МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

М. Л. ДОЛЖЕНКОВА

О. В. КАРАВАЕВА

ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

Киров
2014

УДК 004.451(07)
Д643

Рекомендовано к изданию методическим советом
факультета автоматики и вычислительной техники
ФГБОУ ВПО «ВятГУ»

Допущено редакционно-издательской комиссией методического совета ФГБОУ ВПО «ВятГУ» в качестве учебного пособия для студентов специальностей 230101.65 «Вычислительные машины, комплексы системы и сети», 090302.65 «Информационная безопасность телекоммуникационных систем»; направлений 230100.62 «Информатика и вычислительная техника», всех профилей подготовки, 090900.62 «Информационная безопасность», 210400.62 «Инфокоммуникационные технологии и системы связи»

Рецензенты:
кандидат технических наук, заведующий кафедрой информатики
и вычислительной техники ВСЭИ г. Киров И. Е. Петров;

кандидат технических наук, доцент кафедры автоматики и телемеханики
ФГБОУ ВПО «ВятГУ» В. В. Куклин

Долженкова, М. Л.
Д644 Основы параллельного программирования: учебное пособие / М. Л. Долженкова, О. В. Караваева. – Киров: ФГБОУ ВПО «ВятГУ», 2014. – 114 с.

УДК 004.42(07)

В учебном пособии рассмотрены принципы организации параллельных вычислительных процессов.

Пособие рассчитано на студентов, изучающих дисциплины «Сетевое программное обеспечение», «Операционные системы», «Параллельное программирование», «Параллельные вычисления». Пособие может использоваться студентами других направлений при решении сложных задач, требующих организации параллельных вычислений.

Тех. редактор А. В. Куликова

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ 3

ВВЕДЕНИЕ. Актуальность проблемы 5

1. Виды аппаратного и программного параллелизма. 12

1.1. Классификация вычислительных систем. 12

1.1.1. Классификация Флинна 12

1.1.2. Классификация Джонсона 15

1.1.3. Классификация архитектур вычислительных систем 25

2. Метрики параллелизма. 40

2.1. Моделирование и анализ параллельных вычислений 40

2.2. Закон Амдала 58

2.3. Закон Густафсона 62

3. Разработка параллельного алгоритма 66

3.1. Этапы разработки параллельных алгоритмов 69

3.1.1. Разделение вычислений на независимые части 69

3.1.2. Выделение информационных зависимостей 78

3.1.3. Масштабирование набора подзадач 81

3.1.4. Распределение подзадач между процессорами 82

3.2. Распараллеливание арифметических вычислений и параллельная сортировка 85

4. Издержки и выигрыш при реализации параллельных и векторных вычислений 93

4.1. Способы векторизации и распараллеливания программ 93

4.2. Применение разных языков программирования 95

4.2.1. Сходство алгоритмов – параллелизм данных 97

4.2.2. Различие алгоритмов – параллелизм действий 99

4.2.3. Предельное быстродействие векторных программ 100

4.2.4. Две части программы – скалярная и векторная 100

4.3. Параллельные ЭВМ и параллельные программы 102

4.3.1. Три части программы – параллельная, последовательная и обмен данными 103

4.3.2. Синхронизация процессов, равномерность загрузки процессов ... 104

5. Распараллеливающие компиляторы 106

Библиографический список 113

Электронные ресурсы 113

ВВЕДЕНИЕ. Актуальность проблемы

Применение параллельных вычислительных систем (ПВС) является стратегическим направлением развития вычислительной техники. Это обстоятельство вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным наличием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно. Так, проблемы «большого вызова» возможностям современной науки и техники – моделирование климата, генная инженерия, проектирование интегральных схем, анализ загрязнения окружающей среды, создание лекарственных препаратов и др. – требуют для своего анализа ЭВМ с производительностью более 1000 миллиардов операций с плавающей запятой в секунду (1 TFlops).

Параллельные вычислительные системы применяются при решении следующих классов задач:

- решение задач, связанных с большим объемом вычислений;
- решение задач реального времени;
- построение системы высокой надежности;
- обработка больших объемов данных.

Проблема создания высокопроизводительных вычислительных систем относится к числу наиболее сложных научно-технических задач современности. Ее разрешение возможно только при всемерной концентрации усилий многих талантливых ученых и конструкторов, предполагает использование всех последних достижений науки и техники и требует значительных финансовых инвестиций.

Организация параллельности вычислений, когда в один и тот же момент выполняется одновременно несколько операций обработки данных, осуществляется в основном за счет введения избыточности функциональных устройств (многопроцессорности). В этом случае можно достичь ускорения процесса решения вычислительной задачи, если осуществить разделение

применяемого алгоритма на информационно независимые части и организовать выполнение каждой части вычислений на разных процессорах. Подобный подход позволяет выполнять необходимые вычисления с меньшими затратами времени, при этом получение максимально возможного ускорения ограничивается только числом имеющихся процессоров и количеством «независимых» частей в выполняемых вычислениях.

До недавнего времени применение параллелизма не имело столь широкого распространения, как это ожидалось многими исследователями. Одной из возможных причин подобной ситуации являлась высокая стоимость высокопроизводительных систем (приобрести суперЭВМ могли себе позволить только крупные компании и организации). Современная тенденция построения параллельных вычислительных комплексов из типовых конструктивных элементов (микропроцессоров, микросхем памяти, коммуникационных устройств), массовый выпуск которых освоен промышленностью, снизила влияние этого фактора, и в настоящий момент практически каждый потребитель может иметь в своем распоряжении многопроцессорные вычислительные системы (МВС) достаточно высокой производительности. Кардинально ситуация изменилась в сторону параллельных вычислений с появлением многоядерных процессоров, которые уже в 2006 г. использовались более чем в 70 % компьютерных систем.

Другая и, пожалуй, теперь основная причина сдерживания массового распространения параллелизма состоит в том, что для проведения параллельных вычислений необходимо «параллельное» обобщение традиционной – последовательной технологии решения задач на ЭВМ. Так, численные методы в случае многопроцессорных систем должны проектироваться как совокупность параллельных и взаимодействующих между собой процессов, допускающих исполнение на независимых процессорах. Применяемые алгоритмические языки и системное

программное обеспечение должны обеспечивать создание параллельных программ, организовывать синхронизацию и взаимоисключение асинхронных процессов и т. п.

При использовании параллельных вычислительных систем можно выделить следующий ряд общих проблем:

1. Высокая стоимость параллельных систем – в соответствии с подтверждаемым на практике законом Гроша[1], производительность компьютера возрастает пропорционально квадрату его стоимости и, как результат, гораздо выгоднее получить требуемую вычислительную мощность приобретением одного производительного процессора, чем использование нескольких менее быстродействующих процессоров.

Для ответа на данное замечание следует учесть, что рост быстродействия последовательных ЭВМ не может продолжаться бесконечно; кроме того, компьютеры подвержены быстрому моральному старению и результативность наиболее мощного процессора на данный момент времени быстро перекрывается новыми моделями компьютеров (а обновление морально устаревшего компьютера требует, как правило, больших финансовых трат).

2. Потери производительности для организации параллелизма – согласно гипотезе Минского, ускорение, достигаемое при использовании параллельной системы, пропорционально двоичному логарифму от числа процессоров.

Безусловно, подобная оценка ускорения может иметь место при распараллеливании определенных алгоритмов. Вместе с тем существует

[1] Закон Гроша – это достаточно известный в компьютерном мире закон, который появился задолго до наступления эры Intel. В 60-х и 70-х годах прошлого столетия, когда компьютеры представляли собой огромные шкафы, занимающие внушительные пространства, скорость выполнения операций того или иного вычислительного комплекса целиком и полностью определялась затраченной на него суммой денег.

Тогда и был актуален Закон Гроша, который гласил, что производительность вычислительной машины всегда пропорциональна квадрату ее стоимости. То есть покупка одного дорогого компьютера была выгоднее, чем покупка двух менее производительных за ту же сумму денег.

С появлением мини-компьютеров этот закон утратил свою актуальность, ибо с развитием технологий распараллеливания несколько небольших компьютеров гораздо лучше справлялись с вычислительными задачами, чем один громоздкий мэйнфрейм.

большое количество задач, при параллельном решении которых достигается стопроцентное использование всех имеющихся процессоров параллельной вычислительной системы.

3. Постоянное совершенствование последовательных компьютеров. В соответствии с законом Мура – мощность последовательных процессоров возрастает практически в 2 раза каждые 18–24 месяцев и, как результат, необходимая производительность может быть достигнута и на «обычных» последовательных компьютерах.

Данное замечание близко по сути к первому из приведенных здесь возражений против параллелизма. В дополнение к имеющемуся ответу на это замечание можно отметить, что аналогичное развитие свойственно и параллельным системам. Кроме того, применение параллелизма позволяет получать желаемое ускорение вычислений без какого-либо ожидания новых более быстродействующих процессоров.

4. Существование последовательных вычислений – в соответствии с законом Амдала, ускорение выполнения программы за счет распараллеливания ее инструкций на множестве вычислителей ограничено временем, необходимым для выполнения ее последовательных инструкций (т. е., например, при наличии всего 10 % последовательных команд в выполняемых вычислениях эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных).

Данное замечание характеризует одну из самых серьезных проблем в области параллельного программирования – алгоритмов без определенной доли последовательных команд практически не существует. Однако часто доля последовательных действий характеризует не возможность параллельного решения задач, а последовательные свойства применяемых алгоритмов. Как результат, доля последовательных вычислений может быть существенно снижена при выборе более подходящих для распараллеливания алгоритмов.

5. Эффективность параллелизма зависит от характерных свойств параллельных систем – в отличие от единственности классической схемы фон Неймана, параллельные системы отличаются существенным разнообразием архитектурных принципов построения, и максимальный эффект от использования параллелизма может быть получен только при полном использовании всех особенностей аппаратуры. Как результат, перенос параллельных алгоритмов и программ между разными типами систем становится затруднительным (если вообще возможен).

Для ответа на данное замечание следует отметить, что «однородность» последовательных ЭВМ также является кажущейся и их эффективное использование тоже требует учета свойств аппаратуры. С другой стороны, при всем разнообразии архитектур параллельных систем, тем не менее, существуют и определенные «устоявшиеся» способы обеспечения параллелизма (конвейерные вычисления, многопроцессорные системы и т. п.). Кроме того, инвариантность создаваемого программного обеспечения может быть обеспечена и при помощи использования типовых программных средств поддержки параллельных вычислений (типа программных библиотек MPI, PVM и др.).

