рис. 96. Сжатие редукцией, ошибка аппроксимации 0,1%

Рис. 97. Сжатие редукцией на трех процессорах, ошибка аппроксимации 0,1%

Платой за соблюдение точности аппроксимации и высокое качество получаемых с помощью методов редукции визуальных образов является значительно большее, чем у других методов, время сжатия. На рисунках 95 и 96 представлены результаты сжатия тестовой поверхности $f = x\sin x + y\sin y$, заданной в области $x \in [-4, 4]$, $y \in [-4, 4]$. На рисунке 99 пред-

ставлен результат сжатия с помощью редукции сферы (рисунок 98).

Аппроксимируемая изоповерхность может иметь достаточно сложную топологию, например, на одно ребро может опираться более двух треугольников. Пример самопересекающейся поверхности (круг пересекается фрагментом цилиндра) и результат ее сжатия с помощью метода редукции приведен на рисунке 100.

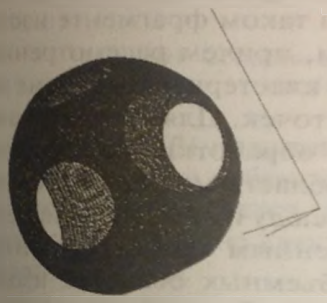


Рис. 98. Триангулированная сфера: точек 98 880, треугольников 195 884

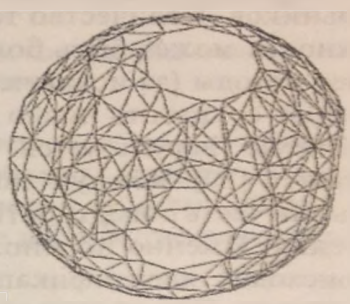


Рис. 99. Сжатие сферы редукцией: точек 215, треугольников 375

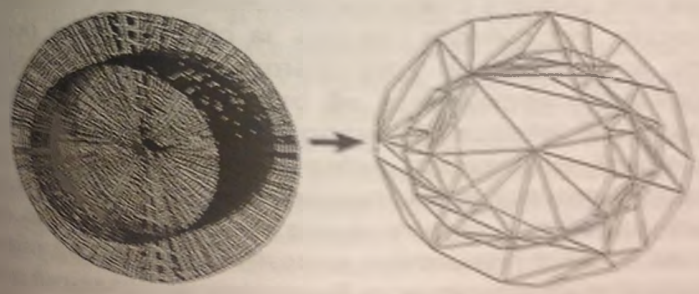


Рис. 100. Сжатие редукцией самопересекающейся поверхности, ошибка аппроксимации 5%

поверхности проверяется возможность его удаления без нарушения заданной точности аппроксимации. При удалении узла и опирающихся на него треугольников соседние с ним вершины соединяются друг с другом так, чтобы образовались треугольники, аппроксимирующие поверхность наилучшим образом. Для контроля точности аппроксимации на каждом шаге сохраняется некоторая информация об удаленных узлах и треугольниках. В простейшем случае удаленные узлы накапливаются в списках, ассоциированных с порожденными треугольниками. Для повышения качества аппроксимации удаление узлов происходит итерационно, при этом на каждом шаге узлы удаляются равномерно по всей поверхности. Не удаляются узлы, смежные с уже удаленными на этом шаге узлами. Просмотр повторяется, если при выполнении последнего шага был удален хотя бы один узел.

Рис. 95. Сжатие редукцией, ошибка аппроксимации 2%

Рис. 96. Сжатие редукцией, ошибка аппроксимации 0,1%

Рис. 97. Сжатие редукцией на трех процессорах, ошибка аппроксимации 0,1%

Платой за соблюдение точности аппроксимации и высокое качество получаемых с помощью методов редукции визуальных образов является значительно большее, чем у других методов, время сжатия. На рисунках 95 и 96 представлены результаты сжатия тестовой поверхности $f = x\sin x + y\sin y$, заданной в области $x \in [-4, 4]$, $y \in [-4, 4]$. На рисунке 99 пред-

ставлен результат сжатия с помощью редукции сферы (рисунок 98).

