Глава 1

# Обзор области параллельных вычислений

Представьте себе такую картину: несколько автомобилей едут из пункта А в пункт В. Машины могут бороться за дорожное пространство и либо следуют в колонне, либо обгоняют друг друга (попадая при этом в аварии!). Они могут также ехать по параллельным полосам дороги и прибыть почти одновременно, не "переезжая" дорогу друг другу. Возможен вариант, когда все машины поедут разными маршрутами и по разным дорогам.

Эта картина демонстрирует суть параллельных вычислений: есть несколько задач, которые должны быть выполнены (едущие машины). Можно выполнять их по одной на одном процессоре (дороге), параллельно на нескольких процессорах (дорожных полосах) или на распределенных процессорах (отдельных дорогах). Однако задачам нужно синхронизироваться, чтобы избежать столкновений или задержки на знаках остановки и светофорах.

Данная книга — это "атлас" параллельных вычислений. В ней рассматриваются типы автомашин (процессов), возможные пути их следования (приложения), схемы дорог (аппаратного обеспечения) и правила дорожного движения (взаимодействие и синхронизация). Так что заправьте полный бак и приготовьтесь к старту.

В данной главе рассказывается о надписях на карте параллельного программирования. В разделе 1.1 представлены основные понятия. В разделах 1.2 и 1.3 описаны виды аппаратной части и приложения, которые делают параллельное программирование интересным и перспективным. В разделах с 1.4 по 1.8 описываются и иллюстрируются пять стилей программирования циклических вычислений: итеративный параллелизм, рекурсивный параллелизм, "производители и потребители", "клиенты и серверы" и, наконец, взаимодействующие каналы. В последнем разделе определена нотация программ, используемая в дальнейшем.

В следующих главах подробно рассмотрены приложения и методы программирования. Книга состоит из трех частей, в которых описано программирование с разделяемыми переменными, распределенное (основанное на сообщениях) и синхронное параллельное. Введение в каждую часть и главу служит картой маршрута, подводя итоги пройденного и предстоящего пути.

## 1.1. Суть параллельного программирования

Параллельная программа содержит несколько процессов, работающих совместно над выполнением некоторой задачи. Каждый процесс — это последовательная программа, а точнее — последовательность операторов, выполняемых один за другим. Последовательная программа имеет *один поток управления*, а параллельная — *несколько.*

Совместная работа процессов параллельной программы осуществляется с помощью их *взаимодействия*. Взаимодействие программируется с применением разделяемых переменных или пересылки сообщений. Если используются разделяемые переменные, то один процесс осуществляет запись в переменную, считываемую другим процессом. При пересылке сообщений один процесс отправляет сообщение, которое получает другой.

При любом виде взаимодействия процессам необходима взаимная *синхронизация*. Существуют два основных вида синхронизации — взаимное исключение и условная синхронизация. *Взаимное исключение* обеспечивает, чтобы критические секции операторов не выполнялись одновременно. *Условная синхронизация* задерживает процесс до тех пор, пока не выполнится определенное условие. Например, взаимодействие процессов производителя и потребителя часто обеспечивается с помощью буфера в разделяемой памяти. Производитель записывает в буфер, потребитель читает из него. Чтобы предотвратить одновременное использование буфера и производителем, и потребителем (тогда может быть считано не полностью записанное сообщение), используется взаимное исключение. Условная синхронизация используется для проверки, было ли считано потребителем последнее записанное в буфер сообщение.

Как и другие прикладные области компьютерных наук, параллельное программирование прошло несколько стадий. Оно возникло благодаря новым возможностям, предоставленным развитием аппаратного обеспечения, и развилось в соответствии с технологическими изменениями. Через некоторое время *специализированные* методы были объединены в набор основных принципов и общих методов программирования.

Параллельное программирование возникло в 1960-е годы в сфере операционных систем. Причиной стало изобретение аппаратных модулей, названных *каналами*, или *контроллерами устройств*. Они работают независимо от управляющего процессора и позволяют выполнять операции ввода-вывода параллельно с инструкциями центрального процессора. Канал взаимодействует с процессором с помощью *прерывания* — аппаратного сигнала, который говорит: "Останови свою работу и начни выполнять другую последовательность инструкций".

Результатом появления каналов стала проблема программирования (настоящая интеллектуальная проблема) — теперь части программы могли быть выполнены в непредсказуемом порядке. Следовательно, пока одна часть программы обновляет значение некоторой переменной, может возникнуть прерывание, приводящее к выполнению другой части программы, которая тоже попытается изменить значение этой переменной. Эта специфическая проблема (*задача критической секции*) подробно рассматривается в главе 3.

Вскоре после изобретения каналов началась разработка многопроцессорных машин, хотя в течение двух десятилетий они были слишком дороги для широкого использования. Однако сейчас все крупные машины являются многопроцессорными, а самые большие имеют сотни процессоров и часто называются *машинами с массовым параллелизмом* (massively parallel processors). Скоро даже персональные компьютеры будут иметь несколько процессоров.

Многопроцессорные машины позволяют разным прикладным программам выполняться одновременно на разных процессорах. Они также ускоряют выполнение приложения, если оно написано (или переписано) для многопроцессорной машины. Но как синхронизировать работу параллельных процессов? Как использовать многопроцессорные системы для ускорения выполнения программ?

Итак, при использовании каналов и многопроцессорных систем возникают и возможности, и трудности. При написании параллельной программы необходимо решать, сколько процессов и какого типа нужно использовать, и как они должны взаимодействовать. Эти решения зависят как от конкретного приложения, так и от аппаратного обеспечения, на котором будет выполняться программа. В любом случае ключом к созданию корректной программы является правильная синхронизация взаимодействия процессов.

Эта книга охватывает все области параллельного программирования, но основное внимание уделяет *императивным программам* с явными параллельностью, взаимодействием и синхронизацией. Программист должен специфицировать действия всех процессов, а также их взаимодействие и синхронизацию. Это контрастирует с *декларативными программами*, например, функциональными или логическими, где параллелизм скрыт и отсутствуют чтение и запись состояния программы. В декларативных программах независимые части программы могут выполняться параллельно; их взаимодействие и синхронизация происходят неявно, когда одна часть программы зависит от результата выполнения другой. Декларативный под-

ход тоже интересен и важен (см. главу 12), но императивный распространен гораздо шире. Кроме того, для реализации декларативной программы на стандартной машине необходимо писать императивную программу.

Изучаются также параллельные программы, в которых процессы выполняются *асинхронно*, т.е. каждый со своей скоростью. Такие программы могут выполняться с помощью чередования процессов на одном процессоре или их параллельного выполнения на мультипроцессоре со многими командами и многими данными (MIMD-процессоре). К этому классу машин относятся также мультипроцессоры с разделяемой памятью, многомашинные системы с распределенной памятью и сети рабочих станций (см. следующий раздел). Несмотря на внимание к асинхронным параллельным вычислениям, в главе 3 мы описываем *синхронную* мультиобработку (SIMD-машины), а в главах 3 и 12 — связанный с ней стиль программирования, параллельного по данным.

# 1.2. Структуры аппаратного обеспечения

В данной главе дается обзор основных атрибутов архитектуры современных компьютеров. В следующем разделе описаны приложения параллельного программирования и использование архитектуры в них. Описание начинается с однопроцессорных систем и кэш-памяти. Затем рассматриваются мультипроцессоры с разделяемой памятью. В конце описываются машины с распределенной памятью, к которым относятся многомашинные системы с распределенной памятью и сети машин.

## 1.2.1. Процессоры и кэш-память

Современная однопроцессорная машина состоит из нескольких компонентов: центрального процессорного устройства (ЦПУ), первичной памяти, одного или нескольких уровней кэш-памяти (кэш), вторичной (дисковой) памяти и набора периферийных устройств (дисплей, клавиатура, мышь, модем, CD, принтер и т.д.). Основными компонентами для выпол-нения программ являются ЦПУ, кэш и память. Отношения между ними изображены на рис. 1.1.

Процессор выбирает инструкции из памяти, декодирует их и выполняет. Он содержит управляющее устройство (УУ), арифметико-логическое устройство (АЛУ) и регистры. УУ вырабатывает сигналы, управляющие действиями АЛУ, системой памяти и внешними устройствами. АЛУ выполняет арифметические и логические инструкции, определяемые набором инструкций процессора. В регистрах хранятся инструкции , данные и состояние машины (включая счетчик команд).

*Рис. 1.1. Процессор, кэш и память в современной вычислительной машине*

*Кэш* — это небольшой по объему, но скоростной модуль памяти, используемый для ускорения выполнения программы. В нем хранится содержимое областей памяти, часто используемых процессором. Причина использования кэш-памяти состоит в том, что в большинстве программ наблюдается *временная локальность*, означающая, что если произошло обращение к некоторой области памяти, то, скорее всего, в ближайшем будущем обращения к этой области повторятся. Например, инструкции внутри циклов выбираются и выполняются многократно.

Когда программа обращается к адресу в памяти, процессор сначала ищет его в кэше. Если данные находятся там (происходит *попадание* в кэш), то они считываются из кэша. Если данных в кэше нет (*промах*), то данные считываются из первичной памяти и в процессор,

и в кэш-память. Аналогично, когда программа записывает данные, они помещаются в первичную память и, возможно, в локальный кэш. В *сквозном кэше* данные помещаются в память немедленно, в *кэше с обратной записью* — позже. Ключевой момент состоит в том, что после записи содержимое первичной памяти временно может не соответствовать содержимому кэша.

Чтобы ускорить передачу (увеличить пропускную способность) между кэшем и первичной памятью, в элемент кэш-памяти обычно включают непрерывную последовательность слов из памяти. Эти элементы называются *блоками* или *строками* кэша. При промахе из памяти в кэш передается полная строка. Это эффективно, поскольку в большинстве программ наблюдается *пространственная локальность*, т.е. за обращением к одному слову памяти вскоре последуют обращения к другим близлежащим словам.

В современных процессорах обычно есть два типа кэша. Кэш уровня 1 находится ближе к процессору, а кэш уровня 2 — между кэшем уровня 1 и первичной памятью. Кэш уровня 1 меньше и быстрее, чем кэш уровня 2, и зачастую иначе организован. Например, кэш-память уровня 1 обычно *отображается непосредственно*, а кэш уровня 2 является *множественно-ассоциативным*.[1] Кроме того, кэш уровня 1 часто содержит отдельные области кэш-памяти для инструкций и данных, в то время как кэш уровня 2 обычно *унифицирован*, т.е. в нем хранятся и данные, и инструкции.