6. Существующее программное обеспечение ориентировано в основном на последовательные ЭВМ. Для большого числа задач уже имеется разработанное программное обеспечение, и все эти программы ориентированы главным образом на последовательные ЭВМ – как результат, переработка такого количества программ для параллельных систем вряд ли окажется возможным.

Ответ на данное возражение сравнительно прост – если существующие программы обеспечивают решение поставленных задач, то, конечно, переработка этих программ и не является необходимой. Однако если последовательные программы не позволяют получать решения задач за приемлемое время или же возникает необходимость решения новых задач, то

необходимой становится разработка нового программного обеспечения и эти программы могут реализовываться в параллельном исполнении.

В настоящее время, в связи с развитием параллельных возможностей современных микропроцессоров можно выделить три основных направления в развитии параллельных вычислительных систем.

Во-первых, это многоядерность, когда несколько ядер находятся в корпусе одного процессора.

Во-вторых, это технологии с явным параллелизмом команд (EPIC – Explicitly Parallel Instruction Computing). Архитектура всех современных 64-разрядных процессоров Intel построена на использовании технологии EPIC. Параллелизм сегодня стал главным ресурсом наращивания вычислительной мощности компьютеров. Существенным препятствием к параллельному выполнению программ являются так называемые точки ветвления, в которых решается вопрос, по какому из нескольких возможных путей пойдет выполнение программы после этой точки. Чем более совершенным является механизм предсказания ветвлений, тем лучше может быть распараллелена программа. При наличии избыточных вычислительных ресурсов можно начать выполнять сразу два возможных направления работы программы, не дожидаясь, пока ее основная ветвь дойдет до точки ветвления. После того как программа достигнет точки ветвления, можно уже окончательно выбрать результаты, полученные по одной из возможных ветвей. Сущность EPIC заключается в том, что блок предсказаний ветвлений выносится из аппаратной логики процессора в компилятор. Анализируя программу, компилятор сам определяет параллельные участки и дает процессору явные инструкции по их выполнению – отсюда и следует название архитектуры EPIC. Модернизация компилятора является более гибкой и простой, чем модификация аппаратной части параллельной вычислительной системы. Например, можно установить новую версию компилятора и посмотреть, как это отразится на скорости работы прикладных программ, создаваемых с помощью этого компилятора. При

этом следует иметь в виду, что изменить устройство процессора в уже работающей вычислительной системе невозможно. Установленный процессор можно только заменить на новый, подходящий для данной системы процессор, если таковой имеется в наличии. Поэтому архитектура EPIC позволяет более эффективно модифицировать работающие вычислительные системы за счет установки более совершенных версий компилятора.

В-третьих, это многопоточность, когда в каждом ядре процессора выполняется одновременно несколько потоков, конкурирующих между собой.

Главная задача в развитии вышеперечисленных направлений – существенное повышение производительности вычислительных систем. Без развития параллельных технологий решить эту задачу не представляется возможным, поскольку современные технологии микроэлектроники подошли к технологическому барьеру, препятствующему дальнейшему существенному увеличению тактовой частоты работы процессора и изготовлению процессоров по технологиям, существенно меньшим 15 нанометров.

1. Виды аппаратного и программного параллелизма

1.1. Классификация вычислительных систем

1.1.1. Классификация Флинна

Майкл Флинн предложил в 1966 году классификацию вычислительных систем, основанную на количестве потоков входных данных и количестве потоков команд, которые эти данные обрабатывают. Таким образом, рамках систематики Флинна основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия последовательностей (потоков) выполняемых команд и обрабатываемых данных. При таком подходе различают следующие основные типы систем:

SISD (Single Instruction, Single Data) – это обычные последовательные компьютеры (рис. 1.1). Программа принимает один поток данных и выполняет один поток инструкций по обработке этих данных. Иными словами, инструкции выполняются последовательно, и каждая инструкция оперирует минимальным количеством данных (например, сложение двух чисел).

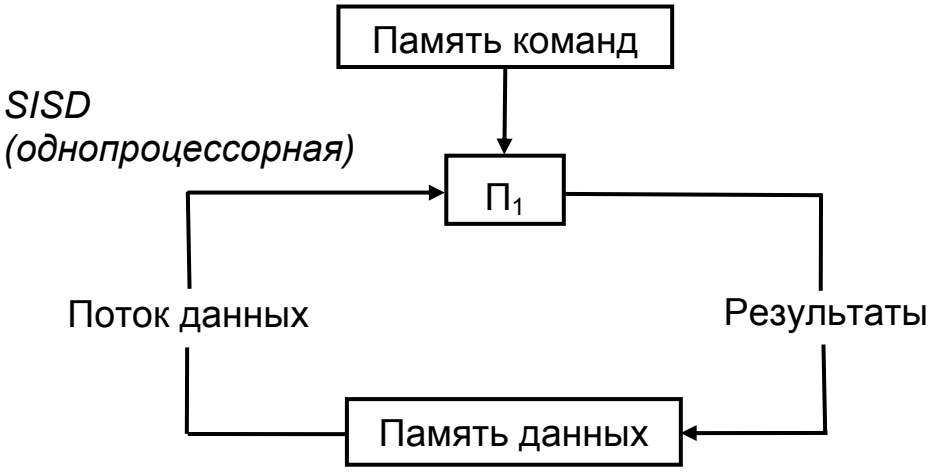


Рис. 1.1. Система SISD. одиночный поток команд, одиночный поток данных

SIMD (Single Instruction, Multiple Data) – системы с одиночным потоком команд и множественным потоком данных (рис. 1.2). Подобный

класс составляют многопроцессорные вычислительные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов; такой архитектурой обладают, например, многопроцессорные системы с единым устройством управления. Например, выполняется сложение одновременно восьми пар чисел. Такие компьютеры называются векторными, так как подобные операции выполняются аналогично операциям с векторами (когда, например, сложение двух векторов означает одновременное сложение всех их компонентов). Зачастую векторные инструкции присутствуют в дополнение к обычным «скалярным» инструкциям, и называются SIMD-расширением (или векторным расширением). Примеры популярных SIMD-расширений: MMX, 3DNow!, SSE и др. Этот подход широко использовался в предшествующие годы (системы ILLIAC IV или CM-1 компании Thinking Machines), в последнее время его применение ограничено, в основном, созданием специализированных систем.

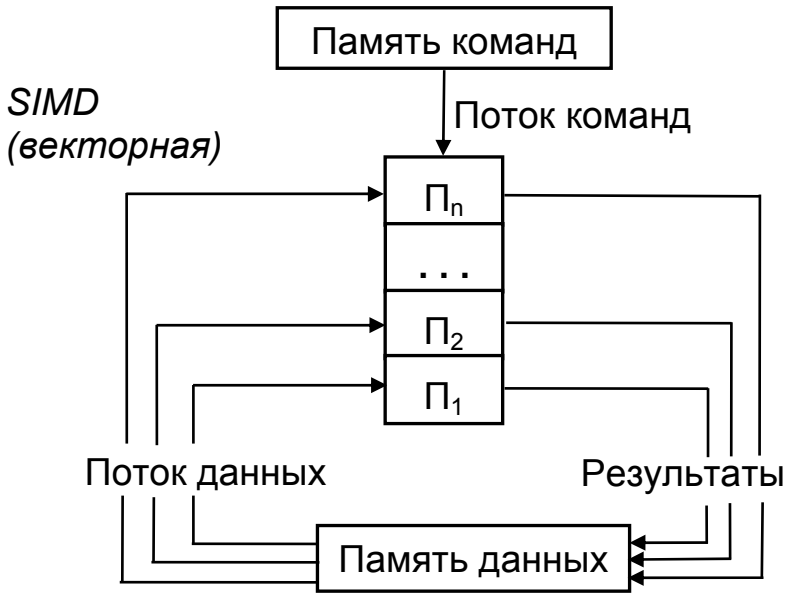


Рис. 1.2. Система SIMD. одиночный поток команд, множественный поток данных

MISD (Multiple Instruction Single Data) – разные потоки инструкций выполняются с одними и теми же данными (рис. 1.3). Обычно такие системы

не приводят к ускорению вычислений, так как разные инструкции оперируют одними и теми же данными, в результате на выходе системы получается один поток данных. К таким системам относят различные системы дублирования и защиты от сбоев, когда, например, несколько процессоров дублируют вычисления друг друга для надежности. Иногда к этой категории относят конвейерные архитектуры. Среди процессоров производства Intel, конвейер присутствует, начиная с процессора Pentium. Однако ряд специалистов считает, что примеров конкретных ЭВМ, соответствующих данному типу вычислительных систем, не существует и введение подобного класса предпринимается для полноты классификации.

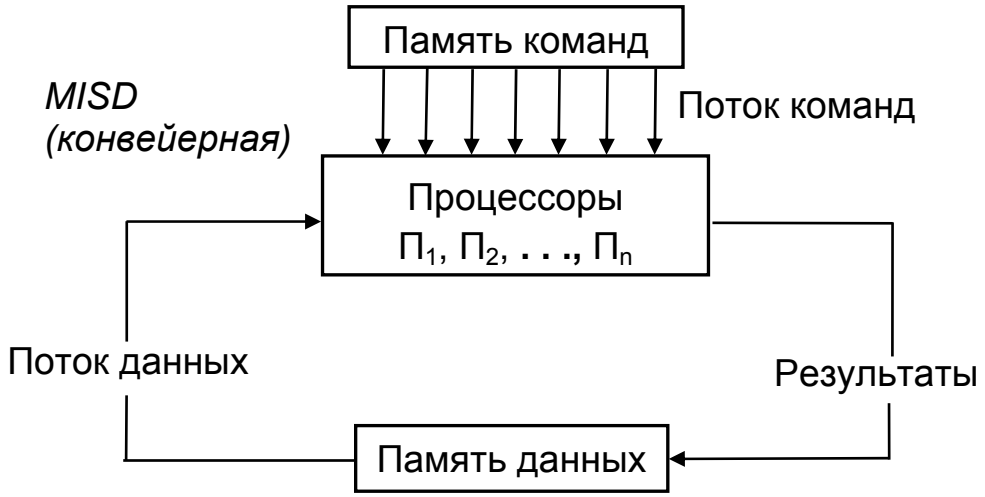


Рис. 1.3. Система MISD – множественный поток команд, одиночный поток данных

MIMD (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных (рис. 1.4). Это системы наиболее общего вида, поэтому их проще всего использовать для решения различных параллельных задач. К подобному классу относится большинство параллельных многопроцессорных вычислительных систем.