Аппроксимируемая изоповерхность может иметь достаточно сложную топологию, например, на одно ребро может опираться более двух треугольников. Пример самопересекающейся поверхности (круг пересекается фрагментом цилиндра) и результат ее сжатия с помощью метода редукции приведен на рисунке 100.

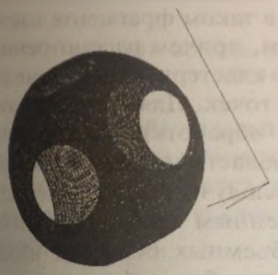


Рис. 98. Триангулированная сфера: точек 98 880, треугольников 195 884

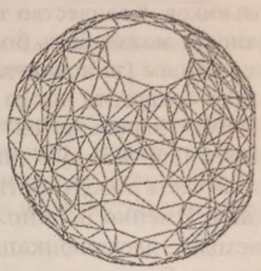


Рис. 99. Сжатие сферы редукцией: точек 215, треугольников 375

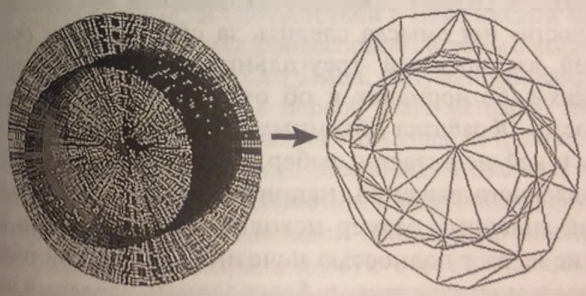


Рис. 100. Сжатие редукцией самопересекающейся поверхности, ошибка аппроксимации 5%

9.4.4. Заполняющие пространство триангуляции

Как уже упоминалось, возможно, что исследуемая сеточная функция принимает одинаковое значение во всех точках некоторой трехмерной области. В этом случае сформированная изоповерхность может содержать фрагменты, заполненные треугольниками, каждое ребро которых является ребром самопересечения поверхности, то есть на каждое из таких ребер опирается более двух треугольников. Количество точек в таком фрагменте изоповерхности может быть большим, причем рассмотренные ранее методы (за исключением кластеризующего) не позволяют сократить число этих точек. Для решения этой проблемы используется этап обработки заполненных областей [80]. Каждую такую область можно рассматривать как часть поверхности, повсюду имеющую самопересечения. Именно по множественным самопересечениям происходит идентификация объемных областей изоповерхности. При сжатии указанных областей отключаются все проверки на допустимость операций удаления ребер, которые выполняются для обычной поверхности. Все рассмотренные ранее проверки в данном случае не нужны, так как для сплошь самопересекающейся объемной «поверхности» нет смысла следить за сохранением особенностей взаимосвязей треугольников, нельзя говорить о каких-либо нормалях и об отклонении от требуемой точности. Обработка объемных областей заканчивается, когда в ней не остается ребер, длины которых меньше определенного значения (например, $L/10$, где L — характерный линейный размер исходных данных). Это ограничение не дает полностью исчезнуть объемной области в процессе многочисленных безусловных удалений ребер. В результате такой обработки на месте старой области появится новая, но с гораздо меньшим числом точек.

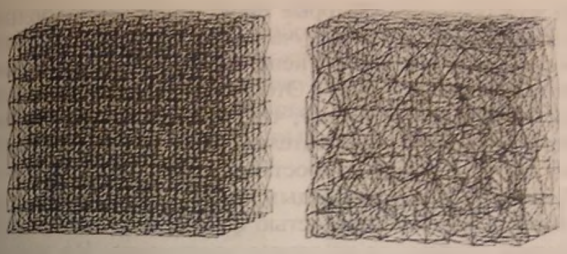


Рис. 101. Область, заполненная треугольниками (до и после сжатия)

На рисунке 101 показан пример обработки изоповерхности, занимающей кубическую область. Видно, что количество точек уменьшается равномерно по всему объему, что обеспечивает вполне адекватное восприятие поведения сеточной функции.