Проиллюстрируем различия в скорости работы уровней иерархии памяти. Доступ к регистрам происходит за один такт работы процессора, поскольку они невелики и находятся внутри ЦПУ. Данные кэш-памяти уровня 1 также доступны за один-два такта. Однако для доступа к кэш-памяти уровня 2 необходимо порядка 10 тактов, а к первичной памяти — от 50 до 100 тактов. Аналогичны и различия в размере типов памяти: ЦПУ содержит несколько десятков регистров, кэш уровня 1 — несколько десятков килобайт, кэш уровня 2 — порядка мегабайта, а первичная память — десятки и сотни мегабайт.

## 1.2.2. Мультипроцессоры с разделяемой памятью

В *мультипроцессоре с разделяемой памятью* процессоры и модули памяти связаны с помощью *соединительной сети* (рис. 1.2). Процессоры совместно используют первичную память, но каждый из них имеет собственный кэш.

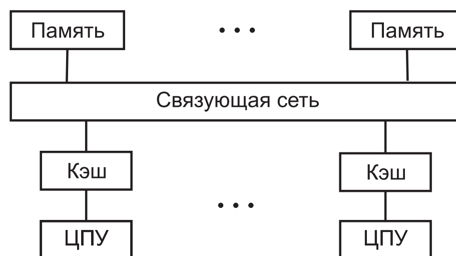


*Рис. 1.2. Структура мультипроцессоров с разделяемой памятью*

[1] В непосредственно отображаемом кэше каждый адрес памяти отображается в один элемент кэша, а в множественно-ассоциативном — во множество элементов (обычно два или четыре). Таким образом, если два адреса памяти отображаются в одну и ту же ячейку, в непосредственно отображаемом кэше может находиться только ячейка, к которой производилось самое последнее обращение, а в ассоциатив-ном — обе ячейки. С другой стороны, непосредственно отображаемый кэш быстрее, поскольку проще выяснить, есть ли данное слово в кэше.

В небольшом мультипроцессоре, имеющем от двух до (порядка) 30 процессоров, соединительная сеть реализована в виде шины памяти или, возможно, матричного коммутатора. Такой мультипроцессор называется *однородным* (UMA machine — от "uniform memory access"), поскольку время доступа каждого из процессоров к любому участку памяти одинаково. Однородные машины также называются *симметричными мультипроцессорами*.

В больших мультипроцессорах с разделяемой памятью, включающих десятки или сотни процессоров, память организована иерархически. Соединительная сеть имеет вид древообразного набора переключателей и областей памяти. Следовательно, одна часть памяти ближе к определенному процессору, другая — дальше от него. Такая организация памяти позволяет избежать перегрузки, возникающей при использовании одной шины или коммутатора, и приводит к неравным временам доступа, поэтому такие мультипроцессоры называются *неоднородными* (NUMA machines).

В машинах обоих типов у каждого процессора есть собственный кэш. Если два процессора ссылаются на разные области памяти, их содержимое можно безопасно поместить в кэш каждого из них. Проблема возникает, когда два процессора обращаются к одной области памяти примерно одновременно. Если оба процессора только считывают данные, в кэш каждого из них можно поместить копию данных. Но если один из процессоров записывает в память, возникает *проблема согласованности кэша*: в кэш-памяти другого процессора теперь содержатся неверные данные. Значит, необходимо либо обновить кэш другого процессора, либо признать содержимое кэша недействительным. В каждом мультипроцессоре протокол согласования кэш-памяти должен быть реализован аппаратно. Один из способов состоит в том, чтобы каждый кэш "следил" за адресной шиной, отлавливая ссылки на области памяти, находящиеся в нем.

Запись в память также приводит к проблеме *согласованности памяти*: когда в действительности обновляется первичная память? Например, если один процессор выполняет запись в область памяти, а другой позже считывает данные из этой области, будет ли считано обновленное значение? Существует несколько различных моделей согласованности памяти. *Последовательная согласованность* — это наиболее сильная модель. Она гарантирует, что обновления памяти будут происходить в некоторой последовательности, причем каждому процессору будет "видна" одна и та же последовательность. *Согласованность процессоров* — более слабая модель. Она обеспечивает, что записи в память, выполняемые каждым процессом, совершаются в том порядке, в котором их производит процессор, но записи, инициированные различными процессорами, для других процессоров могут выглядеть по-разному. Еще более слабая модель — *согласование освобождения*, при которой первичная память обновляется в указанных программистом точках синхронизации.

Проблема согласования памяти представляет противоречия между простотой программирования и расходами на реализацию. Программист интуитивно ожидает последовательного согласования, поскольку программа считывает и записывает переменные независимо от того, в какой части машины они хранятся в действительности. Когда процесс присваивает переменной значение, программист ожидает, что результаты этого присваивания станут немедленно известными всем процессам программы. С другой стороны, последовательное согласование очень дорого в реализации и замедляет работу машины. Дело в том, что при каждой записи аппаратная часть должна проверить все кэши (и, возможно, обновить их или сделать недействительными) и модифицировать первичную память. Вдобавок, эти действия должны быть *неделимыми*. Вот почему в мультипроцессорах обычно реализуется более слабая модель согласования памяти, а программистам необходимо вставлять инструкции синхронизации памяти. Это часто обеспечивают компиляторы и библиотеки, так что прикладной программист этим может не заниматься.

Как было отмечено, строки кэш-памяти часто содержат последовательности слов, которые блоками передаются из памяти и обратно. Предположим, что переменные $x$ и $y$ занимают по одному слову и хранятся в соседних ячейках памяти, отображенных в одну и ту же строку

кэш-памяти. Пусть некоторый процесс выполняется на процессоре 1 мультипроцессора и производит записи и чтения переменной *x*. Наконец, допустим, что еще один процесс, выполняемый на процессоре 2, считывает и записывает переменную *y*. Тогда при каждом обращении процессора 1 к переменной *x* строка кэш-памяти этого процессора будет содержать и копию переменной *y*. Аналогичная картина будет и в кэше процессора 2.

Ситуация, описанная выше, называется *ложным разделением*: процессы в действительности не разделяют переменные *x* и *y*, но аппаратная часть кэш-памяти интерпретирует обе переменные как один блок. Следовательно, когда процессор 1 обновляет переменную *x*, должна быть признана недействительной и обновиться строка кэша в процессоре 2, содержащая и *x*, и *y*. Таким же образом, когда процессор 2 обновляет значение *y*, строка кэш-памяти, содержащая значения *x* и *y*, в процессоре 1 тоже должна быть обновлена или признана недействительной. Эти операции замедляют работу системы памяти, поэтому программа будет выполняться намного медленнее, чем тогда, когда две переменные попадают в разные строки кэша. Главное здесь — чтобы программист ясно понимал, что процессы не разделяют переменные, когда фактически система памяти вынуждена обеспечивать их разделение и тратить время на управление им.

Чтобы избежать ложного разделения, программист должен гарантировать, что переменные, записываемые различными процессами, не будут находиться в смежных областях памяти. Одно из решений этой проблемы заключается в *выравнивании*, т.е. резервировании фиктивных переменных, которые просто занимают место и отделяют реальные переменные друг от друга. Это пример противоречия между временем и пространством: для сокращения времени выполнения приходится тратить пространство.

Итак, мультипроцессоры используют кэш-память для повышения производительности. Однако иерархичность памяти порождает проблемы согласованности кэша и памяти, а также возможность ложного разделения. Следовательно, для получения максимальной производительности на данной машине программист должен знать характеристики системы памяти и писать программы, учитывая их. К этим вопросам мы еще вернемся.

## 1.2.3. Мультикомпьютеры с распределенной памятью и сети

В *мультипроцессоре с распределенной памятью* тоже есть соединительная сеть, но каждый процессор имеет собственную память. Как показано на рис. 1.3, соединительная сеть и модули памяти меняются местами по сравнению с их положением в мультипроцессоре с разделяемой памятью. Соединительная сеть поддерживает *передачу сообщений*, а не чтение и запись в память. Следовательно, процессоры взаимодействуют, передавая и принимая сообщения. У каждого процессора есть свой кэш, но из-за отсутствия разделения памяти нет проблем согласованности кэша и памяти.

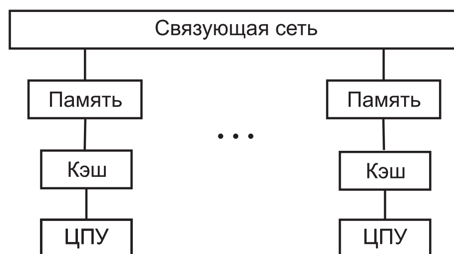


*Рис. 1.3. Структура машин с распределенной памятью*

*Мультикомпьютер* (*многомашинная система*) — это мультипроцессор с распределенной памятью, в котором процессоры и сеть расположены физически близко (в одном помещении). По этой причине такую многомашинную систему часто называют *тесно связанной* (*спаренной*) *машиной*. Она одновременно используется одним или небольшим количеством приложений; каждое приложение задействует выделенный набор процессоров. Соединительная сеть с большой пропускной способностью предоставляет высокоскоростной путь связи между процессорами. Обычно она организована в гиперкуб или матричную структуру. (Машины со структурой типа гиперкуб были одними из первых многомашинных систем.)

*Сетевая система* — это многомашинная система с распределенной памятью, узлы которой связаны с помощью локальной сети типа Ethernet или такой глобальной сети, как Internet. Сетевые системы называются *слабо связанными мультикомпьютерами*. Здесь процессоры взаимодействуют также с помощью передачи сообщений, но время их доставки больше, чем в многомашинных системах, и в сети больше конфликтов. С другой стороны, сетевая система строится на основе обычных рабочих станций и сетей, тогда как в многомашинной системе часто есть специализированные компоненты, особенно у связующей сети.

Сетевая система, состоящая из набора рабочих станций, часто называется *сетью рабочих станций* (network of workstations — NOW) или *кластером рабочих станций* (cluster of workstations — COW). Все рабочие станции выполняют одно или несколько приложений, возможно, связанных между собой. Популярный сейчас способ построения недорогого мультипроцессора с распределенной памятью — собрать так называемую *машину Беовулфа* (Beowulf). Она состоит из базового аппаратного обеспечения и таких бесплатных программ, как чипы к процессорам Pentium, сетевые карты, диски и операционная система Linux. (Имя Беовулф взято из старинной английской поэмы, первого шедевра английской литературы.)