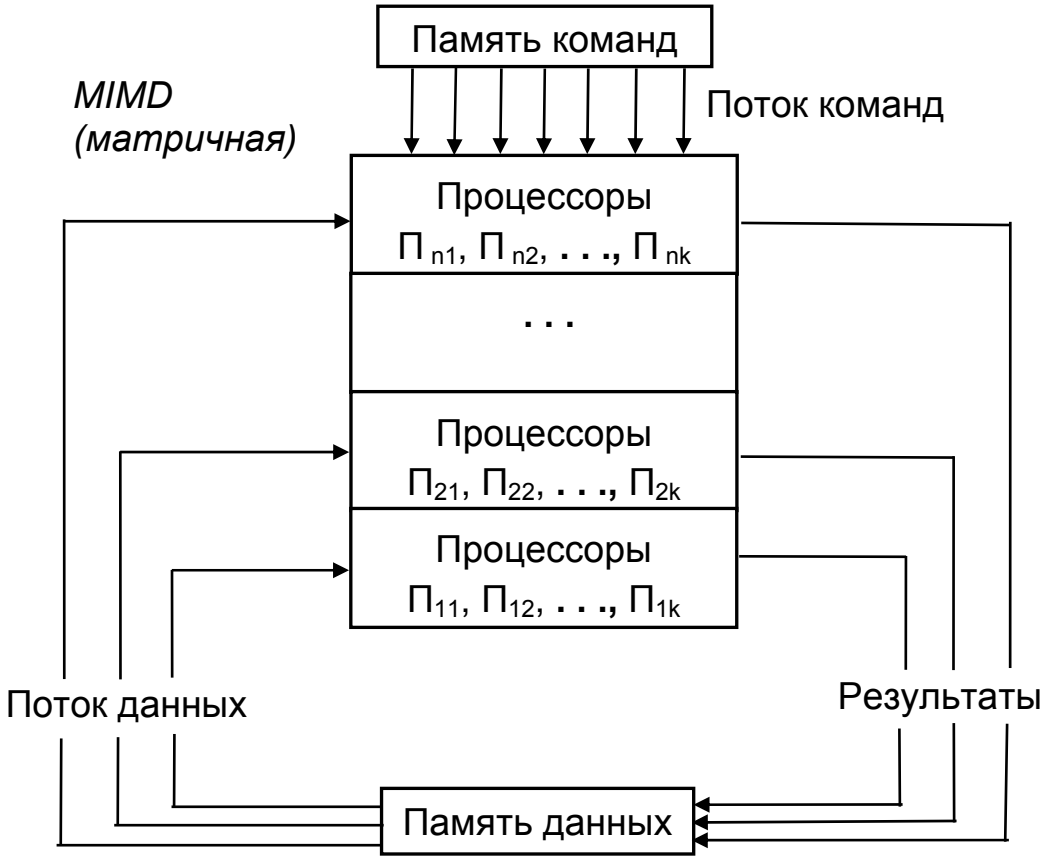


Рис. 1.4. Система MIMD – множественный поток команд, множественный поток данных

Следует отметить, что хотя систематика Флинна широко используется при конкретизации типов компьютерных систем, такая классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разнородность) оказываются отнесены к одной группе MIMD. Как результат, многими исследователями предпринимались неоднократные попытки детализации систематики Флинна. Так, например, для класса MIMD предложена практически общепризнанная структурная схема, в которой дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах (рис. 1.5).

1.1.2. Классификация Джонсона

Одной из самых известных детализаций систематики Флинна для MIMD-систем, в принято считать классификацию Джонсона.

Е. Джонсон предложил проводить классификацию MIMD архитектур на основе структуры памяти и реализации механизма взаимодействия и синхронизации между процессорами.

По структуре оперативной памяти существующие вычислительные системы делятся на две большие группы: либо это системы с общей памятью, прямо адресуемой всеми процессорами, либо это системы с распределенной памятью, каждая часть которой доступна только одному процессору. Одновременно с этим, и для межпроцессорного взаимодействия существуют две альтернативы: через разделяемые переменные или с помощью механизма передачи сообщений. Исходя из таких предположений, можно получить четыре класса MIMD архитектур, уточняющих систематику Флинна:

- общая память – разделяемые переменные (GMSV);
- распределенная память – разделяемые переменные (DMSV);
- распределенная память – передача сообщений (DMMP);
- общая память – передача сообщений (GMMP).

Опираясь на такое деление, Джонсон вводит названия для некоторых классов. Так вычислительные системы, использующие общую разделяемую память для межпроцессорного взаимодействия и синхронизации, он называет системами с разделяемой памятью, например, CRAY Y-MP (по его классификации это класс 1). Системы, в которых память распределена по процессорам, а для взаимодействия и синхронизации используется механизм передачи сообщений он называет архитектурами с передачей сообщений, например NCube, (класс 3). Системы с распределенной памятью и синхронизацией через разделяемые переменные, как в BBN Butterfly, называются гибридными архитектурами (класс 2).

В качестве уточнения классификации автор отмечает возможность учитывать вид связи между процессорами: общая шина, переключатели, разнообразные сети и т. п.

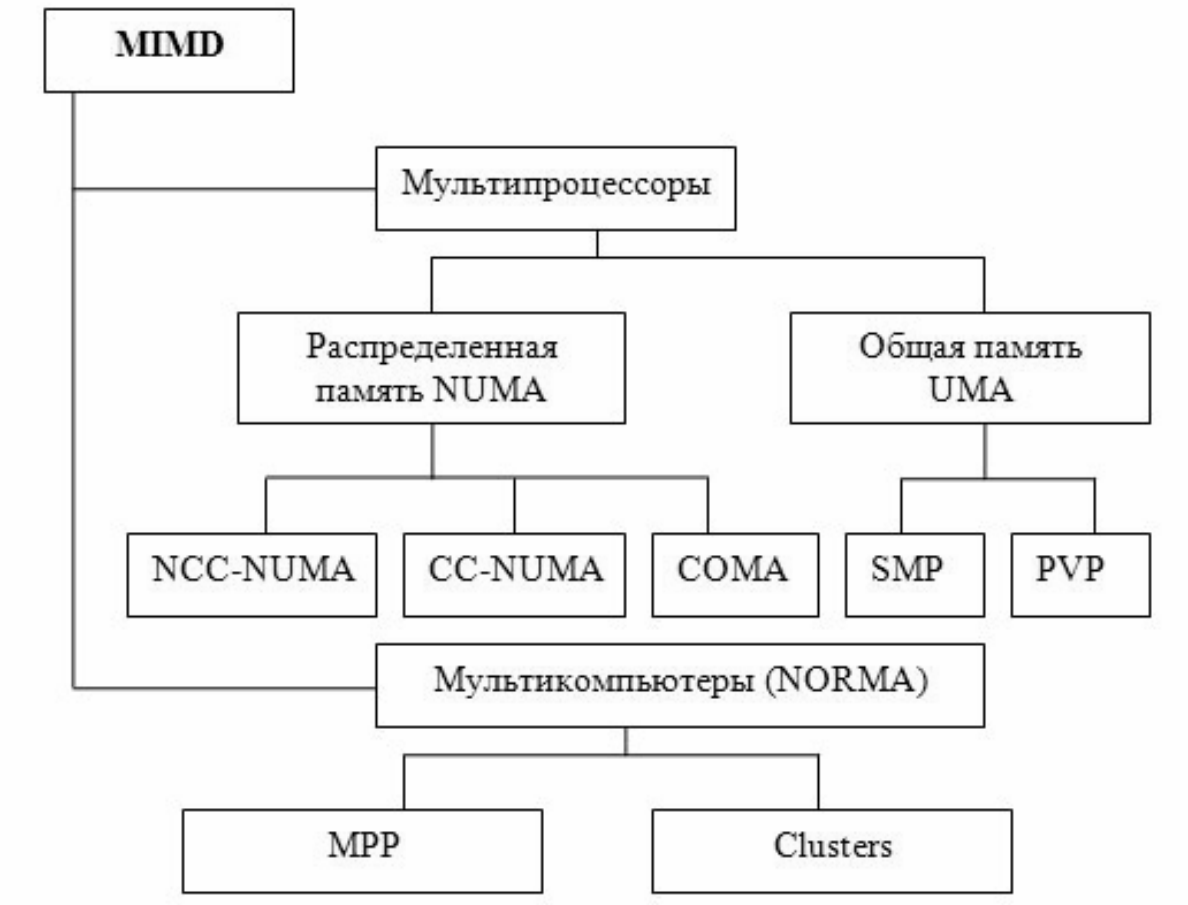


Рис. 1.5. Классификация многопроцессорных вычислительных систем

Такой подход позволяет различать два важных типа многопроцессорных систем – multiprocessors (мультипроцессоры или системы с общей разделяемой памятью) и multicomputers (мультикомпьютеры или системы с распределенной памятью).

Для дальнейшей систематики мультипроцессоров учитывается способ построения общей памяти. Первый возможный вариант – использование единой (централизованной) общей памяти (shared memory) (рис. 1.6 а). Системами с общей памятью называют системы, в которых несколько процессоров имеют общую оперативную память. Чаще всего встречающиеся системы этого типа – компьютеры с многоядерными процессорами (multi-core).

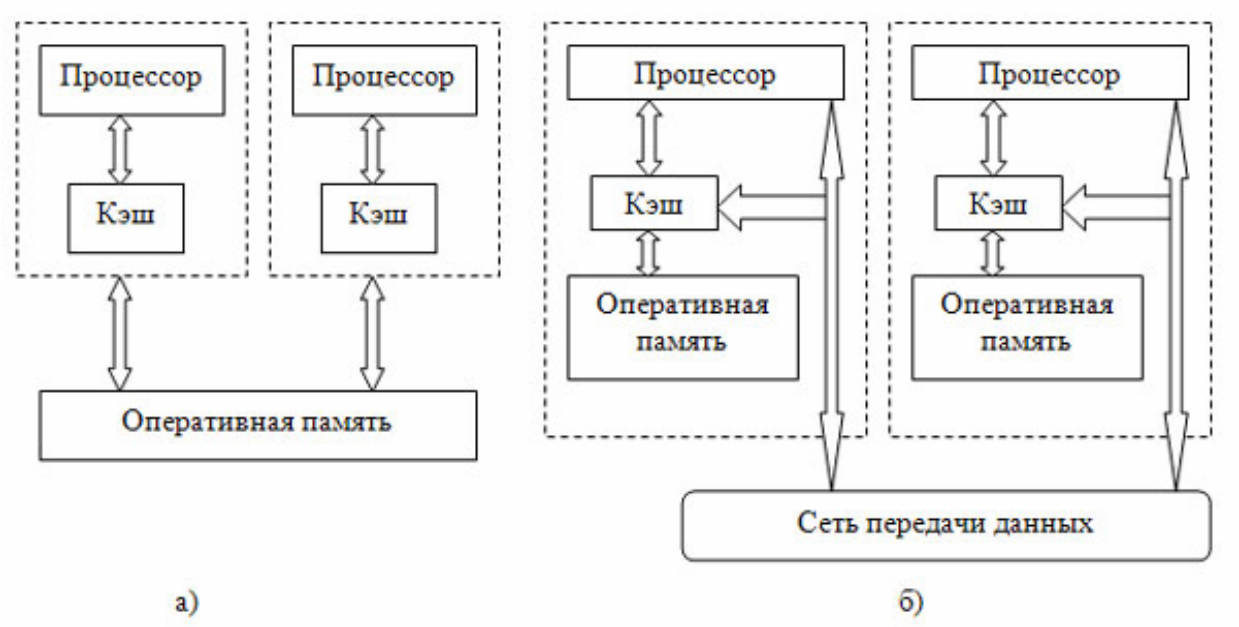


Рис. 1.6. Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным (б) доступом к памяти

При использовании систем с общей памятью не требуется обмен данными: данные, помещенные в память одним процессором, автоматически становятся доступными другим процессорам. Соответственно, система не должна тратить время на пересылку данных.