9.4.5. Параллельные алгоритмы построения аппроксимирующих триангуляций

Параллельные алгоритмы построения и аппроксимации изоповерхности разработаны на основе метода геометрического параллелизма. В соответствии с ним, каждый вычислительный модуль многопроцессорной системы обрабатывает компактную часть сетки (домен, полученный, например, с помощью изложенных в первой главе методов). Передач данных между процессорами на этом этапе не требуется, что, тем не менее, не обязательно обеспечивает высокую эффективность параллельной обработки. Следует подчеркнуть, что при параллельной визуализации данных, определенных на сетках большого размера, в первую очередь ставится задача обеспечения самой возможности выполнения визуализации, во вторую очередь преследуется цель снижения времени обработки для комфортной работы в интерактивном режиме.

При этих условиях некоторые процессоры могут большую часть времени не принимать участия в обработке, снижая таким образом эффективность использования вычислительной мощности системы в целом. Это не является существенным недостатком, хотя бы потому, что при интерактивной работе между запросами пользователя может проходить заметное время, в течение которого простаивают все процессоры.

После формирования каждым процессором аппроксимированных с заданной точностью фрагментов поверхности, результаты собираются на одном процессоре. На нем происходит объединение фрагментов в единое целое, «сшивка» границ фрагментов и аппроксимация уже всей поверхности. Полученная триангуляция передается по каналам Интернет или локальной сети на персональный компьютер пользователя, где и происходит окончательное построение изображения. Двухэтапная обработка: сначала по частям с сохранением границ доменов, затем всей поверхности, а фактически именно границ между доменами, приводит к тому, что области, прилегающие к границам доменов, сжимаются несколько хуже, чем внутренние зоны и могут выделяться на рисунке или на экране монитора. Пример огрубления тестовой функции (рис. 96) на трех процессорах (перед итоговым сглаживанием на одном процессоре) приведен на рисунке 97, на котором выделяются две горизонтальные полосы, соответствующие границам между тремя доменами.

9.4.6. Многоуровневое огрубление больших сеток

В общем случае стратегии, рассматриваемой в предыдущем разделе, не достаточно для обработки произвольно большой сетки. Напомним, что при огрублении фрагментов поверхности нельзя модифицировать узлы, лежащие на границах между процессорами. Их отчетливо видно на рисунках 97, 102 и 103. Несогласованная модификация узлов приведет к появлению таких артефактов, как разрывы

поверхности и пересечение треугольников, что резко снижает качество визуального образа. Поэтому на рисунке 102 прослеживаются дуги, точек на которых больше, чем внутри полос, где таковых почти нет. Дуги и есть границы между процессорами. Следующим шагом следует собрать все получившиеся фрагменты на один процессор и выполнить окончательную обработку.

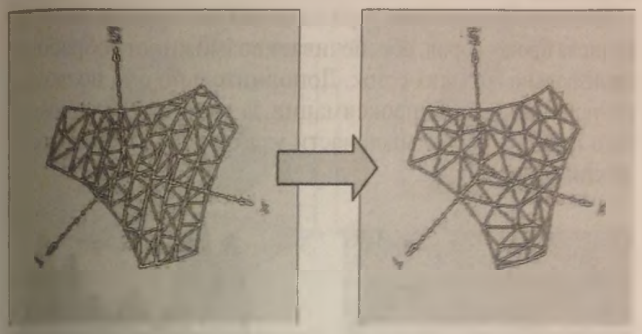


Рис. 102. Результат огрубления сетки, предварительно обработанной на 9 процессорах

На рисунке 103 показан результат визуализации с использованием 25 процессоров. Сравнивая его с рисунком 97, можно увидеть, что чем больше процессоров, тем большая часть точек не подлежит редуцированию. Таким образом, возможна ситуация, в которой поверхность, полученная объединением всех огрубленных фрагментов сетки, не может быть размещена на одном единственном процессоре в силу нехватки на нем оперативной памяти.