Существуют также гибридные сочетания мультипроцессоров с распределенной и разделяемой памятью. Узлами системы с распределенной памятью могут быть мультипроцессоры с разделяемой памятью, а не обычные процессоры. Возможен вариант, когда связующая сеть поддерживает и механизмы передачи сообщений, и механизмы прямого доступа к удаленной памяти (на сегодня это наиболее мощные машины). Наиболее общая комбинация — машина с поддержкой *распределенной разделяемой памяти*, т.е. распределенной реализации абстракции разделяемой памяти. Она облегчает программирование многих приложений, но ставит проблемы согласованности кэша и памяти. (В главе 10 описана распределенная разделяемая память и ее реализация в программном обеспечении.)

# 1.3. Приложения и стили программирования

Параллельное программирование обеспечивает способ организации программного обеспечения, состоящего из относительно независимых частей. Оно также позволяет использовать множественные процессоры. Существует три обширных перекрывающихся класса приложений — многопоточные системы, распределенные системы и синхронные параллельные вычисления — и три соответствующих им типа параллельных программ.

Напомним, что процесс — это последовательная программа, которая при выполнении имеет собственный поток управления. Каждая параллельная программа содержит несколько процессов, поэтому имеет несколько потоков. Однако термин *многопоточный* обычно означает, что программа содержит больше процессов (потоков), чем существует процессоров для их выполнения. Следовательно, процессы на процессорах выполняются по очереди. Многопоточная программная система управляет множеством независимых процессов, например таких:

- оконные системы на персональных компьютерах или рабочих станциях;
- многопроцессорные операционные системы и системы с разделением времени;

- системы реального времени, управляющие электростанциями, космическими аппаратами и т.д.

Эти системы разработаны как многопоточные программы, поскольку организовать код и структуры данных в виде набора процессов намного проще, чем реализовать огромную последовательную программу. Кроме того, каждый процесс может планироваться и выполняться независимо. Например, когда пользователь нажимает кнопку мыши персонального компьютера, посылается сигнал процессу, управляющему окном, в котором в данный момент находится курсор мыши. Этот процесс (поток) может выполняться и отвечать на щелчок мыши. Приложения в других окнах могут продолжать при этом свое выполнение в фоновом режиме.

Второй широкий класс приложений образуют *распределенные вычисления*, в которых компоненты выполняются на машинах, связанных локальной или глобальной сетью. По этой причине процессы взаимодействуют, обмениваясь сообщениями. Вот примеры:

- файловые серверы в сети;
- системы баз данных для банков, заказа авиабилетов и т.д.;
- Web-серверы сети Internet;
- предпринимательские системы, объединяющие компоненты производства;
- отказоустойчивые системы, которые продолжают работать независимо от сбоев в компонентах.

Такие системы пишутся для распределения обработки (как в файловых серверах), обеспечения доступа к удаленным данным (как в базах данных и в Web), интеграции и управления данными, распределенными по своей сути (как в промышленных системах), или повышения надежности (как в отказоустойчивых системах). Многие распределенные системы организованы как системы типа клиент-сервер. Например, файловый сервер предоставляет файлы данных для процессов, выполняемых на клиентских машинах. Компоненты распределенных систем часто сами являются многопоточными программами.

Синхронные параллельные вычисления — третий широкий класс приложений. Их цель — быстро решить данную задачу или за то же время решить бóльшую задачу. Примеры синхронных вычислений:

- научные вычисления, которые моделируют и имитируют такие явления, как глобальный климат, эволюция солнечной системы или результат действия нового лекарства;
- графика и обработка изображений, включая создание спецэффектов в кино;
- крупные комбинаторные или оптимизационные задачи, например, планирование авиаперелетов или экономическое моделирование.

Программы решения таких задач требуют больших вычислительных мощностей. Для достижения высокой производительности они выполняются на параллельных процессорах, причем обычно количество процессов (потоков) равно числу процессоров. Параллельные вычисления описываются в виде *программ, параллельных по данным*, в которых все процессы выполняют одни и те же действия, но с собственной частью данных, или в виде *программ, параллельных по задачам*, в которых различные процессы решают различные задачи.

В данной книге рассмотрены все три вида приложений и, что более важно, показано, как их программировать. В многопоточных программах процессы (потоки) взаимодействуют, используя разделяемые переменные. В распределенных системах взаимодействие процессов обеспечивается с помощью обмена сообщениями или удаленного вызова операций. При выполнении синхронных параллельных вычислений процессы взаимодействуют, используя или разделяемые переменные, или передачу сообщений, в зависимости от аппаратного обеспечения, на котором выполняется программа. В части 1 этой книги показано, как написать про-

грамму, использующую для взаимодействия и синхронизации разделяемые переменные. Часть 2 описывает передачу сообщений и удаленные операции. В части 3 подробно рассмотрено синхронное параллельное программирование, ориентированное на научные вычисления.

Существует немало параллельных программных приложений, однако в них используется лишь небольшое число моделей решений, или *парадигм*. В частности, существует пять основных парадигм: 1) итеративный параллелизм, 2) рекурсивный параллелизм, 3) "производители и потребители" (конвейеры), 4) "клиенты и серверы", 5) взаимодействующие равные. С использованием одной или нескольких из этих парадигм и программируются приложения.

*Итеративный параллелизм* используется, когда в программе есть несколько процессов (часто идентичных), каждый из которых содержит один или несколько циклов. Таким образом, каждый процесс является итеративной программой. Процессы программы работают совместно над решением одной задачи; они взаимодействуют и синхронизируются с помощью разделяемых переменных или передачи сообщений. Итеративный параллелизм чаще всего встречается в научных вычислениях, выполняемых на нескольких процессорах.

*Рекурсивный параллелизм* может использоваться, когда в программе есть одна или несколько рекурсивных процедур, и их вызовы независимы, т.е. каждый из них работает над своей частью общих данных. Рекурсия часто применяется в императивных языках программирования, особенно при реализации алгоритмов типа "разделяй и властвуй" или "перебор с возвращением" (backtracking). Рекурсия является одной из фундаментальных парадигм и в символических, логических, функциональных языках программирования. Рекурсивный параллелизм используется для решения таких комбинаторных проблем, как сортировка, планирование (задача коммивояжера) и игры (шахматы и другие).

*Производители и потребители* — это взаимодействующие процессы. Они часто организуются в *конвейер*, через который проходит информация. Каждый процесс конвейера является *фильтром*, который потребляет выход своего предшественника и производит входные данные для своего последователя. Фильтры встречаются на уровне приложений (оболочки) в операционных системах типа ОС Unix, внутри самих операционных систем, внутри прикладных программ, если один процесс производит выходные данные, которые потребляет (читает) другой процесс.

*Клиенты и серверы* — наиболее распространенная модель взаимодействия в распределенных системах, от локальных сетей до World Wide Web. Клиентский процесс запрашивает сервис и ждет ответа. Сервер ожидает запросов от клиентов, а затем действует в соответствии с этими запросами. Сервер может быть реализован как одиночный процесс, который не может обрабатывать одновременно несколько клиентских запросов, или (при необходимости параллельного обслуживания запросов) как многопоточная программа. Клиенты и серверы представляют собой параллельное программное обобщение процедур и их вызовов: сервер выполняет роль процедуры, а клиенты ее вызывают. Но если код клиента и код сервера расположены на разных машинах, обычный вызов процедуры использовать нельзя. Вместо этого необходимо использовать *удаленный вызов процедуры* или *рандеву*, как это описано в главе 8.

*Взаимодействующие равные* — последняя парадигма взаимодействия. Она встречается в распределенных программах, в которых несколько процессов для решения задачи выполняют один и тот же код и обмениваются сообщениями. Взаимодействующие равные используются для реализации распределенных параллельных программ, особенно при итеративном параллелизме и децентрализованном принятии решений в распределенных системах. Некоторые приложения и схемы взаимодействия описаны в главе 9.

В следующих пяти разделах приводятся примеры, иллюстрирующие применение каждой модели. В примерах представлена программная нотация, используемая во всей книге. (Обзор

нотации дается в разделе 1.9.) Еще больше примеров приведено в тексте и упражнениях последующих глав.

# 1.4. Итеративный параллелизм: умножение матриц

Итеративная последовательная программа использует для обработки данных и вычисления результатов циклы типа `for` и `while`. Итеративная параллельная программа содержит несколько итеративных процессов. Каждый процесс вычисляет результаты для подмножества данных, а затем эти результаты собираются вместе.

В качестве простого примера рассмотрим задачу из области научных вычислений. Предположим, даны матрицы `a` и `b`, у каждой по `n` строк и столбцов, и обе инициализированы. Цель — вычислить произведение матриц, поместив результат в матрицу `c` размером `n`×`n`. Для этого нужно вычислить $n^2$ промежуточных произведений, по одному для каждой пары строк и столбцов.

Матрицы являются разделяемыми переменными, объявленными следующим образом.

```
double a[n,n], b[n,n], c[n,n];
```

При условии, что `n` уже объявлено и инициализировано, это объявление резервирует память для трех массивов действительных чисел двойной точности. По умолчанию индексы строк и столбцов изменяются от `0` до `n-1`.

После инициализации массивов `a` и `b` можно вычислить произведение матриц по такой последовательной программе.

```
for [i = 0 to n-1] {
  for [j = 0 to n-1] {
    # вычислить произведение a[i,*] и b[*,j]
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Внешние циклы (с индексами `i` и `j`) повторяются для каждой строки и столбца. Во внутреннем цикле (с индексом `k`) вычисляется промежуточное произведение строки `i` матрицы `a` и столбца `j` матрицы `b`; результат сохраняется в ячейке `c[i,j]`. Строка с символом # в начале является комментарием.

Умножение матриц — это пример *приложения с массовым параллелизмом*, поскольку программа содержит большое число операций, которые могут выполняться параллельно. Две операции могут выполняться параллельно, если они *независимы*. Предположим, что *множество чтения* операции содержит переменные, которые она читает, но не изменяет, а *множество записи* — переменные, которые она изменяет (и, возможно, читает). Две операции являются независимыми, если их множества записи не пересекаются. Говоря неформально, процессы всегда могут безопасно читать переменные, которые не изменяются. Однако двум процессам в общем случае небезопасно выполнять запись в одну и ту же переменную или одному процессу читать переменную, которая записывается другим. (Эта тема рассматривается подробно в главе 2.)

При умножении матриц вычисления промежуточных произведений являются независимыми операциями. В частности, строки с 4 по 6 приведенной выше программы выполняют инициализацию и последующее вычисление элемента матрицы `c`. Внутренний цикл программы читает строку матрицы `a` и столбец матрицы `b`, а затем читает и записывает один элемент матрицы `c`. Следовательно, множество чтения для внутреннего произведения — это строка матрицы `a` и столбец матрицы `b`, а множество записи — элемент матрицы `c`.

Поскольку множества записи внутренних произведений не пересекаются, их можно выполнять параллельно. Возможны варианты, когда параллельно вычисляются результирующие строки, результирующие столбцы или группы строк и столбцов. Ниже будет показано, как запрограммировать такие параллельные вычисления.

Сначала рассмотрим параллельное вычисление строк матрицы c. Его можно запрограммировать с помощью оператора co (от "concurrent" — "параллельный"):

```
co [i = 0 to n-1] { # параллельное вычисление строк
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
   }
}
```

Между этой программой и ее последовательным вариантом есть лишь *одно* синтаксическое различие — во внешнем цикле оператор for заменен оператором co. Но семантическая разница велика: оператор co определяет, что его тело для каждого значения индекса i будет выполняться *параллельно* (если не в действительности, то, по крайней мере, теоретически, что зависит от числа процессоров).

Другой способ выполнения параллельного умножения матриц состоит в параллельном вычислении столбцов матрицы c. Его можно запрограммировать следующим образом.