Для таких систем просто писать программы: можно, например, создать несколько вычислительных потоков, или же снабдить программу специальными директивами (например, технология OpenMP), которые подскажут компилятору, как распараллеливать программу. Кроме того, возможно полностью автоматическое распараллеливание программы компилятором.

Компактность систем: может быть реализована в виде нескольких процессоров на одной материнской плате, и/или в виде нескольких ядер внутри процессора.

Такой подход обеспечивает однородный доступ к памяти (uniform memory access или UMA) и служит основой для построения векторных параллельных процессоров (parallel vector processor или PVP) и симметричных мультипроцессоров (symmetric multiprocessor или SMP).

Среди примеров первой группы – суперкомпьютер Cray T90, ко второй группе относятся IBM eServer, Sun StarFire, HP Superdome, SGI Origin и др.

Одной из основных проблем, которые возникают при организации параллельных вычислений на таких системах, является доступ с разных процессоров к общим данным и обеспечение в связи с этим однозначности (когерентности) содержимого разных кэшей. Дело в том, что при наличии общих данных копии значений одних и тех же переменных могут оказаться в кэшах разных процессоров. Если в такой ситуации (при наличии копий общих данных) один из процессоров выполнит изменение значения разделяемой переменной, то значения копий в кэшах других процессоров окажутся не соответствующими действительности и их использование приведет к некорректности вычислений.

Обеспечение однозначности кэшей обычно реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти. Следует отметить, что необходимость обеспечения когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при параллельных вычислениях приводит к необходимости синхронизации взаимодействия одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить взаимоисключение, чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Допустим, у нас есть объект, содержащий числа A и B, и для правильной работы объекта нужно, чтобы A + B всегда было равно нулю. Если первый процесс изменит A, и не успеет изменить B прежде, чем второй процесс прочтет A и B, то второй процесс получит неправильный объект, в котором A + B не равно нулю. Для решения подобных проблем можно использовать критические секции. Если

поток инструкций первого процесса входит в критическую секцию с идентификатором N, то поток инструкций другого процесса не сможет войти в критическую секцию с тем же идентификатором, и будет ждать, пока первый процесс не выйдет из этой секции. Проблема совместного доступа к памяти: нужно осторожно работать с теми участками памяти, для которых возможно одновременное выполнение записи одним процессором и другой операции (записи или чтения) другим процессором.

Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Проблема медленного обращения к оперативной памяти и ее ограниченного объема возникает из-за того, что процессор работает быстро, а память – медленно, поэтому даже одному процессору приходится ждать загрузки данных из оперативной памяти. Если же процессоров несколько, то им приходится ждать еще дольше. Скорость работы каждого процессора с памятью становится тем меньше, чем большее число процессоров имеется в системе. Кроме того, объем памяти не может быть сделан сколь угодно большим, так как для этого придется увеличивать разрядность шины памяти.

Еще одна проблема заключается в том, что очень сложно сделать масштабируемую систему с большим числом процессоров, так как очень сильно возрастает стоимость и падает эффективность работы из-за описанных выше проблем. Практически все подобные системы имеют не более 8 процессоров.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти, при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти (рис. 1.6 б). Такой подход именуется неоднородным доступом к памяти (non-uniform memory access или NUMA). Среди систем с таким типом памяти выделяют:

– системы, в которых для представления данных используется только локальная кэш-память имеющихся процессоров (cache-only memory architecture или COMA); примерами являются KSR-1 и DDM;

– системы, в которых обеспечивается когерентность локальных кэшей разных процессоров (cache-coherent NUMA или CC-NUMA); среди таких систем: SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000;

– системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (non-cache coherent NUMA или NCC-NUMA); например, система Cray T3E.

Использование распределенной общей памяти (distributed shared memory или DSM) упрощает проблемы создания мультипроцессоров (известны примеры систем с несколькими тысячами процессоров), однако возникающие при этом проблемы эффективного использования распределенной памяти (время доступа к локальной и удаленной памяти может различаться на несколько порядков) приводят к существенному повышению сложности параллельного программирования.

Мультикомпьютеры (многопроцессорные системы с распределенной памятью) уже не обеспечивают общего доступа ко всей имеющейся в системах памяти (no-remote memory access или NORMA). Система содержит несколько процессоров, каждый имеет свою оперативную память. При всей схожести подобной архитектуры с системами с распределенной общей памятью, мультикомпьютеры имеют принципиальное отличие: каждый процессор системы может использовать только свою локальную память, в то время как для доступа к данным, располагаемым на других процессорах, необходимо явно выполнить операции передачи сообщений. Для обеспечения обмена информацией процессоры соединены каналами связи. По характеру связей такие системы делятся на системы с универсальной коммутацией – каждый процессор может передать информацию любому другому процессору и системы с жесткой (фиксированной) коммутацией –

каждый процессор может передать информацию только ограниченному числу других процессоров.

Данный подход применяется при построении двух важных типов многопроцессорных вычислительных систем – массивно-параллельных систем (massively parallel processor или MPP) и кластеров (clusters). Среди представителей первого типа систем – IBM RS/6000 SP2, Intel PARAGON, ASCI Red, транспьютерные системы Parsytec и др.; примерами кластеров являются, например, системы AC3 Velocity и NCSA NT Supercluster.

Системы с распределенной памятью, в которых каждый вычислительный узел представляет собой полноценный компьютер со своей копией операционной системы, называют кластерными. Кластеры обычно представляют собой шкафы с компактными системными блоками, которые соединены друг с другом каналами связи (посредством специальных коммутаторов), передающими данные со скоростью 10 ГБит/сек и более.

Рис. 1.7. Суперкомпьютер Columbia, имеющий 10240 процессоров

К преимуществам Систем с распределенной памятью можно отнести:

– простота и дешевизна построения: можно взять большое количество обычных компьютеров, соединить их каналами связи (например, Ethernet), и получить кластер;

– эффективное решение задач, требующих малого обмена данными: каждый компьютер будет работать в полную мощность, не ожидая, пока освободится доступ к оперативной памяти;

– возможность решать задачи, требующие очень больших объемов оперативной памяти: суммарный объем памяти системы можно сделать сколь угодно большим; требуется лишь, чтобы задача разбивалась на относительно независимые подзадачи;

– возможность масштабирования: можно соединить сколько угодно вычислительных узлов вместе, при этом стоимость системы будет пропорциональна числу узлов.

В связи с этим большинство самых мощных вычислительных систем в мире являются кластерными.

Недостатки:

– проблема обмена данными: обмен данными в таких системах обычно идет очень медленно по сравнению со скоростью вычислений (и с большими задержками), поэтому задачи, требующие интенсивного обмена, невозможно решить на таких системах эффективно;

– сложное программирование, т. к. программист должен продумать обмен данными, который будет присутствовать в системе, должен сам запрограммировать этот обмен (например, с помощью MPI); при неправильном программировании велика вероятность взаимных блокировок, когда, например, два процессора ждут данных друг от друга; проблема блокировок есть и в системах с общей памятью, но здесь она проявляет себя гораздо чаще; автоматическая организация обмена данными возможна лишь для некоторых частных случаев.

– большой размер систем и большое энергопотребление: кластерные системы занимают целые комнаты (рис. 1.6) и даже здания.

Многие современные системы представляют собой иерархию описанных выше систем. Например, современные процессоры являются конвейерными процессорами, и имеют набор векторных инструкций (MMX, SSE и т. п.), позволяющих выполнять одновременные вычисления с разными данными. Кроме того, процессор может иметь два ядра, или может быть несколько процессоров в компьютере. Таким образом, на этом уровне система представляет собой систему с общей памятью. Затем можно соединить несколько таких компьютеров в кластер, образовав новый уровень иерархии: систему с распределенной памятью.

Следует отметить чрезвычайно быстрое развитие многопроцессорных вычислительных систем кластерного типа. Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров либо же сконструированы из типовых компьютерных элементов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени устранить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров. Тем самым, для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов, что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых данных между компьютерами кластера. Вместе с этим следует отметить, что организация взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным временным задержкам, и это накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

Отдельные исследователи обращают особое внимание на отличие понятия кластера от сети компьютеров (network of workstations или NOW). Для построения локальной компьютерной сети, как правило, используют более простые сети передачи данных (порядка 100 Мбит/сек). Компьютеры сети обычно более рассредоточены, и пользователи могут применять их для выполнения каких-либо дополнительных работ.

При разработке параллельных алгоритмов решения сложных научно-технических задач принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого ускорения процесса вычислений (сокращения времени решения задачи). Формирование подобных оценок ускорения может осуществляться применительно к выбранному вычислительному алгоритму (оценка эффективности распараллеливания конкретного алгоритма). Другой важный подход состоит в построении оценок максимально возможного ускорения процесса решения задачи конкретного типа (оценка эффективности параллельного способа решения задачи).

1.1.3. Классификация архитектур вычислительных систем

Основными широко применяемыми архитектурами вычислительных систем с распределенными ресурсами являются многопроцессорные системы, вычислительные кластеры, векторные процессоры, VLIW-процессоры, суперскалярные процессоры и процессоры, ориентированные на конкретную задачу.