Естественный выход из положения — использование иерархического принципа огрубления. Все процессоры следует разбить на группы. Число процессоров в группе должно выбираться из условия возможности объединения на одном из них (назовем его процессором первого уровня)

всех обработанных процессорами группы данных. П
выполнения огрубления процессорами первого ур
следует, действуя по аналогии, разбить процессоры пе
го уровня на группы, в каждой из которых выделить по
ному процессору второго уровня. Полученная таким об
зом иерархия на последнем уровне должна содержать од
процессор, обеспечивающий получение окончательно
результата.

Многоуровневая схема, при наличии достаточно больш
го числа процессоров, обеспечивает возможность обработк
произвольно больших сеток. Дополнительно она позволяе
улучшить качество аппроксимации за счет более равномер
ного исключения большей части узлов, расположенных на
фиктивных границах.

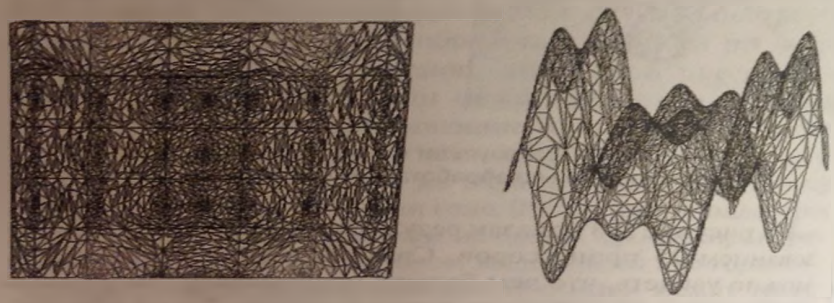


Рис. 103. Визуализация изоповерхности на 25 процессорах с помощью метода редуцирования

9.4.7. Примеры визуализации

На рисунке 104 представлены результаты визуализации изоповерхностей плотности, полученные при численном моделировании задачи обтекания сферы набегающим потоком газа: слева исходные данные (со всеми точками), справа — результаты огрубления (уже меньшее число то-

чек). С одной стороны, искажения достаточно велики. Например, изначально круглое отверстие стало треугольным. С другой стороны, отверстие осталось. Соответствующий фрагмент моделируемой области можно выделить отдельно и выполнить визуализацию с большим разрешением и меньшими потерями точности. В результате, будет видно исходное круглое отверстие, каковым оно и является.

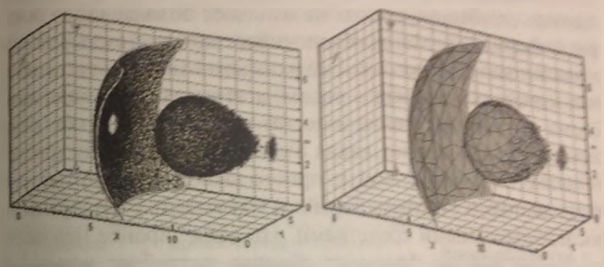


Рис. 104. Изоповерхность: исходная и после сжатия с точностью 0,5%

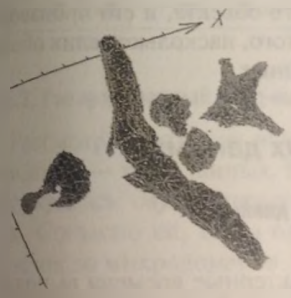


Рис. 105. Изоповерхности поля плотности, визуализация разработанной системой RemoteViewer