```
co [j = 0 to n-1] { #параллельное вычисление столбцов
  for [i = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь два внешних цикла (по i и по j) *поменялись местами*. Если тела двух циклов независимы и приводят к вычислению одинаковых результатов, их можно безопасно менять местами, как это было сделано здесь. (Это и другие аналогичные преобразования программ рассматриваются в главе 12.)

Программу для параллельного вычисления всех промежуточных произведений можно составить несколькими способами. Используем один оператор co для двух индексов.

```
co [i = 0 to n-1, j = 0 to n-1] { # все строки и
  c[i,j] = 0.0;                    # все столбцы
  for [k = 0 to n-1]
    c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Тело оператора co выполняется параллельно для каждой комбинации значений индексов i и j, поэтому программа задает $n^2$ процессов. (Будут ли они выполняться параллельно, зависит от конкретной реализации.) Другой способ параллельного вычисления промежуточных произведений состоит в использовании вложенных операторов co.

```
co [i = 0 to n-1] {   # строки параллельно, затем
  co [j = 0 to n-1] { # столбцы параллельно
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь для каждой строки (внешний оператор co) и затем для каждого столбца (внутренний оператор co) задается по одному процессу. Третий способ написать эту программу — поменять местами две строки последней программы. Результат всех трех программ будет одинаковым: выполнение внутреннего цикла для всех $n^2$ комбинаций значений i и j. Разница между ними — в задании процессов, а значит, и во времени их создания.

Заметим, что все параллельные программы, приведенные выше, были получены заменой оператора for на co. Но мы сделали это только для индексов i и j. А как быть с внутренним циклом по индексу k? Нельзя ли и этот оператор заменить оператором co? Ответ — "нет", поскольку тело внутреннего цикла как читает, так и записывает переменную c[i,j]. Промежуточное произведение — цикл for с переменной k — можно вычислить, используя двоичный параллелизм, но для большинства машин это непрактично (см. упражнения в конце главы).

Другой способ определить параллельные вычисления, показанные выше, — использовать декларацию (объявление) process вместо оператора co. В сущности, process — это оператор co, выполняемый как "фоновый". Например, первая параллельная программа из показанных выше — та, что параллельно вычисляет строки результата, — может быть записана следующим образом.

```
process row[i = 0 to n-1] { # строки параллельно
  for [j = 0 to n-1] {
    c[i,j] = 0.0;
    for [k = 0 to n-1]
      c[i,j] = c[i,j] + a[i,k]*b[k,j];
  }
}
```

Здесь определен массив процессов — row[1], row[2] и т.д. — по одному для каждого значения индекса i. Эти n процессов создаются и начинают выполняться, когда встречается данная строка описания. Если за декларацией процесса следуют операторы, то они выполняются параллельно с процессом, тогда как операторы, записанные после оператора co, не выполняются до его завершения. Декларации процесса, в отличие от операторов co, не могут быть вложены в другие декларации или операторы. Декларации процессов и операторы co подробно описаны в разделе 1.9.

В программах, приведенных выше, для каждого элемента, строки или столбца результирующей матрицы использовано по одному процессу. Предположим, что число процессоров в системе меньше n (так обычно и бывает, особенно когда n велико). Остается еще очевидный способ полного использования всех процессоров: разделить матрицу на полосы (строк или столбцов) и для каждой полосы создать *рабочий* процесс. В частности, каждый рабочий процесс вычисляет результаты для элементов своей полосы. Предположим, что есть P процессоров и n кратно P (т.е. n делится на P без остатка). Тогда при использовании полос строк рабочие процессы можно запрограммировать следующим образом.

```
process worker[w = 1 to P] {  # полосы параллельно
  int first = (w-1) * n/P;    # первая строка полосы
  int last = first + n/P - 1; # последняя строка полосы
  for [i = first to last] {
    for [j = 0 to n-1] {
      c[i,j] = 0.0;
      for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
  }
}
```

Отличие этой программы от предыдущей состоит в том, что n строк делятся на P полос, по n/P строк каждая. Для этого в программу добавлены операторы, необходимые для определения первой и последней строки каждой полосы. Затем строки полосы указываются в цикле (по индексу i), чтобы вычислить промежуточные произведения для этих строк.

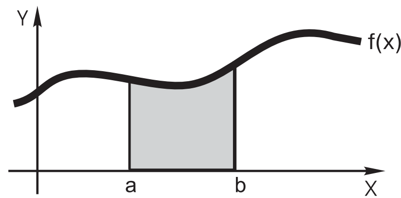


*Рис. 1.4. Задача квадратуры*

Итак, существенным условием распараллеливания программы является наличие независимых вычислений, т.е. вычислений с непересекающимися множествами записи. Для произведения матриц независимыми вычислениями являются промежуточные произведения, поскольку каждое из них записывает (и читает) свой элемент c[i,j] результирующей матрицы. Поэтому можно параллельно вычислять все промежуточные произведения, строки, столбцы или полосы строк. И, наконец, параллельные программы можно записывать, используя операторы co или обьявления process.

# 1.5. Рекурсивный параллелизм: адаптивная квадратура

Программа считается рекурсивной, если она содержит процедуры, которые вызывают сами себя — прямо или косвенно. Рекурсия дуальна итерации в том смысле, что рекурсивные программы можно преобразовать в итеративные и наоборот. Однако каждый стиль программирования имеет свое применение, поскольку одни задачи по своей природе рекурсивны, а другие — итеративны.

В теле многих рекурсивных процедур обращение к самой себе встречается больше одного раза. Например, алгоритм *quicksort* часто используется для сортировки. Он разбивает массив на две части, а затем дважды вызывает себя: первый раз для сортировки левой части, а второй — для правой. Многие алгоритмы для работы с деревьями и графами имеют подобную структуру.

Рекурсивную программу можно реализовать с помощью параллелизма, если она содержит несколько независимых рекурсивных вызовов. Два вызова процедуры (или функции) являются независимыми, если их множества записи не пересекаются. Это условие выполняется, если: 1) процедура не обращается к глобальным переменным или только читает их; 2) аргументы и результирующие переменные (если есть) являются различными переменными. Например, если процедура не обращается к глобальным переменным и имеет только параметры-значения, то любой ее вызов будет независимым. (Хорошо, если процедура читает и записывает только локальные переменные, тогда каждый экземпляр процедур имеет локальную копию переменных.) В соответствии с этими требованиями можно запрограммировать и алгоритм быстрой сортировки. Рассмотрим еще один интересный пример.

*Задача квадратуры* состоит в аппроксимации интеграла непрерывной функции. Предположим, что это функция f(x). Как показано на рис. 1.4, интеграл функции f(x) от a до b — это площадь между графиком f(x) и осью абсцисс от прямой x = a до прямой x = b.

Существует два основных способа аппроксимации значения интеграла. Первый — разделить интервал от a до b на фиксированное число отрезков, а затем аппроксимировать площадь на каждом из них по правилу трапеций или по правилу Симпсона.

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
```

```
    area = area + (fleft + fright) * width / 2;
    fleft = fright;
  }
```

Каждая итерация вычисляет площадь малой фигуры по правилу трапеций и добавляет ее к общему значению площади. Переменная width — ширина каждой трапеции. Отрезки перебираются слева направо, поэтому правое значение каждой итерации становится левым значением следующей итерации.

Второй способ аппроксимации интеграла — использовать парадигму "разделяй и властвуй" и переменное число интервалов. В частности, сначала вычисляют значение m — середину отрезка между a и b. Затем аппроксимируют площадь трех областей под кривой, определенной функцией f(): от a до m, от m до b и от a до b. Если сумма меньших площадей равна большей площади с некоторой заданной точностью EPSILON, то аппроксимацию можно считать достаточной. Если нет, то большая задача — от a до b — делится на две подзадачи — от a до m и от m до b, и процесс повторяется. Этот способ называется *адаптивной квадратурой*, поскольку алгоритм адаптируется к форме кривой. Его можно запрограммировать так.

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # рекурсия для интегрирования обоих значений
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

Интеграл функции f(x) от a до b аппроксимируется таким вызовом функции:

```
area = quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
```

В функции снова используется правило трапеции. Значения функции f() в крайних точках отрезка и приближенная площадь этого интервала передаются в каждый вызов функции quad, чтобы не вычислять их больше одного раза.

Итеративную программу нельзя распараллелить, поскольку тело цикла и считывает, и записывает значение переменной area. Тем не менее в рекурсивной программе вызовы функции quad независимы при условии, что вычисление функции f(x) не дает побочных эффектов. В частности, аргументы функции quad передаются по значению, и в ее теле нет присваивания глобальным переменным. Таким образом, для задания параллельного выполнения рекурсивных вызовов функции можно использовать оператор co.