1. *Многопроцессорные системы*

Многопроцессорные вычислительные системы представлены двумя типами. Первый, наиболее распространенный — это многопроцессорные серверы с общей памятью. Ведущие производители выпускают многопроцессорные серверы, стремясь предоставить пользователям программное окружение, доступное в среде традиционных однопроцессорных компьютеров. Типичными представителями данного

класса являются системы SMP (symmetric multiprocessors) – симметричные мультипроцессоры, в которых все процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым.

Второй тип многопроцессорных систем – параллельные суперкомпьютеры MPP (massive parallel processing) с большим количеством процессоров и разделяемой памятью. Использующие подобную модель суперкомпьютеры построены как массив отдельных машин (узлов), взаимодействующих через высокоскоростные каналы связи. Каждый узел получает доступ только к локальной памяти, выполняемое параллельное приложение оказывается разделенным на ряд параллельно выполняемых слабо взаимодействующих процессов, обменивающихся информацией путем передачи и приема сообщений.

Системы обоих классов часто строятся из тех же микропроцессоров, на базе которых выпускаются персональные компьютеры и рабочие станции.

2. *Вычислительные кластеры и вычислительные сети*

Вычислительные кластеры представляют собой системы, состоящие из множества узлов, связанных коммуникационной средой. В качестве узлов могут использоваться компьютеры в составе локальной или глобальной сети. Каждый узел имеет локальную память, общей оперативной памяти нет. В составе кластера могут быть узлы с различной архитектурой и производительностью. В случае, когда все узлы кластера имеют одинаковую архитектуру и производительность, кластер называют однородным, иначе – неоднородным.

Архитектура вычислительного кластера подобна суперкомпьютерам MPP, кластеры часто используются в качестве их дешевой альтернативы, поскольку могут быть построены на базе уже имеющихся в организации персональных компьютеров. Любой кластер можно рассматривать как единую аппаратно-программную систему, имеющую единую коммуникационную систему, единый центр управления и планирования

загрузки. Часто в целях уменьшения стоимости кластера в качестве узлов используются доступные в данный момент компьютеры, имеющие разные характеристики, и, возможно, частично загруженные решением других задач.

Распределенные вычислительные системы (GRID) объединяют ресурсы множества кластеров, многопроцессорных и однопроцессорных компьютеров, имеющих различное географическое расположение, принадлежащих разным организациям и подчиняющихся разным дисциплинам использования. Особая разновидность систем данного класса – глобальные вычислительные сети, использующие Интернет в качестве коммуникационной среды.

Программное обеспечение неоднородных кластеров и вычислительных сетей должно обеспечивать адаптацию к динамическому изменению конфигурации, учитывать неоднородность архитектуры и возможность сбоев отдельных узлов.

Особенностью реализации параллельной обработки является то, что обрабатываемая информация должна распределяться между узлами относительно большими порциями, поскольку интервалы времени, необходимые для передачи информации между узлами, обычно на несколько порядков превосходят время передачи в пределах одного узла.

3. Векторные процессоры

Альтернативное название – процессоры SIMD-архитектуры. Имеется один поток команд, содержащий векторные инструкции. Одна векторная инструкция выполняет определенную арифметическую операцию одновременно для всех элементов вектора данных.

Процессоры различных архитектур могут иметь векторные инструкции для ускорения выполнения некоторых операций. Векторные процессоры применяются в некоторых суперкомпьютерах. Недостатком векторных процессоров является их высокая стоимость.

4. *Использование VLIW-технологии*

Процессоры с длинным словом инструкции (Very Long Instruction Word, VLIW) реализуют простейший способ параллельной обработки на уровне отдельных инструкций. Каждая команда VLIW-процессора может состоять из нескольких параллельно исполняемых инструкций, работающих с разными данными.

При использовании VLIW-процессоров распараллеливание алгоритма осуществляется либо компилятором на этапе компиляции программы, либо программистом при программировании на ассемблере. Архитектура процессоров данного типа накладывает существенные ограничения на возможность параллельного исполнения, поскольку в них невозможно создание нескольких параллельно исполняемых потоков инструкций. Фактически существует только один поток инструкций, но каждая из них может содержать несколько команд обработки, выполняемых одновременно.

5. *Суперскалярные процессоры*

Исполняемый код суперскалярных процессоров обычно не содержит информации о параллельной обработке. Распараллеливание инструкций происходит на этапе исполнения программы. Для этого в процессоре присутствует специальный блок анализа. Архитектура таких процессоров значительно сложнее VLIW-архитектуры, поэтому они редко применяются при проектировании встроенных систем.

Наиболее часто такие процессоры используются в качестве более производительных моделей, совместимых с предыдущими, не имевшими возможностей параллельной обработки. Например, процессоры компании Intel, начиная с Pentium, относятся к этому классу. На базе суперскалярных процессоров построены практически все современные персональные компьютеры и многопроцессорные системы.

6. *Процессоры, ориентированные на конкретную задачу*

Разрабатываются специально для решения некоторой задачи. Параллельная обработка может осуществляться на уровне аппаратной реализации некоторых алгоритмов.

Разработка таких процессоров осуществляется с помощью языков описания аппаратуры, например, VHDL или Verilog.

Недостатком такого подхода является сложность проектирования и отладки, а также отсутствие высокоуровневых средств разработки программного обеспечения для полученного процессора.

7. *Сравнение эффективности и возможности применения различных методов построения параллельных систем*

Вычислительные системы с распределенными ресурсами применяются для решения самых различных задач.

Многопроцессорные суперкомпьютеры различных архитектур, вычислительные кластеры и вычислительные сети применяются для решения задач, требующих огромных объемов вычислений.

Многопроцессорные SMP-серверы также широко используются для обработки данных в реальном времени, например, в качестве WEB-серверов.

Другое направление использования параллельной обработки – увеличение производительности отдельных микропроцессоров. Универсальные микропроцессоры, используемые в персональных компьютерах и многопроцессорных системах, имеют суперскалярную архитектуру. В некоторых суперкомпьютерах применяются и векторные процессоры.

Отдельно следует отметить процессоры, используемые во встроенных системах. При выборе процессора для встроенной системы особое внимание уделяется производительности, стоимости и потребляемой энергии. Параллельная обработка часто позволяет наиболее эффективно использовать аппаратные ресурсы. Поэтому во встроенных системах часто применяются

VLIW-процессоры, а также специализированные решения, аппаратно реализованные с использованием ПЛИС или СБИС.

Уровни параллелизма

Распараллеливание операций – перспективный путь повышения производительности вычислений. Согласно закону Мура – число транзисторов экспоненциально растет, что позволяет в настоящее время включать в состав CPU большое количество исполнительных устройств самого разного назначения. Прошли времена, когда функционирование ЭВМ подчинялось принципам фон Неймана.

В 70-е годы стал активно применяться принцип конвейеризации вычислений. Сейчас конвейер Intel Pentium 4 состоит из 20 ступеней. Такое распараллеливание на микроуровне – первый шаг на пути эволюции процессоров. На принципах конвейеризации базируются и внешние устройства. Например, динамическая память (организация чередования банков) или внешняя память (организация RAID).

Использование микроуровневого параллелизма позволяло лишь уменьшать CPI (Cycles Per Instruction), так как миллионы транзисторов при выполнении одиночной инструкции простаивали. На следующем этапе эволюции в 80-е годы стали использовать параллелизм уровня команд посредством размещения в CPU сразу нескольких конвейеров. Такие суперскалярные CPU позволяли достигать CPI<1. Параллелизм уровня инструкций (ILP) породил неупорядоченную модель обработки, динамическое планирование, станции резервации и т. д. От CPI перешли к IPC (Instructions Per Clock). Но ILP ограничен алгоритмом исполняемой программы. Кроме того, при увеличении количества ALU сложность оборудования экспоненциально растет, увеличивается количество горизонтальных и вертикальных потерь в слотах выдачи. Параллелизм уровня инструкций исчерпал свои резервы, а тенденции Мура позволили процессоростроителям осваивать более высокие уровни параллелизма.

Современные методики повышения ILP основаны на использовании процессоров класса SIMD – это векторное процессирование, матричные процессоры, архитектура VLIW.

Параллелизм уровня потоков и уровня заданий применяется в процессорах класса MIMD. Многопоточные процессоры позволяют снижать вертикальные потери в слотах выдачи, а Simultaneous Multithreading (SMT) процессоры – как вертикальные, так и горизонтальные потери. Закон Мура обусловил также выпуск многоядерных процессоров (CMP). Лучшие современные вычислители – это мультикомпьютерные мультипроцессорные системы.

Параллелизм всех уровней свойственен не только процессорам общего назначения (GPP), но и процессорам специального назначения (ASP (Application-Specific Processor), DSP (Digital Signal Processor)).

Иногда классифицируют параллелизм по степени гранулярности как отношение объема вычислений к объему коммуникаций. Различают мелкозернистый, среднезернистый и крупнозернистый параллелизм (рис. 1.7).

Параллелизм заданий – каждый процессор загружается своей собственной независимой от других вычислительной задачей. Параллелизм такого типа представляет интерес скорее для системных администраторов, чем рядовых пользователей.

Параллелизм на уровне программы – вычислительная программа разбивается на несколько достаточно больших частей, которые могут выполняться одновременно на различных процессорах (подпрограммы, независимые ветви программ).

Параллелизм циклов и итераций – в большинстве случаев это векторная обработка, иногда (при независимости по данным последующих шагов) – параллельная обработка.

Параллелизм команд – обычно реализован на низком уровне, например внутри процессора (конвейеры команд и т. д.).

Параллелизм на уровне машинных слов и арифметических операций – в некоторых ситуациях (например, сложение двух операндов) выполняется одновременным сложением всех их двоичных разрядов.