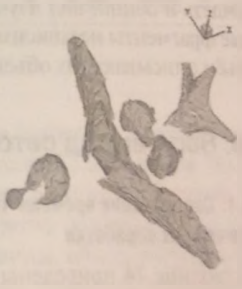


Рис. 106. Изоповерхности поля плотности, визуализация системой Tecplot

Приведенные на рисунках 105 и 106 результаты показывают хорошее совпадение формы изоповерхностей, полученных разными инструментами, что подтверждает высокое качество образов, формируемых описанными алгоритмами визуализации. На них представлены результаты визуализации изоповерхностей плотности с помощью обсуждаемых методов с потерей точности и с помощью системы Tecplot, выполняющей визуализацию без огрубления. Визуально они близки: огрубление данных не означает безнадежную потерю информации. Метод обеспечивает возможность анализа произвольно большого объема данных — позволяет оценить вид изучаемого объекта в целом, а это и есть главная проблема — увидеть течение в целом на большой сетке. Если заранее точно известно, какая именно часть объекта требует детального изучения, можно просто выводить необходимую часть данных в отдельный файл и изучать их стандартными последовательными средствами. Но если заранее неизвестно, какой именно фрагмент объекта потребует изучения? Если именно общий взгляд на весь объект позволяет определить потенциально интересные зоны, то без рассмотренной технологии не обойтись. Она нужна именно для того, чтобы увидеть и общий вид изучаемого объекта, и его произвольные фрагменты независимо от того, насколько велик общий объём описывающих объект данных.

9.5. Ввод-вывод сеточных данных

9.5.1. Соотношение времени чтения данных и времени их обработки

В таблице 14 приведены характерные времена выполнения визуализации гладкой функции. В первом столбце указаны размеры обрабатываемой сетки. Далее в 3-х столбцах представлены значения длительностей интервалов времени требуемых на построение изоповерхности с учетом времени

чтения. Последние три значения отражают чистое время построения изоповерхности и передачи требуемых данных на клиентскую часть (без учета времени чтения данных). Анализу подвергался файл, находящийся на отдельном файловом сервере, поэтому ввод данных выполнялся по протоколу NFS. Представленные данные показывают, что время ввода-вывода данных составляет существенную, а часто — определяющую часть общего времени визуализации. Сокращение времени чтения данных с помощью обсуждаемых в конце главы методов обеспечивает возможность комфортной интерактивной визуализации, поскольку даже для сеток содержащих 250 млн узлов, время обработки на нескольких десятках процессоров составляет доли секунды.

Таблица 14

Время выполнения визуализации на 20 процессорах

Размеры сетки	Время чтения и время обработки, с			Время обработки, с		
50×50×10	0,050	0,042	0,047	0,011	0,009	0,009
500×500×100	5,043	4,745	5,066	0,075	0,075	0,076
500×500×1000	41,628	36,046	36,139	0,660	0,666	0,663

9.5.2. Распределенный ввод-вывод

Рассмотрим один из наиболее длительных этапов визуализации — ввод данных. В разделе 7.7.2 (рисунок 78) уже обсуждалась *двухуровневая схема* обработки больших сеток. Согласно ей, сетка однократно разбивается на большое число микродоменов. Вершины, объединяемые одним микродоменом, сохраняются на внешнем носителе как единое целое. На этапе визуализации назначается на процессор и читается тоже сразу вся группа вершин микродомена. Тем самым, во-первых, по сравнению с поэлемент-

ной обработкой, существенно повышается скорость чтения данных. Во-вторых, обеспечивается возможность независимого параллельного чтения процессорами микродоменов. В-третьих, при решении задач визуализации, появляется дополнительная возможность значительного уменьшения объема хранимых данных. Рассмотрим последнюю возможность подробнее.

Отображаемый визуальный образ практически всегда является результатом аппроксимации исходных цифровых данных, что уже вносит в них некоторые искажения. Дополнительные искажения обусловлены проецированием трехмерных данных на двумерные устройства отображения. Используемые сегодня устройства визуализации имеют дискретную природу, как правило, они отображают растровые изображения, что также ведет к дополнительным потерям, вызванным дискретизацией визуализируемого объекта. Этот список можно продолжать, но уже сейчас сделаем вывод о том, что на этапе записи данных, предназначенных для визуализации, допустима частичная потеря точности их представления.