```
co larea = quad(left, mid, fleft, fmid, larea);
// rarea = quad(mid, right, fmid, fright, rarea);
oc
```

Это единственное изменение, необходимое для того, чтобы сделать данную программу рекурсивной. Поскольку оператор co не заканчивается до тех пор, пока не будут завершены оба вызова функций, значения переменных larea и rarea вычисляются до того, как функция quad возвратит их сумму.

В операторах co программ умножения матриц содержатся списки инструкций, выполняемых для каждого значения счетчиков (i и j). В операторе co, приведенном выше, содер-

жатся два вызова функций, разделенных знаками //. Первая форма оператора co используется для выражения итеративного параллелизма, вторая — рекурсивного.

Итак, программу с несколькими рекурсивными вызовами функций можно легко преобразовать в параллельную рекурсивную программу, если вызовы независимы. Однако существует чисто практическая проблема: параллельно выполняемых операций может стать слишком много. Каждый оператор co в приведенной выше программе создает два процесса, по одному для каждого вызова функции. Если глубина рекурсии велика, то появится большое число процессов, возможно, слишком большое для параллельного выполнения. Решение этой проблемы состоит в *сокращении*, или *отсечении*, дерева рекурсии при достаточном количестве процессов, т.е. переключении с параллельных рекурсивных вызовов на последовательные. Эта тема рассматривается в упражнениях и далее в этой книге.

# 1.6. Производители и потребители: каналы ОС Unix

Процесс-производитель выполняет вычисления и выводит поток результатов. Процесс-потребитель вводит и анализирует поток значений. Многие программы в той или иной форме являются производителями и/или потребителями. Сочетание становится особенно интересным, если производители и потребители объединены в *конвейер* — последовательность процессов, в которой каждый из них потребляет данные выхода предшественника и производит данные для последующего процесса. Классическим примером являются конвейеры в операционной системе Unix, рассматриваемые здесь. Другие примеры приводятся в последующих главах.

Обычно прикладной процесс в ОС Unix считывает данные из *стандартного файла ввода* stdin и записывает в *стандартный файл вывода* stdout. Обычно файл ввода — это клавиатура терминала, с которого вызвано приложение, а файл вывода — дисплей этого терминала. Но одной из наиболее мощных функций, предложенных в ОС Unix, была возможность привязки стандартных "устройств" ввода-вывода к различным типам файлов. В частности, файлы stdin и/или stdout могут быть связаны с файлом данных или с "файлом" особого типа, который называется каналом.

*Канал* — это буфер (очередь типа FIFO, работающая по принципу "First in — first out", т.е. "первым вошел, первым вышел") между процессом-производителем и процессом-потребителем. Он содержит связанную последовательность символов. Новые значения дописываются к ней, когда производитель выполняет запись в канал. Символы удаляются, когда процесс-потребитель считывает их из канала.

Прикладная программа в ОС Unix только читает данные из файла stdin, не заботясь о том, откуда в действительности они туда попали. Если файл stdin связан с клавиатурой, на вход поступают символы, набранные на клавиатуре. Если файл stdin связан с определенным файлом, вводится последовательность символов из этого файла. Если файл stdin связан с каналом, то вводится последовательность символов, записанных в этот канал. Аналогично приложение выполняет запись в файл stdout, не заботясь о том, куда в действительности поступают данные.

Каналы ОС Unix обычно определяются с помощью одного из командных языков, например csh (C shell — "оболочка C"). В частности, печатные страницы оригинала этой книги создавались с помощью команды на языке csh, похожей на следующую:

```
sed -f Script $* | tbl |eqn | groff Macros -
```

Этот конвейер содержит четыре команды: 1) sed, потоковый текстовый редактор; 2) tbl, процессор таблиц; 3) eqn, процессор уравнений и 4) groff, программа, создающая данные

в формате Postscript из исходных файлов в формате `troff`. Каждая пара команд разделена вертикальной чертой, обозначающей канал в C shell.

На рис. 1.5 показана структура этого конвейера. Каждая команда является процессом-*фильтром*. Вход фильтра `sed` образован файлом редактирующих команд (`Script`) и аргументами командной строки (`$*`), которыми в данном случае являются соответствующие исходные файлы текста книги. Выход редактора `sed` передается программе `tbl`, направляющей свои выходные данные программе `eqn`, а та передает свой выход программе `groff`. Фильтр `groff` читает файл `Macros` для этой книги, считывает и обрабатывает свой стандартный вход, а затем отсылает выход на принтер в офисе автора.

*Рис. 1.5. Конвейер процессов*

Каждый поток на рис. 1.5 реализован связанным буфером: синхронизированной очередью значений типа FIFO. Процесс-производитель ожидает (при необходимости), пока в буфере появится свободное место, затем добавляет в конец буфера новую строку. Процесс-потребитель ожидает (при необходимости), пока в буфере не появится строка данных, затем забирает ее. В части 1 показано, как реализовать такие буферы с использованием разделяемых переменных и различных примитивов синхронизации (флагов, семафоров и мониторов). В части 2 представлены каналы взаимодействия и примитивы пересылки сообщений `send` (отослать) и `receive` (получить). Затем будет показано, как с их использованием программируются фильтры, а с помощью буферов реализуются каналы и передача сообщений.

# 1.7. Клиенты и серверы: файловые системы

Между производителем и потребителем существует однонаправленный поток информации. Этот вид межпроцессного взаимодействия часто встречается в параллельных программах и не имеет аналогов в последовательных, поскольку в последовательной программе только один поток управления, тогда как производители и потребители — независимые процессы с собственными потоками управления и собственными скоростями выполнения.

Еще одной типичной схемой в параллельных программах является взаимосвязь типа клиент-сервер. Процесс-*клиент* запрашивает сервис, затем ожидает обработки запроса. Процесс-*сервер* многократно ожидает запрос, обрабатывает его, затем посылает ответ. Как показано на рис. 1.6, существует двунаправленный поток информации: от клиента к серверу и обратно. Отношения между клиентом и сервером в параллельном программировании аналогичны отношениям между программой, вызывающей подпрограмму, и самой подпрограммой в последовательном программировании. Более того, как подпрограмма может быть вызвана из нескольких мест программы, так и у сервера обычно есть много клиентов. Запросы каждого клиента должны обрабатываться независимо, однако параллельно может обрабатываться несколько запросов, подобно тому, как одновременно могут быть активны несколько вызовов одной и той же процедуры.

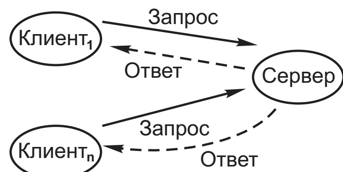


*Рис. 1.6. Клиенты и серверы*

Взаимодействие типа клиент-сервер встречается в операционных системах, объектно-ориентированных системах, сетях, базах данных и многих других программах. Типичный пример — чтение и запись файла. Для определенности предположим, что есть модуль файлового сервера, обеспечивающий две операции с файлом: `read` (читать) и `write` (писать). Когда процесс-клиент хочет получить доступ к

файлу, он вызывает операцию чтения или записи в соответствующем модуле файлового сервера.

На однопроцессорной машине или в другой системе с разделяемой памятью файловый сервер обычно реализуется набором подпрограмм (для операций `read`, `write` и т.д.) и структурами данных, представляющими файлы (например, дескрипторами файлов). Следовательно, взаимодействие между процессом-клиентом и файлом обычно реализуется вызовом соответствующей процедуры. Однако, если файл разделяемый, важно, чтобы запись в него велась одновременно только одним процессом, а читаться он может одновременно несколькими. Эта разновидность задачи — пример так называемой задачи о “читателях и писателях”, классической задачи параллельного программирования, которая ставится и решается в главе 4, а также упоминается в последующих главах.

В распределенной системе клиенты и серверы обычно расположены на различных машинах. Например, рассмотрим запрос по World Wide Web, который возникает, когда пользователь открывает новый адрес URL в окне программы-броузера. Web-броузер является клиентским процессом, выполняемым на машине пользователя. Адрес URL косвенно указывает на другую машину, на которой расположена Web-страница. Сама Web-страница доступна для процесса-сервера, выполняемого на другой машине. Этот процесс-сервер может уже существовать или может быть создан; в любом случае он читает Web-страницу, определяемую адресом URL, и возвращает ее на машину клиента. В действительности при преобразовании адреса URL могут использоваться или создаваться дополнительные процессы на промежуточных машинах по пути следования.

Клиенты и серверы программируются одним из двух способов в зависимости от того, выполняются они на одной или на разных машинах. В обоих случаях клиенты являются процессами. На машине с разделяемой памятью сервер обычно реализуется набором подпрограмм. Для защиты критических секций и определения очередности выполнения эти подпрограммы обычно реализуются с использованием взаимных исключений и условной синхронизации. На сетевых машинах или машинах с распределенной памятью сервер реализуется одним или несколькими процессами, которые обычно выполняются не на клиентских машинах. В обоих случаях сервер часто представляет собой многопоточную программу с одним потоком для каждого клиента.

В частях 1 и 2 представлены многочисленные приложения типа клиент-сервер, включая файловые системы, базы данных, программы резервирования памяти, управления диском, а также две классические задачи — об “обедающих философах” и о “спящем парикмахере”. В части 1 показано, как реализовать серверы в виде подпрограмм, используя для синхронизации семафоры или мониторы. В части 2 — как реализовать серверы в виде процессов, взаимодействующих с клиентами с помощью пересылки сообщений, удаленных вызовов процедур или рандеву.

## 1.8. Взаимодействующие равные: распределенное умножение матриц

Ранее было показано, как реализовать параллельное умножение матриц с помощью процессов, разделяющих переменные. Здесь представлены два способа решения этой задачи с использованием процессов, взаимодействующих с помощью пересылки сообщений. Более сложные алгоритмы представлены в главе 9. Первая программа использует управляющий процесс и массив независимых рабочих процессов. Во второй программе рабочие процессы равны и их взаимодействие обеспечивается круговым конвейером. Рис. 1.7 иллюстрирует структуру схем взаимодействия этих процессов. Как показано в части 2, они часто встречаются в распределенных параллельных вычислениях.

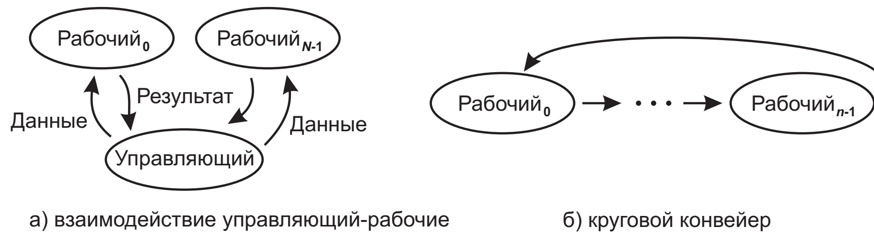


*Рис. 1.7. Умножение матриц с использованием передачи сообщений*

На машинах с распределенной памятью каждый процессор имеет доступ только к собственной локальной памяти. Это значит, что программа не может использовать глобальные переменные, поэтому любая переменная должна быть локальной для некоторого процесса и может быть доступной только этому процессу или процедуре. Следовательно, для взаимодействия процессы должны использовать передачу сообщений.

Допустим, что нам необходимо получить произведение матриц a и b, а результат поместить в матрицу c. Предположим, что каждая из них имеет размер n×n и существует n процессоров. Можно использовать массив из n рабочих процессов, поместив по одному на каждый процессор и заставив каждый рабочий процесс вычислять одну строку результирующей матрицы c. Программа для рабочих процессов будет выглядеть следующим образом.