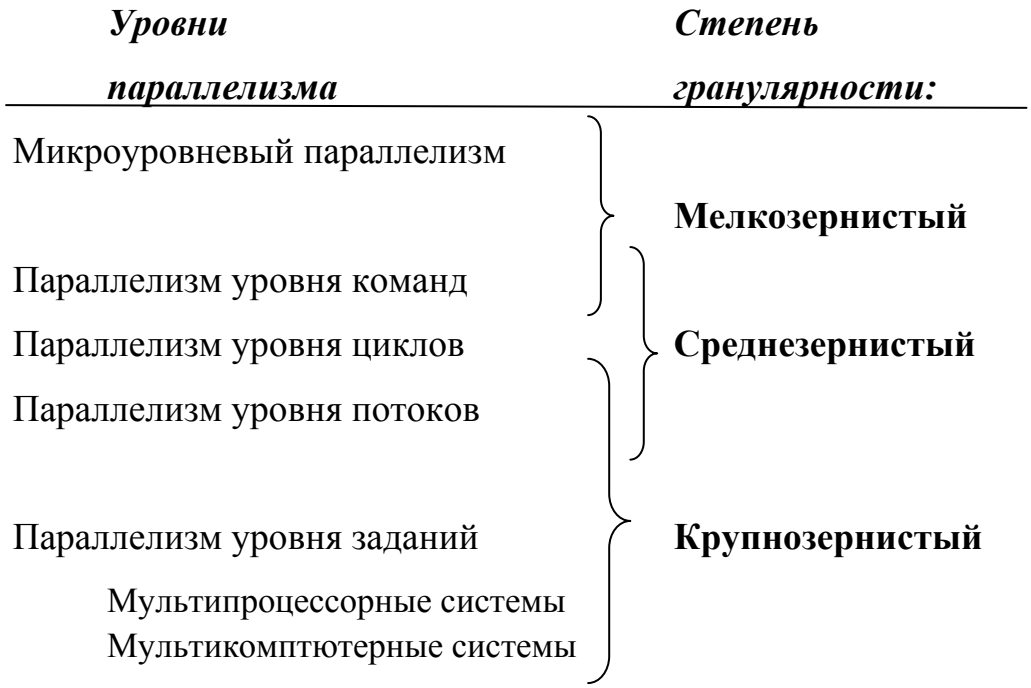


Рис. 1.8. Уровни параллелизма

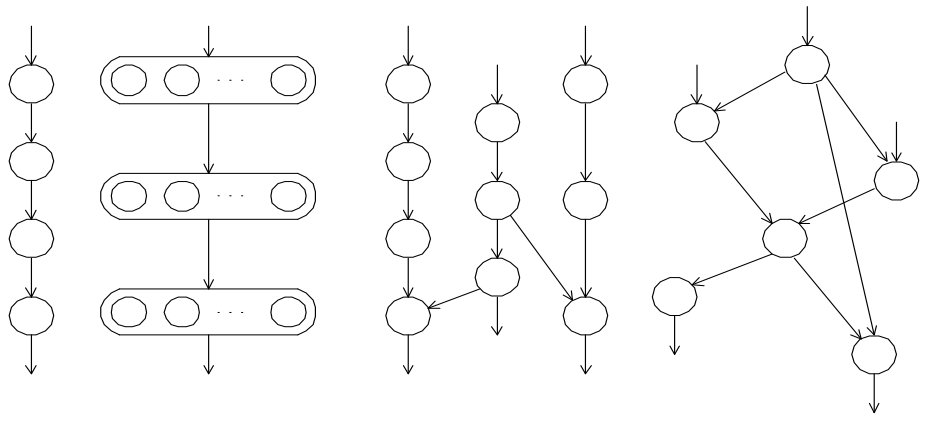
Мелкозернистый параллелизм обеспечивает сам CPU, но компилятор может и должен ему помочь для обеспечения большего IPC. Среднезернистый параллелизм – прерогатива программиста, которому необходимо разрабатывать многопоточные алгоритмы. Здесь роль компилятора заключается в выборе оптимальной последовательности инструкций (с большим IPC) посредством различных методик (например, символическое разворачивание циклов). Крупнозернистый параллелизм обеспечивает ОС.

Вложенность этих уровней определяет глубину распараллеливания и является одним из важнейших свойств при анализе параллельных вычислений. Поскольку концепция алгоритма подразумевает иерархию (любая вычислительная конструкция в свою очередь может быть описана в виде алгоритма, состоящего из еще более «простых» вычислительных конструкций) – это приводит к понятию детализации параллелизма. Детализацию называют мелкой, если вычислительные конструкции

алгоритма являются примитивными (т. е. реализуемыми одной командой), и крупной, если эти конструкции являются сложными (т. е. реализуются с помощью конструкций более низкого уровня). Спектр значений детализации простирается от очень мелкой до очень крупной. Соответственно вычислительные системы могут обладать мелко-, средне- или крупноблочной структурой.

Алгоритмические структуры отличаются друг от друга содержанием простых вычислительных конструкций и типом отношения следования. Рассмотрим четыре основных типа отношений следования вычислительных конструкций и соответствующие им типы логической организации систем.

В обычной последовательной алгоритмической структуре (рис. 1.9 а) в каждый момент времени исполняется только одна команда. Соответствующая вычислительная машина проста: в ней имеется единственная память для хранения данных и программ, одно арифметическое устройство, исполняющее текущую команду, одно устройство управления, осуществляющее контроль за исполнением, счетчик команд, хранящий адрес текущей команды, и простой механизм его модификации. Взаимные связи между перечисленными компонентами также просты. Быстродействие системы ограничено ее последовательной сущностью и определяется временем исполнения каждой команды.



а) *б)* *в)* *г)*

Рис. 1.9. Алгоритмические структуры:
а – последовательная; б – последовательно групповая; в – совокупность слабосвязанных потоков; г – параллельная структура общего вида

Важное преимущество последовательного подхода состоит в том, что для машин данного типа разработано огромное количество алгоритмов и программ, накоплен богатый опыт программирования, разработаны фундаментальные языки и технологии программирования. По этой причине дальнейшее совершенствование организации шло так, что последовательная структура сохранялась, однако к ней добавлялись механизмы, позволяющие исполнять несколько команд одновременно. Все эти механизмы так или иначе сводятся к общему принципу опережающего просмотра команд, поскольку для того, чтобы иметь возможность одновременно исполнять несколько команд, процессор должен просматривать поток команд на несколько шагов вперед и находить те команды, которые допускают одновременное выполнение. Выявление параллелизма в этом случае сводится к тому, чтобы исключить те зависимости, которые привнесены в алгоритм в процессе его последовательной записи, и сохранить только те зависимости, которые обусловлены самой природой алгоритма. Такие зависимости бывают двух типов: зависимости по данным (когда команда не может исполниться до тех пор, пока не подготовлены все операнды, которые являются результатами исполнения других команд) и зависимости по управлению (когда условный переход не может быть осуществлен до тех пор, пока не вычислено условное выражение).

Для одновременного выполнения команд, не являющихся зависимыми, существуют две дополнительные возможности: использование нескольких процессоров (по числу команд) и применение одного процессора конвейерного типа.

Конвейеризация обработки – это общий метод повышения пропускной способности систем, выполняющих повторяющиеся операции. Основа этого подхода состоит в том, что система может быть разделена на ступени обрабатывающих устройств, позволяющих начинать исполнение новой команды прежде, чем будет завершена предыдущая. В такой системе пропускная способность определяется временем прохождения самого

медленного звена. Данный подход может быть применен, например, к обработке команд, поскольку процесс исполнения команды состоит из нескольких стадий, таких как вызов команды, расшифровка, вычисление адреса, вызов операнда и др.

Факторы, от которых зависит ускорение вычислений за счет опережающего просмотра, следующие:

1. Степень параллелизма, заключенного в программе. В типовых программах число зависимостей по данным невелико. Это создает широкие предпосылки для использования параллелизма. Но с другой стороны, число зависимостей по управлению, соответствующих условным переходам, весьма значительно, и оно накладывает ограничения на возможность использования параллелизма. В некоторых вычислительных машинах имеется механизм, который не останавливается на условных переходах, а продолжает опережающий просмотр по обеим ветвям или по предпочтительной ветви, определяемой с помощью механизма прогнозирования перехода. Этот метод требует сложного дополнительного оборудования в устройстве управления, и в то же время имеет ограниченный эффект из-за высокой вероятности того, что еще до завершения предыдущего ветвления появится новая ветвь.

2. Способность устройства управления обнаруживать зависимости, но если поставлена задача обнаружить все или почти все операции, допускающие одновременное исполнение, то устройство управления получается довольно сложным.

3. Наличие конвейерного процессора и/или набора функциональных устройств, предназначенных для одновременного исполнения нескольких команд.

4. Наличие памяти с широким трактом доступа, способным обеспечить поставку команд и данных процессору в таком темпе, чтобы он оказывался постоянно активным.

Поскольку быстродействие памяти обычно ниже, чем процессора, то для организации широкого тракта доступа необходимо разбить память на

блоки, обращение к которым может происходить одновременно. При этом максимальная пропускная способность тракта, связывающего процессор с памятью, зависит от быстродействия блоков и их общего количества. Однако, поскольку возможны конфликты по доступу, пропускная способность оказывается зависящей и от других факторов, таких как механизм адресации, адресный поток, а также возможность приоритетного обслуживания запросов.

Еще один подход к уменьшению отрицательного влияния на производительность зависимостей по данным и по управлению состоит в том, что процессор работает как бы в мультиплексном (по времени) режиме, обслуживая несколько процессов. Этот принцип похож на используемый в мультипрограммировании, с той лишь разницей, что в данном случае, мультиплексирование осуществляется с квантом, равным времени выполнения одной команды.

В последовательно-групповой структуре вычислительные конструкции, образующие алгоритм, объединены в группы, а отношение следования состоит в том, что конструкции внутри группы могут выполняться одновременно, а сами группы последовательно (рис. 1.9 б). Вычислительные конструкции внутри одной группы могут быть одинаковыми или разными. Реализация таких алгоритмов на высокопроизводительных скалярных процессорах с помощью обычных циклов сопровождается рядом факторов, ограничивающих максимальную скорость вычислений:

– перед каждой скалярной операцией необходимо вызвать и декодировать скалярную команду;

– для каждой команды необходимо вычислять адреса элементов данных;

– данные должны вызываться из памяти, а результаты запоминаться в памяти;

– необходимо осуществлять упорядочение выполнения операций в функциональных устройствах, которые в целях увеличения производительности строятся по конвейерному принципу;

– реализация команд построения циклов (счетчик и переход) сопровождается накладными расходами.

Влияние этих факторов уменьшается при введении векторных команд, с помощью которых задается одна и та же операция над элементами одного или нескольких векторов, и организации системы, которая обеспечивает эффективное исполнение таких команд. Этот подход реализуется в системах двух типов: матричных и векторно-конвейерных. В векторно-конвейерных системах вычислительные конструкции внутри групп исполняются конвейерным процессором, в то время как в матричных системах для этого используются несколько одновременно работающих процессоров.

Оба подхода позволяют достичь значительного ускорения по сравнению со скалярными машинами. Более того, ускорение в системах матричного типа может быть больше, чем в конвейерных, поскольку увеличить число процессорных элементов проще, чем число ступеней в конвейерном устройстве.

Для современного уровня технологии векторно-конвейерные системы являются более гибкими и эффективными с точки зрения стоимости.

В самом широком смысле векторизация – это преобразование операций, выполняемых по ходу процесса решения задачи, из скалярной традиционной формы в векторную.