9.5.3. Огрубление и сжатие скалярных сеточных функций

Время чтения больших объемов сеточных данных составляет существенную часть общего времени их обработки в ходе визуализации (равно как и время записи в ходе вычислений). Для его сокращения можно прибегнуть к сжатию данных перед их записью на диск, однако, использование стандартных методов сжатия без потерь применительно к вещественным числам неэффективно. Для увеличения степени сжатия данных, предназначенных для анализа методом визуализации, допустимо игнорировать часть младших разрядов мантиссы каждого из значений сеточной функции, что значительно увеличивает коэффициент компрессии. Относительная ошибка огрубления данных, полученная в результате отбрасывания k двоичных разрядов

мантиссы, приближенно может быть оценена как 2^{k-24}. Использование иерархической двухуровневой схемы хранения позволяет задавать параметр огрубления независимо для каждого из записываемых блоков, что позволяет записывать значения сеточной функции, соответствующие разным фрагментам сетки, с разной точностью.

На графике (рис. 107) приведен фрагмент параболы $y = 1 - (x - 1)^2$, построенной по точкам с координатами $x_i = i/21$. Разные кривые соответствуют разному числу оставленных в представлении значений y_i бит мантиссы. Визуально кривая, соответствующая семи оставленным битам, неотличима от исходной неогрубленной кривой. Подчеркнем, что семь оставленных бит — это 18 отброшенных. Именно эти 18 бит плохо поддаются компрессии с помощью алгоритмов сжатия данных общего назначения. Обнуление этих бит, практически ничего не меняя с точки зрения визуального восприятия образа данных, позволяет радикально повысить степень сжатия хранимых на диске данных. Данные сжимаются в тысячи раз без ощутимых потерь в их визуальном представлении. При подготовке обсуждаемых далее примеров использовалась кроссплатформенная библиотека сжатия общего назначения *zlib* [81] и библиотека распределенного ввода/вывода решеток *Bjn10* [82].

Дополнительно увеличить коэффициент компрессии данных можно за счет перегруппировки разрядов вещественных чисел. Идея заключается в том, что соседние в массиве числа с большой вероятностью имеют одинаковый порядок и одинаковый знак. Выделим биты, отвечающие за сохранение порядка и знака каждого числа в один массив, а остальные биты — в другой. Раздельная компрессия получившихся массивов в общем случае обеспечит меньший объем сжатых данных, чем компрессия исходных чисел.

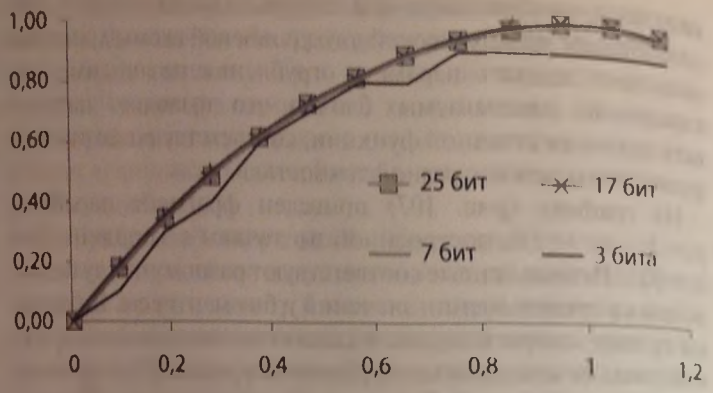


Рис. 107. Зависимость вида кривой от числа необнулённых бит мантиссы 32-битных float-чисел

На рисунке 107 показано влияние огрубления на вид визуализируемых данных. Фактически, все представленные кривые совпадают, что свидетельствует в пользу возможности применения данного подхода на практике.

Результаты записи тестового файла, содержащего 10^9 узлов, с различными значениями точности представления данных представлены на рисунке 108. Кривая ошибки показывает относительную локальную ошибку, выраженную в процентах. Вторая кривая соответствует размеру хранимой на диске информации, выраженной в гигабайтах. Исходные данные — 4-байтовые вещественные числа. Объем массива без сжатия и огрубления — 3,54 ГБ.