```
process worker[i = 0 to n-1] {
  double a[n];   # строка i матрицы a
  double b[n,n]; # вся матрица b
  double c[n];   # строка i матрицы c
  receive начальные значения вектора a и матрицы b;
  for [j = 0 to n-1] {
    c[j] = 0.0;
    for [k = 0 to n-1]
      c[j] = c[j] + a[k] * b[k,j];
  }
  send вектор-результат c управляющему процессу;
}
```

Рабочий процесс i вычисляет строку i результирующей матрицы c. Чтобы это сделать, он должен получить строку i исходной матрицы a и всю исходную матрицу b. Каждый рабочий процесс сначала получает эти значения от отдельного управляющего процесса. Затем рабочий процесс вычисляет свою строку результатов и отсылает ее обратно управляющему. (Или, возможно, исходные матрицы являются результатом предшествующих вычислений, а результирующая — входом для последующих; это пример распределенного конвейера.)

Управляющий процесс инициирует вычисления, собирает и выводит их результаты. В частности, сначала управляющий процесс посылает каждому рабочему соответствующую строку матрицы a и всю матрицу b. Затем управляющий процесс ожидает получения строк матрицы c от каждого рабочего. Схема управляющего процесса такова.

```
process coordinator {
  double a[n,n]; # исходная матрица a
  double b[n,n]; # исходная матрица b
  double c[n,n]; # результирующая матрица c
  инициализировать a и b;
  for [i = 0 to n-1] {
    send строку i матрицы a процессу worker[i];
    send всю матрицу b процессу worker[i];
  }
```

```
    for [i = 0 to n-1]
      receive строку i матрицы c от процесса worker[i];
    вывести результат, который теперь в матрице c;
  }
```

Операторы send и receive, используемые управляющим процессом, — это *примитивы* (*элементарные действия*) *передачи сообщений*. Операция send упаковывает сообщение и пересылает его другому процессу; операция receive ожидает сообщение от другого процесса, а затем сохраняет его в локальных переменных. Подробно передача сообщений будет описана в главе 7 и использована в программировании многочисленных приложений в частях 2 и 3.

Как и ранее, предположим, что каждый рабочий процесс получает одну строку матрицы a и должен вычислить одну строку матрицы c. Однако теперь допустим, что у каждого процесса есть только один столбец, а не вся матрица b. Итак, в начальном состоянии рабочий процесс i имеет столбец i матрицы b. Имея лишь эти исходные данные, рабочий процесс может вычислить только значение c[i,i]. Для того чтобы рабочий процесс i мог вычислить всю строку матрицы c, он должен получить все столбцы матрицы b. Для этого можно использовать круговой конвейер (см. рис. 1.7, *б*), в котором по рабочим процессам циркулируют столбцы. Каждый рабочий процесс выполняет последовательность *раундов*; в каждом раунде он отсылает свой столбец матрицы b следующему процессу и получает другой ее столбец от предыдущего. Программа имеет следующий вид.

```
process worker[i = 0 to n-1] {
  double a[n];   # строка i матрицы a
  double b[n];   # один столбец матрицы b
  double c[n];   # строка i матрицы c
  double sum = 0.0; # для промежуточных произведений
  int nextCol = i;  # следующий столбец результатов
  receive строку i матрицы a и столбец i матрицы b;
  # вычислить c[i,i] = a[i,*] × b[*,i]
  for [k = 0 to n-1]
    sum = sum + a[k] * b[k];
  c[nextCol] = sum;
  #пустить по кругу столбцы и вычислить остальные c[i,*]
  for [j = 1 to n-1] {
    send мой столбец матрицы b следующему процессу;
    receive новый столбец матрицы b от предыдущего;
    sum = 0.0;
    for [k = 0 to n-1]
      sum = sum + a[k]*b[k];
    if (nextCol == 0)
      nextCol = n-1;
    else
      nextCol = nextCol-1;
    c[nextCol] = sum;
  }
  send вектор-результат c управляющему процессу;
}
```

В данной программе рабочие процессы упорядочены в соответствии с их индексами. (Для процесса n-1 следующим является процесс 0, а предыдущим для 0 — n-1.) Столбцы матрицы b передаются по кругу между рабочими процессами, поэтому каждый процесс в конце концов получит каждый столбец. Переменная nextCol отслеживает, куда в векторе c поместить очередное промежуточное произведение. Как и в первом вычислении, предполагается, что

управляющий процесс отправляет строки матрицы a и столбцы матрицы b рабочим, а затем получает от них строки матрицы c.

Во второй программе использовано отношение между процессорами, которое называется *взаимодействующие равные* (interacting peers), или просто *равные*. Каждый рабочий процесс выполняет один и тот же алгоритм и взаимодействует с другими рабочими, чтобы вычислить свою часть необходимого результата. Дальнейшие примеры взаимодействующих равных мы увидим в частях 2 и 3. В одних случаях, как и здесь, каждый из рабочих процессов общается только с двумя своими соседями, в других — каждый из рабочих взаимодействует со всеми остальными процессами.

В первой из приведенных выше программ значения из матрицы b *дублируются* в каждом рабочем процессе. Во второй программе в любой момент времени у каждого процесса есть одна строка матрицы a и только один столбец матрицы b. Это снижает затраты памяти для каждого процесса, но вторая программа выполняется дольше первой, поскольку на каждой ее итерации каждый рабочий процесс должен отослать сообщение одному соседу и получить сообщение от другого. Данные программы иллюстрируют классическое противоречие между временем и пространством в вычислениях. В разделе 9.3 представлены другие алгоритмы для распределенного умножения матриц, иллюстрирующие дополнительные противоречия между временем и пространством.

# 1.9. Обзор программной нотации

В пяти предыдущих разделах были представлены примеры циклических схем в параллельном программировании: итеративный параллелизм, рекурсивный параллелизм, производители и потребители, клиенты и серверы, а также взаимодействующие равные. Многочисленные примеры этих схем еще будут приведены. В примерах также была введена программная нотация. В данном разделе дается ее обзор, хотя она очевидна из примеров.

Напомним, что параллельная программа содержит один или несколько процессов, а каждый процесс — это последовательная программа. Таким образом, наш язык программирования содержит механизмы и параллельного, и последовательного программирования. Нотация последовательных программ основана на базовых понятиях языков C, C++ и Java. В нотации параллельного программирования используются операторы co и декларации process. Они были представлены ранее и определяются ниже. В следующих главах будут определены механизмы синхронизации и межпроцессного взаимодействия.

## 1.9.1. Декларации

*Декларация* (*объявление*, или *определение*) переменной задает тип данных и перечисляет имена одной или нескольких переменных этого типа. При объявлении переменную можно инициализировать, например:

```
int i, j = 3;
double sum = 0.0;
```

Массив объявляется добавлением размера каждого измерения к имени массива. Диапазон индексов массива по умолчанию находится в пределах от 0 до значения, меньшего на 1, чем размер измерения. В качестве альтернативы можно непосредственно указать нижнюю и верхнюю границы диапазона. Массивы также можно инициализировать при их объявлении. Вот примеры:

```
int a[n];   # то же, что и "int a[0:n-1];"
int b[1:n]; # массив из n целых, b[1] ... b[n]
```

```
int c[1:n]=([n]0);  # вектор нулей
double c[n,n] = ([n] ([n] 1.0)); # матрица единиц
```

Каждая декларация сопровождается комментарием, который начинается знаком # (см. раздел 1.9.4). Последняя декларация говорит, что `c` — это матрица чисел двойной точности. Индексы каждого ее измерения находятся в пределах от `0` до `n-1`, а начальное значение каждого ее элемента — `1.0`.

## 1.9.2. Последовательные операторы

В операторе присваивания есть целевая переменная (слева), знак равенства и выражение (справа). Для операций инкремента (увеличения), декремента (уменьшения) и других присваиваний используется короткая форма этого оператора. Примеры:

```
a[n] = a[n] + 1;  # то же, что "a[n]++;"
x = (y+z) * f(x); # f(x) - вызов функции
```

Из управляющих операторов используются `if`, `while` и `for`. Простой оператор `if` имеет такой вид.

```
if (условие)
  оператор;
```

Здесь `условие` — это булево выражение (со значением "истина" или "ложь"), а `оператор` — одиночный оператор. Если при истинности условия нужно выполнить несколько операторов, они отделяются фигурными скобками.

```
if (условие) {
  оператор1;
  ...
  операторN;
}
```

В последующих главах такой список операторов часто обозначается через `S`. Оператор `if/then/else` имеет такой вид.

```
if (условие)
  оператор1;
else
  оператор2;
```

Операторы здесь также могут представлять собой списки операторов в фигурных скобках.

Оператор `while` имеет следующий общий вид.

```
while (условие) {
  оператор1;
  ...
  операторN;
}[1]
```

Если `условие` имеет значение "истина", то выполняются вложенные операторы (тело цикла), а затем оператор `while` повторяется. Цикл `while` завершается, если `условие` имеет значение "ложь". Если в теле цикла только один оператор, фигурные скобки опускаются.

Операторы `if` и `while` идентичны соответствующим операторам в языках C, C++ и Java, но оператор `for` записывается более компактно. Его общий вид таков.

---

[1] В действительности его общий вид — `while (условие) оператор;`. Это же относится и к следующему оператору `for`. — *Прим. перев.*

```
for [квантификатор1, ..., квантификаторM] {
  оператор1;
  ...
  операторN;
}
```

Каждый *квантификатор* вводит новую индексную переменную (параметр цикла), инициализирует ее и указывает диапазон ее значений. Квадратные скобки вокруг квантификаторов используются для определения диапазона значений, как и в декларациях массивов.

Предположим, что `a[n]` — это массив целых чисел. Тогда следующий оператор инициализирует каждый элемент массива `a[i]` значением `i`.

```
for [i = 0 to n-1]
  a[i] = i;
```

Здесь `i` — новая переменная; ее не обязательно определять выше в программе. Область видимости переменной `i` — тело данного цикла `for`. Ее начальное значение `0`, и она принимает по порядку значения от `0` до `n-1`.

Предположим, что `m[n,n]` — массив целых чисел. Рассмотрим оператор `for` с двумя квантификаторами.