Наиболее общим типом векторизации является синтаксическая векторизация, т. е. такой способ преобразования последовательности команд, при котором не учитывается смысловое значение команд и данных, а учитывается только форма. Самая сильная сторона синтаксической векторизации одновременно является и ее самой слабой. С одной стороны, поскольку учитывается только форма представления программ и данных, имеется теоретическая возможность реализовать процесс векторизации с

помощью специальных программных средств – так называемых автоматических векторизаторов. Но с другой стороны, что же делать, если эти программные векторизаторы не найдут формального преобразования, позволяющего преобразовать некоторую программно-информационную структуру в векторную?

Для того, чтобы избежать данной ситуации, необходимо обратиться к другому подходу – семантической векторизации. Семантическая векторизация подразумевает интерпретацию содержащихся в программах структур данных и подстановку векторных конструкций, выполняемых с учетом смыслового содержания задачи, а не ее формы.

В широком смысле существуют два препятствия на пути семантической векторизации. Во-первых, структура данных, над которой должна быть произведена та или иная операция, может и не быть вектором. Это может происходить по разным причинам, начиная от постоянного присутствия «небольших» дыр в структурах данных до сильно разреженных структур. Второе препятствие – это рекурсивные вычисления. Ключевое условие реализации векторной или параллельной обработки состоит в том, чтобы элементы, участвующие в одних и тех же операциях, не взаимодействовали между собой при исполнении этих операций. Рекурсивные отношения делают это невозможным.

Таким образом, с точки зрения архитектуры вычислительных систем векторная обработка представляет собой совершенно особый эволюционный этап в развитии параллельных систем и вычислений, и соответственно, требует оборудования, обладающего большими логическими возможностями.

Следующий путь ускорения вычислений за счет параллельного выполнения операций – это использование алгоритмов, структура которых представляет собой совокупность слабосвязанных потоков команд (последовательных процессов). В этом случае программа распадается на несколько последовательных процессов, между которыми существует

относительно небольшое число взаимосвязей (рис. 1.9 в). Каждый процесс может исполняться на отдельном последовательном процессоре, который при необходимости осуществляет взаимодействие с другими процессорами. Такие вычислительные системы называются многопроцессорными, и ключевым элементом в них является механизм синхронизации и взаимосвязи между процессами.

2. Метрики параллелизма

2.1. Моделирование и анализ параллельных вычислений

Для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач может быть использована модель в виде графа «операции – операнды». Для уменьшения сложности излагаемого материала при построении модели будет предполагаться, что время выполнения любых вычислительных операций является одинаковым и равняется 1 (в тех или иных единицах измерения). Кроме того, принимается, что передача данных между вычислительными устройствами выполняется мгновенно без каких-либо затрат времени (что может быть справедливо, например, при наличии общей разделяемой памяти в параллельной вычислительной системе).

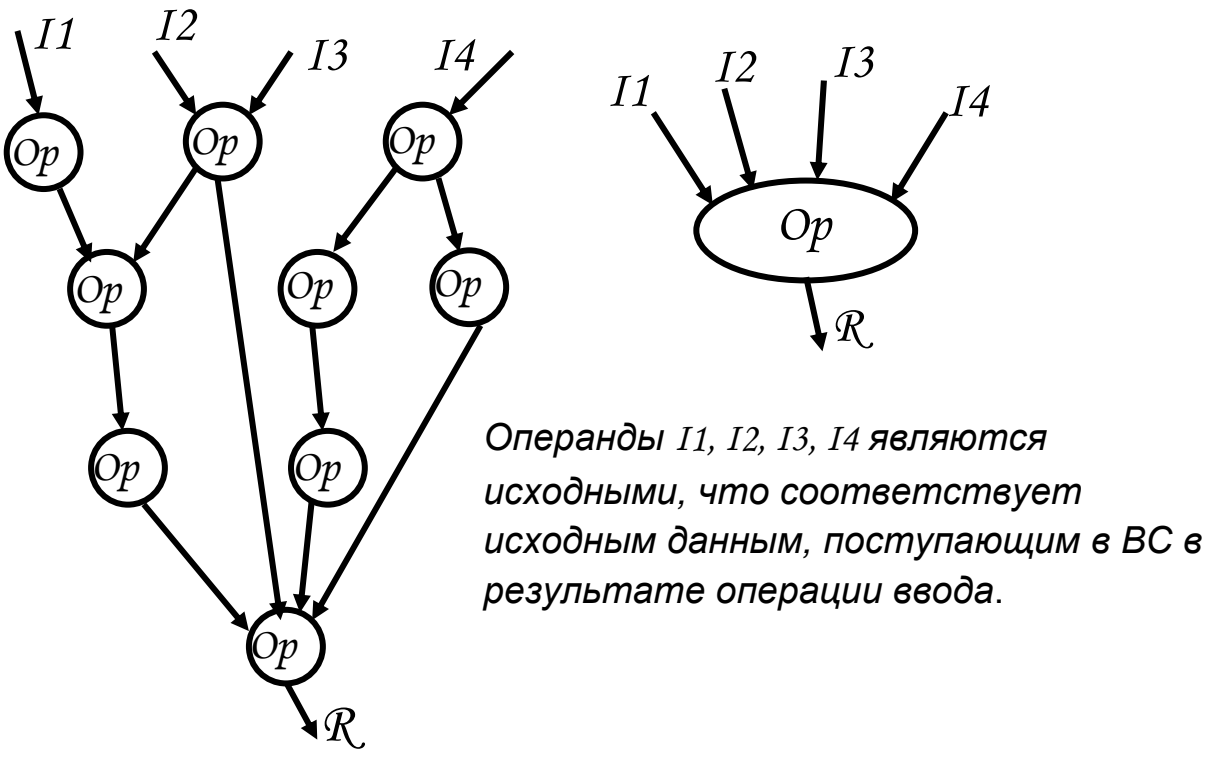


Рис. 2.1. Пример вычислительной модели алгоритма в виде графа «операции – операнды» (общий вид)

Представим множество операций, выполняемых в исследуемом алгоритме решения вычислительной задачи, и существующие между

операциями информационные зависимости в виде ациклического ориентированного графа G = (V, R), где V = {1,...,|V|} – множество вершин графа, представляющих выполняемые операции алгоритма, а R – множество дуг графа (при этом дуга r = (i, j) принадлежит графу только в том случае, если операция j использует результат выполнения операции i). Для примера на рисунке 1 показан граф алгоритма вычисления площади прямоугольника, заданного координатами двух противолежащих углов.

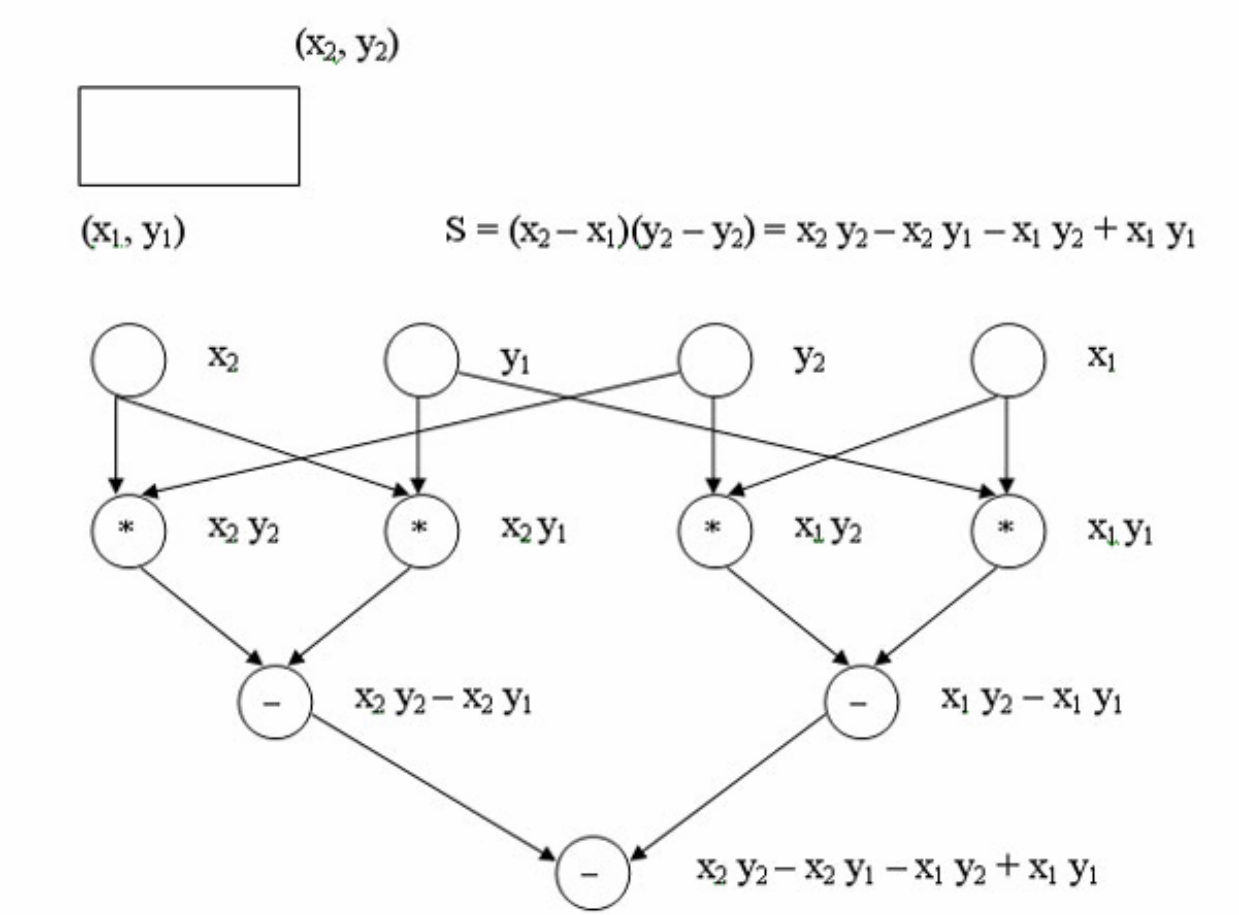


Рис. 2.2. Пример вычислительной модели алгоритма в виде графа «операции – операнды» (вычисление площади прямоугольника)

Как можно заметить по приведенному примеру, для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений и построены соответственно разные вычислительные модели. Как будет показано далее, разные схемы вычислений обладают различными возможностями для распараллеливания и, тем самым, при построении модели вычислений может быть поставлена задача выбора

наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.

В рассматриваемой вычислительной модели алгоритма вершины без входных дуг могут использоваться для задания операций ввода, а вершины без выходных дуг – для операций вывода. Обозначим через V множество вершин графа без вершин ввода, а через d(G) – диаметр (длину максимального пути) графа.