Рассмотрим пример компрессии файла, содержащего 10^9 узлов. Пусть каждому узлу соответствует одно значение вещественной скалярной функции. В этом случае бинарный образ массива будет занимать около 4 Гбайт.

Двоичное представление каждого вещественного числа описывается четырьмя байтами, что позволяет рассматривать одномерный массив чисел как двумерный массив

байтов. Перекомпануем массивы по старшинству байтов (рис. 108). Младшие байты поместим в первый массив, следующие по старшинству — во второй и так далее. Полученные массивы могут быть сжаты с более высоким коэффициентом, поскольку во многих случаях близко расположенные в исходном массиве вещественные числа имеют одинаковый или близкий по величине порядок и значения старших цифр мантиссы. Массивы, содержащие соответствующие байты, сжимаются стандартными методами с высокой степенью. Как правило, степень сжатия остальных массивов ниже.

Рис. 108. Перегруппировка байтов элементов массивов вещественных чисел

Если сжать массив с помощью описанного метода (отдельно выполнить компрессию записанных в один массив первых байтов, отдельно вторых байт и так далее), то суммарный объем получившихся четырех массивов составит примерно 1 ГБ. Следующим шагом в увеличении коэффициента сжатия станет сжатие с контролируемой потерей точности: сжатие после предварительного обнуления младших бит мантиссы каждого числа.

На рисунке 109 представлена зависимость объема результата сжатия от количества обнуленных бит мантиссы вещественных чисел. Там же приведена вторая кривая, показывающая привносимую относительную ошибку, выраженную в процентах. Как правило, объем данных падает с увеличением числа отбрасываемых бит. Например, если обнулить

14 младших бит, то для хранения результата сжатия потребуется 100 МБ. Если обнулить 18 бит, то потребуется только 28 МБ. Отметим, что уже при 14 обнуленных битах мантиссы, для хранения каждого вещественного числа потребуется меньше одного бита (не байта!). Объем данных сокращается существенно, но качество визуального образа остается высоким (рисунок 107).

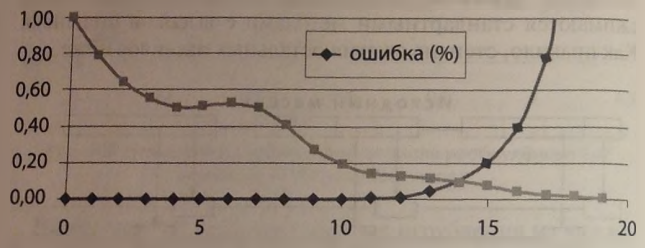


Рис. 109. Зависимость объема хранимых данных и точности их представления от числа отброшенных бит мантиссы 4-байтовых float-чисел

На рисунке 110 представлены достигнутые коэффициенты сжатия с огрублением сеточных данных, полученных при численном моделировании системы охлаждения процессора на сетке 1000×3500×150 = 525 млн узлов. Результаты, представленные на рисунках 109 и 110, показывают, что объем хранимых данных может быть сокращен в сотни и более раз без существенной, с точки зрения визуализации, потери точности их представления. Данный подход обеспечивает возможность сокращения на несколько порядков и времени ввода-вывода данных и объемов требуемой дисковой памяти.