```
for [i = 0 to n-1, j = 0 to n-1]
  m[i,j] = 0;
```

Этому оператору эквивалентны вложенные операторы `for`.

```
for [i = 0 to n-1]
  for [j = 0 to n-1]
    m[i,j] = 0;
```

В обоих случаях $n^2$ значений матрицы `m` инициализируются нулями. Рассмотрим еще два примера квантификаторов.

```
[i = 1 to n by 2]      #нечетные значения от 1 до n
[i = 0 to n-1 st i!=x] #каждое значение, кроме i==x
```

Обозначение `st` во втором квантификаторе — это сокращение слов "such that" ("такой, для которого").

Операторы `for` записываются с использованием синтаксиса, приведенного выше, по нескольким причинам. Во-первых, этим подчеркивается отличие наших операторов `for` от тех же операторов в языках C, C++ и Java. Во-вторых, такая нотация предполагает их использование с массивами, у которых индексы заключаются в квадратные, а не круглые скобки. В-третьих, наша запись упрощает программы, поскольку избавляет от необходимости объявлять индексную переменную. (Сколько раз вы забывали это сделать?) В-четвертых, зачастую удобнее использовать несколько индексных переменных, т.е. записывать несколько квантификаторов. И, наконец, те же формы квантификаторов используются в операторах `co` и декларациях `process`.

## 1.9.3. Параллельные операторы, процессы и процедуры

По умолчанию операторы выполняются последовательно, т.е. один за другим. Оператор `co` (concurrent — параллельный, происходящий одновременно) указывает, что несколько операторов могут выполняться параллельно. В одной форме оператор `co` имеет несколько *ветвей* (arms).

```
co оператор1;
// ...
```

```
// операторN;
oc
```

Каждая ветвь содержит оператор (или список операторов). Ветви отделяются символом параллелизма //. Оператор, приведенный выше, означает следующее: начать параллельное выполнение всех операторов, затем ожидать их завершения. Оператор co, таким образом, завершается после выполнения всех его операторов.

В другой форме оператор co использует один или несколько квантификаторов, которые указывают, что набор операторов должен выполняться параллельно для каждой комбинации значений параметров цикла. Например, следующий тривиальный оператор инициализирует массивы a[n] и b[n] нулями.

```
co[i = 0 to n-1] {
  a[i] = 0; b[i] = 0;
}
```

Этот оператор создает n процессов, по одному для каждого значения переменной i. Область видимости счетчика — описание процесса, и у каждого процесса свое, отличное от других, значение переменной i. Две формы оператора co можно смешивать. Например, одна ветвь может иметь квантификатор в квадратных скобках, а другая — нет.

Декларация процесса является, по существу, сокращенной формой оператора co с одной ветвью и/или одним квантификатором. Она начинается ключевым словом process и именем процесса, а заканчивается ключевым словом end. Тело процесса содержит определения локальных переменных, если такие есть, и список операторов.

В следующем простом примере определяется процесс foo, который суммирует числа от 1 до 10, записывая результат в глобальную переменную x.

```
process foo {
  int sum = 0;
  for [i = 1 to 10]
    sum += i;
  x = sum;
}
```

Декларация process записывается на синтаксическом уровне декларации procedure; это не оператор, в отличие от co. Кроме того, объявляемые процессы выполняются в фоновом режиме, тогда как выполнение оператора, следующего за оператором co, начинается после завершения процессов, созданных этим co.

Еще один простой пример: следующий процесс записывает значения от 1 до n в стандартный файл вывода.

```
process bar1 {
  for [i = 1 to n]
    write(i);  # то же, что "printf("%d\n",i);"
}
```

Массив процессов объявляется добавлением квантификатора (в квадратных скобках) к имени процесса.

```
process bar2[i = 1 to n] {
  write(i);
}
```

И bar1, и bar2 записывают в стандартный вывод значения от 1 до n. Однако порядок, в котором их записывает массив процессов bar2, *недетерминирован*, поскольку массив bar2 состоит из n отдельных процессов, выполняемых в произвольном порядке. Существует n! различных порядков, в которых этот массив процессов мог бы записать числа (n! — число перестановок n значений).

Процедуры и функции объявляются и вызываются так же, как это делается в языке C, например, так.

```
int addOne(int v) { # функция возвращает целое число
  return (v + 1);
}
main() {  # "void"-процедура
  int n, sum;
  read(n);  # прочитать целое число из stdin
  for [i = 1 to n]
    sum = sum + addOne(i);
  write("Окончательным значением является ", sum);
}
```

Если входное значение n равно 5, эта программа выведет такую строку.

```
Окончательным значением является 20
```

### 1.9.4. Комментарии

Комментарии записываются двумя способами. Однострочные комментарии начинаются символом # и завершаются в конце строки. Многострочные комментарии начинаются символами /* и оканчиваются символами */. Для однострочных комментариев используется символ #, поскольку символ однострочных комментариев // языков C++ и Java уже давно использовался в параллельном программировании как разделитель ветвей в параллельных операторах.

*Утверждение* — это предикат, определяющий условие, которое должно выполняться в некоторой точке программы. (Утверждения подробно описаны в главе 2.) Утверждения можно рассматривать как предельно точные комментарии, поэтому они записываются в отдельных строках, начинающихся двумя символами #:

```
## x > 0
```

Данный комментарий утверждает, что значение x положительно.

## Историческая справка

Как уже отмечалось, параллельное программирование возникло в 1960-х годах после появления независимых аппаратных контроллеров (каналов). Операционные системы были первыми программными системами, организованными как многопоточные параллельные программы. Исследование и первоначальные прототипы привели на рубеже 1960-х и 1970-х годов к современным операционным системам. Первые книги по операционным системам появились в начале 1970-х годов.

Создание компьютерных сетей привело в 1970-х годах к разработке распределенных систем. Изобретение в конце 1970-х сети Ethernet существенно ускорило темпы развития. Почти сразу появилось изобилие новых языков, алгоритмов и приложений; их создание стимулировалось развитием аппаратного обеспечения. Например, как только рабочие станции и локальные сети стали относительно дешевыми, была разработана модель вычислений типа клиент-сервер; развитие сети Internet привело к рождению языка Java, Web-броузеров и множества новых приложений.

Первые мультипроцессоры появились в 1970-х годах, и наиболее заметными из них были SIMD-мультипроцессоры Illiac, разработанные в университете Иллинойса. Первые машины были дорогими и специализированными, однако в течение многих лет трудоемкие научные

вычисления выполнялись на векторных процессорах. Изменения начались в середине 1980-х годов с изобретения гиперкубовых машин в Калифорнийском технологическом институте и их коммерческой реализации фирмой Intel. Затем фирма Thinking Machines представила Connection Machine с массовым параллелизмом. Кроме того, фирма Cray Research и другие производители векторных процессоров начали производство многопроцессорных версий своих машин. В течение нескольких лет появлялись и вскоре исчезали многочисленные компании и машины. Однако сейчас группа производителей машин достаточна стабильна, а высокопроизводительные вычисления стали почти синонимом обработки с массовым параллелизмом.

В исторических справках следующих глав описываются разработки, связанные с самими этими главами. Ниже приведено несколько общих ссылок, касающихся аппаратного обеспечения, операционных систем, распределенных систем и параллельных вычислений. Это книги, к которым автор чаще всего обращался за справкой при написании данной книги. (Чтобы побольше узнать о какой-то теме, можно использовать хорошую поисковую программу в Internet.)

На данный момент классической книгой по архитектуре компьютеров является книга [Hennessy, Patterson, 1996]. Если вы хотите узнать больше о кэш-памяти, соединительных сетях и работе мультипроцессоров, начинайте с нее. В книге [Hwang, 1993] описана архитектура высокопроизводительных компьютеров, а книга [Almasi, Gottlieb, 1994] дает обзор приложений синхронных параллельных вычислений, программного обеспечения и архитектуры.

По операционным системам наиболее широко известны учебники [Tanenbaum, 1992] и [Silberschatz, Peterson, Galvin, 1998]. Книга [Tanenbaum, 1995] также дает превосходный обзор систем распределенного программирования. Книга [Mullender, 1993] содержит великолепный набор глав по всем темам распределенных систем, включая надежность и отказоустойчивость, обработку транзакций, файловые системы, системы реального времени и безопасность. Два дополнительных учебника — [Bacon, 1998] и [Bernstein, Lewis, 1993] — охватывают многопоточные параллельные системы, включая системы распределенных баз данных.

За последние несколько лет было написано множество книг по синхронным параллельным вычислениям. Два конкурирующих издания — [Kumar, Grama, Gupta, Karypis, 1994] и [Quinn, 1994] — описывают и анализируют параллельные алгоритмы для решения многочисленных комбинаторных и численных задач. Обе книги до некоторой степени охватывают темы программного и аппаратного обеспечения, но особое внимание уделяется разработке и анализу параллельных алгоритмов.

Еще четыре книги посвящены программным аспектам параллельных вычислений. В [Brinch Hansen, 1995] рассматриваются интересные вычислительные задачи и решаются с помощью небольшого числа парадигм программирования; задачи описаны исключительно ясно. В книге [Foster, 1995] представлены идеи и средства параллельного программирования, в особенности для машин с распределенной памятью. В книге есть отличные описания двух наиболее актуальных инструментальных средств — High Performance Fortran (HPF) и Message Passing Interface (MPI). В [Wilson, 1995] описаны четыре важные модели программирования (параллелизм по данным, разделяемые переменные, передача сообщений, генеративное взаимодействие) и показано, как с их помощью решать научные и инженерные задачи. В одном из приложений этой книги превосходно изложена краткая история основных событий в параллельных вычислениях. Последняя книга [Wilkinson, Allen, 1999] описывает множество приложений и показывает, как писать параллельные программы для их решения. Основное внимание в книге уделено передаче сообщений, но описываются также вычисления с разделяемыми переменными.

Книга [Foster, Kesselman, 1999] представляет новый многообещающий подход к распределенным быстродействующим вычислениям, использующим так называемую вычислительную сеть. (Этот подход описан в главе 12.) Книга начинается исчерпывающим введением

в вычислительные сети, а затем рассматривает приложения, инструменты программирования, сервис и инфраструктуру. Главы книги написаны экспертами, которые работают над тем, чтобы вычислительные сети стали реальностью.

# Литература

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing.* 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Bacon, J. 1998. *Concurrent Systems: Operating Systems, Database and Distributed Systems: An Integrated Approach.* 2nd ed. Reading, MA: Addison-Wesley.

Bernstein, A. J., and P. M. Lewis. 1993. *Concurrency in Programming and Database Systems.* Boston, MA: Jones and Bartlett.

Brinch Hansen, P. 1995. *Studies in Computational Science.* Englewood Cliffs, NJ: Prentice-Hall.

Foster, I. 1995. *Designing and Building Parallel Programs.* Reading, MA: Addison-Wesley.

Foster, I., and C. Kesselman, eds. 1999. *The Grid: Blueprint for a New Computing Infrastructure.* San Francisco, CA: Morgan Kaufmann.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach,* 2nd ed. San Francisco, CA: Morgan Kaufmann.

Hwang, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York, NY: McGraw-Hill.

Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Menlo Park, CA: Benjamin/Cummings.

Mullender, S., ed. 1993. *Distributed Systems,* 2nd ed. Reading, MA: ACM Press and Addison-Wesley.

Quinn, M. J. 1994. *Parallel Computing: Theory and Practice.* New York, NY: McGraw-Hill.

Silberschatz, A., J. Peterson, and P. Galvin. 1998. *Operating System Concepts,* 5th ed. Reading, MA: Addison-Wesley.

Tanenbaum, A. S. 1992. *Modern Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Tanenbaum, A. S. 1995. *Distributed Operating Systems.* Englewood Cliffs, NJ: Prentice-Hall.

Wilkinson, B., and M. Allen. 1999. *Parallel Programming: Techniques and Applications Using* Networked Workstations and РАЛ= 2,21*arallel Computers.* Englewood Cliffs, NJ: Prentice-Hall.

Wilson, G. V. 1995. *Practical Parallel Programming.* Cambridge, MA: MIT Press.

# Упражнения

1.1. Определите характеристики доступных вам многопроцессорных машин. Сколько процессоров в каждой машине и каковы их рабочие частоты? Насколько велик размер их кэш-памяти, как она организована? Каково время доступа? Какой используется протокол согласования памяти? Как организована связующая сеть? Каково время удаленного доступа к памяти или передачи сообщения?

1.2. Многие задачи можно решить более эффективно с помощью параллельной, а не последовательной программы (конечно, при наличии соответствующего аппаратного обеспечения). Рассмотрите программы, которые вы писали раньше, и выберите две из них, которые можно переписать как параллельные. Одна из них должна быть итеративной, а другая — рекурсивной. Затем: а) запишите кратко условия задач и б) разработайте псевдокод параллельных программ, решающих поставленные задачи.

1.3. Рассмотрите умножение матриц в разделе 1.4:

а) напишите последовательную программу для решения этой задачи. Размер матрицы n должен быть аргументом командной строки. Инициализируйте каждый элемент матриц a и b значением 1.0 (тогда значение каждого элемента результирующей матрицы c будет n);

б) напишите параллельную программу для решения этой задачи. Вычислите полосы результата параллельно, используя P рабочих процессов. Размер матрицы n и число рабочих процессов P должны быть аргументами командной строки. Вновь инициализируйте каждый элемент матриц a и b значением 1.0;

в) сравните производительность ваших программ. Поэкспериментируйте с разными значениями параметров n и P. Подготовьте график результатов и объясните, что вы заметили;

г) преобразуйте ваши программы для умножения прямоугольных матриц. Размер матрицы a должен быть p×q, а матрицы b — q×r. Тогда размер результирующей матрицы будет p×r. Повторите часть *в* данного упражнения для новых программ.

1.4. В программах умножения матриц при вычислении промежуточных произведений используется умножение пар элементов и сложение результатов. Все умножения можно выполнять параллельно. Пары произведений тоже можно склады-вать параллельно:

а) создайте двоичное дерево выражений для иллюстрации работы этого процесса. Предположим, что векторы имеют длину n, и для простоты допустим, что n является степенью числа 2. Листья дерева должны быть элементами вектора (строки матрицы a и столбца матрицы b). Другие узлы дерева должны быть отмечены операциями сложения или умножения;

б) многие современные процессоры являются так называемыми сверхскалярными процессорами. Это значит, что они могут начать выполнение сразу нескольких инструкций. Рассмотрим машину, которая может начать выполнять сразу две инструкции. Напишите псевдокод на уровне языка ассемблера для реализации двоичного дерева выражения, построенного вами в части *а*. Предположим, что существуют инструкции для загрузки, сложения и умножения регистров и доступно любое число регистров. Максимизируйте число пар инструкций, выполнение которых можно начать одновременно;

в) предположим, что загрузка регистра занимает 1 такт работы процессора, сложение — тоже 1, а умножение — 8 тактов. Каким будет время выполнения вашей программы?

1.5. В первой строке первой программы для параллельного умножения матриц (см. раздел 1.4) находится оператор co для строк, а во второй строке — оператор for для столбцов. Допустим, что оператор co изменен на for, for — на co, а остальная часть программы осталась без изменений. Программа вычисляет строки последовательно, а столбцы для каждой строки — параллельно:

а) будет ли программа верна? Точнее, будут ли вычисляться те же результаты?

б) будет ли программа настолько же эффективной? Объясните свой ответ.

1.6. Точки на единичной окружности с центром в начале координат определяются функцией f(x) = sqrt(1-x$^2$).[1] Напомним, что площадь круга вычисляется по формуле $\pi r^2$, где r — радиус. Используйте адаптивную квадратурную программу из раздела 1.5 для аппроксимации значения $\pi$ путем вычисления площади первого (верхнего правого)

[1] Точнее, эта функция определяет точки верхней полуокружности. — *Прим. перев.*

квадранта единичной окружности и умножения результата на 4. (Можно также проинтегрировать от 0.0 до 1.0 функцию f(x) = 4/(1+$x^2$).)

1.7. Рассмотрите задачу квадратуры, описанную в разделе 1.5:

а) напишите четыре программы для ее решения: 1) последовательную итеративную программу, в которой используется фиксированное число интервалов, 2) последовательную рекурсивную программу с адаптивной квадратурой, 3) параллельную итеративную программу с фиксированным числом интервалов и 4) параллельную рекурсивную программу, использующую адаптивную квадратуру. Проинтегрируйте функцию с графиком интересной формы, например sin(x)*exp(x). Ваши программы должны использовать параметры командной строки для интервала значений x, числа отрезков (при фиксированной квадратуре) и значения EPSILON (при адаптивной квадратуре);

б) поэкспериментируйте со своими программами при различных значениях аргументов. Какова длительность их выполнения? Насколько точны ответы? Быстро ли сходятся программы? Объясните свои наблюдения.

1.8. Программа параллельной адаптивной квадратуры (см. раздел 1.5) создает большое число процессов, обычно превышающее число процессоров в системе:

а) измените программу, чтобы она создавала приблизительно T процессов, где T — аргумент командной строки, определяющий пороговое значение. Используйте глобальную переменную для хранения числа созданных вами процессов. (В этой задаче предположим, что глобальную переменную можно безопасно увеличивать.) Если внутри тела функции quad создано больше, чем T процессов, используйте последовательную рекурсию. В противном случае используйте параллельную рекурсию и каждый раз добавляйте 2 к глобальному счетчику;

б) реализуйте и протестируйте параллельную программу из текста книги и из вашего ответа к пункту *а*. Выберите для интегрирования интересную функцию, поэкспериментируйте с различными значениями переменных T и EPSILON. Сравните производительность обеих программ.

1.9. Напишите последовательную рекурсивную программу для реализации алгоритма быстрой сортировки массива из n чисел. Преобразуйте свою программу для использования рекурсивного параллелизма. Будьте внимательны, чтобы обеспечить независимость параллельных вызовов. Реализуйте обе программы и сравните их производительность.

1.10. Соберите данные о каналах ОС Unix. Как они реализованы в вашей системе? Каков максимальный размер канала? Как синхронизируются операции read и write? Проверьте, можете ли вы провести эксперимент, который приведет к блокировке операции write из-за переполнения канала. Можете ли вы создать параллельную программу, которая “зависнет”, т.е. все процессы будут ждать друг друга?

1.11. Большая часть вычислительных возможностей серверов сейчас используется для электронной почты, Web-страниц, файлов и т.д. Какие типы серверов используются в ваших вычислительных средствах? Выберите один из серверов (если есть) и выясните, как на нем организовано программное обеспечение. Какие он выполняет процессы (потоки)? Как они планируются? Что представляют собой клиенты? Выделяется ли в нем по одному потоку на клиентский запрос, существует ли фиксированное число серверных потоков, или это происходит как-то иначе? Какие данные разделяются потоками сервера? Когда и зачем потоки сервера синхронизируются друг с другом?

1.12. Рассмотрите две программы для распределенного умножения матриц из раздела 1.8:

а) сколько сообщений отсылается и принимается каждой программой? Каковы размеры сообщений? Не забудьте, что во второй программе есть и управляющий процесс;

б) измените программу, чтобы она использовала P рабочих процессов, где P — делитель числа n. В частности, каждый рабочий процесс должен вычислять не одну строку, а n/P строк (или столбцов) результата;

в) сколько сообщений отсылается программой для части *б* и какого они размера?

1.13. Матрица T является транспонированной матрицей M, если для всех значений i и j выполняется равенство T[i,j] = M[j,i]:

а) напишите параллельную программу, которая с использованием разделяемых переменных транспонирует матрицу M размером n×n. Используйте P рабочих процессов. Для простоты предположим, что число n кратно P;

б) напишите параллельную программу транспонирования квадратной матрицы M, в которой используется пересылка сообщений. Вновь используйте P рабочих процессов и предположите, что n кратно P. Выясните, как распределить данные и собрать результаты;

в) измените ваши программы, чтобы они обрабатывали случай, когда n не кратно P;

г) поэкспериментируйте с вашими программами при различных значениях n и P. Какова их производительность?

1.14. Семейство программ grep ОС Unix выполняет поиск шаблонов в файлах. Напишите упрощенную версию программы grep, которая использует два аргумента: цепочку символов и имя файла. Все строки файла с найденной цепочкой программа должна выводить в файл stdout:

а) измените программу, чтобы она обрабатывала один за другим два файла (добавьте третий аргумент для указания имени файла);

б) измените программу, чтобы она выполняла поиск в двух файлах параллельно. Оба процесса должны выводить данные в файл stdout;

в) поэкспериментируйте с вашими программами к пунктам *а* и *б*. Их выход должен отличаться, по крайней мере, в некоторые моменты времени. Более того, выход параллельной программы может не всегда быть одним и тем же. Проверьте, можете ли вы наблюдать это явление.