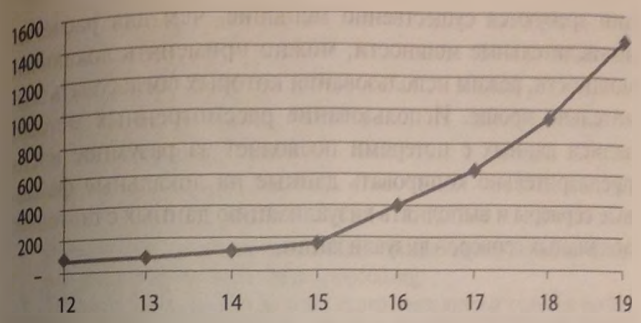


Рис. 110. Зависимость коэффициента сжатия результатов моделирования теплового режима устройства охлаждения от числа отброшенных бит мантиссы 4-байтовых float-чисел

Объем данных в 28 МБ можно прочесть с диска достаточно быстро, что обеспечивает возможность работы с системой визуализации в интерактивном режиме. Более того, подобный объем вполне можно передать с суперкомпьютерного центра на персональный компьютер. В принципе, появляется возможность, запустив сервер визуализации на персональном компьютере, обработать на нем результаты в потоковом режиме — микродомен за микродоменом, хотя скорость такой обработки и оставит желать лучшего. Представляет интерес и развитие в другом направлении — возможность использовать для отображения данных выделенного корпоративного сервера визуализации.

Одной из существенных организационных проблем анализа результатов расчетов, выполненных на вычислительных мощностях центов коллективного пользования, является наличие на них систем очередей. С точки зрения пользователя, сервер визуализации должен запускаться именно тогда, когда появилась необходимость выполнить визуализацию. Система очередей является в этом отношении существенной помехой. Учитывая, что для визуализа-

ции требуются существенно меньшие, чем для расчетов, вычислительные мощности, можно применять локальные мощности, режим использования которых согласовать значительно проще. Использование рассмотренных методов записи данных с потерями позволяет за разумное время предварительно копировать данные на локальные файловые серверы и выполнять визуализацию данных с помощью локальных серверов визуализации.

Список ссылок

1. Рейтинг 500 самых мощных общественно известных компьютерных систем мира. URL: http://top500.org/
2. Таблицы технических данных суперкомпьютеров списков top500.org. URL: http://top500.org/static/lists/2011/11/TOP500_201111.xls, http://top500.org/static/lists/2010/11/TOP500_201011.xls, http://top500.org/static/lists/2009/11/TOP500_200911.xls, ...
3. *Кнут Д.Э.* Искусство программирования. — Т. 2. — Получисленные алгоритмы, 3-е издание. : Пер с английского: — М.: Вильямс, 2001. — 832 с.
4. *Дейкстра Э.* Взаимодействие последовательных процессов // Языки программирования / Ред. Ф. Женуи, пер. с англ. В.П. Кузнецова, под ред. В.М. Курочкина. — М. : Мир, 1972. — С. 9—87.
5. *Dijkstra E.W.* Cooperating Sequential Processes in Programming Languages / Ed. F. Genuys. — NY: Academic Press, New York, 1968.
6. *Кнут Д.Э.* Искусство программирования. — Т. 3. — Сортировка и поиск, 2-е изд.— М.: Вильямс, 2001. — 824 с.
7. *Воеводин В.В.* Суперкомпьютеры и КПД паровоза. URL: http://www.intuit.ru/video/70/
8. *Лацис А.О.* Параллельная обработка данных. — М.: Академия, 2010, 336 с.
9. *Кузьминский М.* Altix UV на платформе x86 // Открытые системы. СУБД. — 2011. — № 3. — С. 17—19. URL: http://www.osp.ru/os/2011/03/13008199/
10. *Крюков В.А.* Операционные системы распределенных вычислительных систем (распределенные ОС). URL: http://parallel.ru/krukov/
11. *Стивенс. У.* Unix: взаимодействие процессов. Мастер-класс / Пер. с англ. Д. Солнышков. — Спб: Питер, 2002. — 576 с.
12. *Андреев А.А.* Функциональные уравнения. Учебное издание. Серия А: Математика. — Вып. 3 / А.А. Андреев, Ю.Н. Кузьмин, А.Н. Савин. — Самара: Пифагор, 1997.
13. *Эндрюс Г.Р.* Основы многопоточного параллельного и распределенного программирования. — М.: Издательский дом «Вильямс», 2003. — 512 с.: ил. — Парал. тит. англ.