Операции алгоритма, между которыми нет пути в рамках выбранной схемы вычислений, могут быть выполнены параллельно (для вычислительной схемы на рисунке 1 предыдущего шага, например, параллельно могут быть реализованы сначала все операции умножения, а затем первые две операции вычитания). Возможный способ описания параллельного выполнения алгоритма может состоять в следующем.

Пусть P есть количество процессоров, используемых для выполнения алгоритма. Тогда для параллельного выполнения вычислений необходимо задать множество (расписание) $H_P = \{(i,P_i,t_i)$: i принадлежит V$\}$, в котором для каждой операции i, принадлежащей V, указывается номер используемого для выполнения операции процессора P_i и время начала выполнения операции t_i. Для того чтобы расписание было реализуемым, необходимо выполнение следующих требований при задании множества H_p:

Для любых i и j, принадлежащих V, если $t_i = t_j$, то P_i не равно P_j, то есть один и тот же процессор не должен назначаться разным операциям в один и тот же момент.

Для любых (i,j), принадлежащих R, $t_j >= t_i+1$, то есть к назначаемому моменту выполнения операции все необходимые данные уже должны быть вычислены.

Можно рассмотреть модели запуска программ на общей и на распределенной памяти.

При запуске программы на распределенной памяти указывается название программы и количество процессоров, которое требуется получить

в распоряжение. После запуска программы выделяется некоторое количество процессорных узлов, число которых может совпадать или не совпадать с запрашиваемым. На каждом процессоре запускается своя копия программы, и у каждой копии программы будет своя оперативная память. Разные копии программы, запущенные на разных процессорных узлах, не имеют доступа к внутренним переменным других копий программы. Каждая копия программы получает две переменные – число процессоров, которое было запрошено, и номер процессора. Обмен данными осуществляется с использованием номера процессора-получателя.

При запуске программы на общей памяти фактически осуществляется запуск одной копии программы. Когда требуется задействовать другие процессоры, имеющиеся в системе, осуществляется вызов функции порождения дополнительной нити кода, и тогда некоторые вычисления будут выполняться в параллельном потоке. Можно сделать число нитей равным числу процессоров в системе, можно сделать число нитей больше, чем число процессоров, тогда часть нитей будет выполняться уже не в параллельном, а в последовательном конкурентном режиме (режиме разделения времени). Для каждой нити порождается свой стек, каждая нить имеет свои локальные переменные, но все нити имеют доступ к глобальным переменным.

Вычислительная схема алгоритма G совместно с расписанием H_p может рассматриваться как модель параллельного алгоритма $A_p(G,H_p)$, исполняемого с использованием p процессоров. Время выполнения параллельного алгоритма определяется максимальным значением времени, применяемым в расписании

$$\mathbf{T_p(G, H_p) = \max_{i \in V}(t_i + 1).}$$

Для выбранной схемы вычислений желательно использование расписания, обеспечивающего минимальное время исполнения алгоритма

$$\mathbf{T_p(G) = \min_{Hp} T_p(G, H_p).}$$

Уменьшение времени выполнения может быть обеспечено и путем подбора наилучшей вычислительной схемы

$$\mathbf{T_p = \min_G T_p(G).}$$

Оценки $T_p(G,H_p)$, $T_p(G)$ и T_p могут быть применены в качестве показателей времени выполнения параллельного алгоритма. Кроме того, для анализа максимально возможного параллелизма можно определить оценку наиболее быстрого исполнения алгоритма

$$\mathbf{T_\infty = \min_{p\geq 1} T_p.}$$

Оценку T с бесконечностью в качестве индекса можно рассматривать как минимально возможное время выполнения параллельного алгоритма при использовании неограниченного количества процессоров (концепция вычислительной системы с бесконечным количеством процессоров, обычно называемой паракомпьютером, широко применяется при теоретическом анализе параллельных вычислений).

Оценка T_1 определяет время выполнения алгоритма при использовании одного процессора и представляет, тем самым, время выполнения последовательного варианта алгоритма решения задачи. Построение подобной оценки является важной задачей при анализе параллельных алгоритмов, поскольку она необходима для определения эффекта использования параллелизма (ускорения времени решения задачи). Очевидно, что

$$\mathbf{T_1(G) = |\bar{V}|,}$$

где модуль V с чертой есть количество вершин вычислительной схемы без вершин ввода. Важно отметить, что если при определении оценки

ограничиться рассмотрением только одного выбранного алгоритма решения задачи и использовать величину

$$\mathbf{T_1 = \min_G T_1(G),}$$

то получаемые при такой оценке показатели ускорения будут характеризовать эффективность распараллеливания выбранного алгоритма. Для оценки эффективности параллельного решения исследуемой вычислительной задачи время последовательного решения следует определять с учетом различных последовательных алгоритмов, т. е. использовать величину

$$T_1^* = \min T_1,$$

где операция минимума берется по множеству всех возможных последовательных алгоритмов решения данной задачи.

Далее приводятся без доказательства теоретические положения, характеризующие свойства оценок времени выполнения параллельного алгоритма.

Теорема 1

Минимально возможное время выполнения параллельного алгоритма определяется длиной максимального пути вычислительной схемы этого алгоритма, т. е.

$$\mathbf{T_\infty(G) = d(G).}$$

Теорема 2

Пусть для некоторой вершины вывода в вычислительной схеме алгоритма существует путь из каждой вершины ввода. Кроме того, пусть входная степень вершин схемы (количество входящих дуг) не превышает двух. Тогда минимально возможное время выполнения параллельного алгоритма ограничено снизу значением

$$T_\infty(G) = \log_2 n,$$

где n есть количество вершин ввода в схеме алгоритма.

Теорема 3

При уменьшении числа используемых процессоров время выполнения алгоритма увеличивается пропорционально величине уменьшения количества процессоров, т. е.

$$\forall q = cp, \qquad 0 < c < 1 \Rightarrow T_p \le cT_q.$$

Теорема 4

Для любого количества используемых процессоров справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма

$$\forall p \Rightarrow T_p < T_\infty + T_1/p.$$

Теорема 5

Времени выполнения алгоритма, которое сопоставимо с минимально возможным временем T с бесконечностью в качестве индекса, можно достичь при количестве процессоров порядка p~T1/T с бесконечностью в качестве индекса, а именно,

$$p \ge T_1/T_\infty \Rightarrow T_p \le 2T_\infty.$$

При меньшем количестве процессоров время выполнения алгоритма не может превышать более чем в 2 раза наилучшее время вычислений при имеющемся числе процессоров, т. е.

$$p < T_1/T_\infty \Rightarrow T_1/p \le T_p \le 2\,T_1/p.$$

Приведенные утверждения позволяют дать следующие рекомендации по правилам формирования параллельных алгоритмов:

– при выборе вычислительной схемы алгоритма должен использоваться граф с минимально возможным диаметром (теорема 1);

– для параллельного выполнения целесообразное количество процессоров определяется величиной p~T1/T с бесконечностью в качестве индекса (теорема 5);

– время выполнения параллельного алгоритма ограничивается сверху величинами, приведенными в теоремах 4 и 5.

Доказательство теоремы 4

Пусть H_∞ есть расписание для достижения минимально возможного времени выполнения T_∞. Для каждой итерации τ, $0 \le \tau \le T_\infty$, выполнения расписания H_∞ обозначим через n_τ количество операций, выполняемых в ходе итерации τ. Расписание выполнения алгоритма с использованием p процессоров может быть построено следующим образом. Выполнение алгоритма разделим на T_∞ шагов; на каждом шаге τ следует выполнить все n_τ операций, которые выполнялись на итерации τ расписания H_∞. Выполнение этих операций может быть выполнено не более, чем за n_τ/p итераций при использовании p процессоров. Как результат, время выполнения алгоритма T_p может быть оценено следующим образом

$$T_p = \sum_{\tau=1}^{T_\infty} \frac{n_\tau}{p} < \sum_{\tau=1}^{T_\infty} \left(\frac{n_\tau}{p} + 1\right) = \frac{T_1}{p} + T_\infty .$$

Доказательство теоремы дает практический способ построения расписания параллельного алгоритма. Первоначально может быть построено расписание без учета ограниченности числа используемых процессоров (расписание для паракомпьютера). Затем, согласно схеме вывода теоремы, может быть построено расписание для конкретного количества процессоров.

Ускорение, получаемое при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений определяется величиной

$$S_p(n)=T_1(n)/T_p(n),$$

т. е. как отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма (величина n применяется для параметризации вычислительной сложности решаемой задачи и может пониматься, например, как количество входных данных задачи).

Эффективность использования параллельным алгоритмом процессоров при решении задачи определяется соотношением

$$E_p(n)=T_1(n)/(pT_p(n))=S_p(n)/p.$$

Величина эффективности определяет среднюю долю времени выполнения алгоритма, в течение которой процессоры реально задействованы для решения задачи.

Из приведенных соотношений можно показать, что в наилучшем случае $S_p(n)=p$ и $E_p(n)=1$. При практическом применении данных показателей для оценки эффективности параллельных вычислений следует учитывать два важных момента:

При определенных обстоятельствах ускорение может оказаться больше числа используемых процессоров $S_p(n)>p$ – в этом случае говорят о существовании сверхлинейного ускорения. Несмотря на парадоксальность таких ситуаций (ускорение превышает число процессоров), на практике сверхлинейное ускорение может иметь место. Одной из причин такого явления может быть неодинаковость условий выполнения последовательной и параллельной программ. Например, при решении задачи на одном процессоре оказывается недостаточно оперативной памяти для хранения всех обрабатываемых данных и тогда становится необходимым использование более медленной внешней памяти (в случае же использования нескольких процессоров оперативной памяти может оказаться достаточно за счет разделения данных между процессорами). Еще одной причиной сверхлинейного ускорения может быть нелинейный характер зависимости сложности решения задачи от объема обрабатываемых данных. Так, например, известный алгоритм пузырьковой сортировки характеризуется

квадратичной зависимостью количества необходимых операций от числа упорядочиваемых данных. Как результат, при распределении сортируемого массива между процессорами может быть получено ускорение, превышающее число процессоров. Источником сверхлинейного ускорения может быть и различие вычислительных схем последовательного и параллельного методов.

При внимательном рассмотрении можно обратить внимание, что попытки повышения качества параллельных вычислений по одному из показателей (ускорению или эффективности) могут привести к ухудшению ситуации по другому показателю, ибо показатели качества параллельных вычислений являются часто противоречивыми. Так, например, повышение ускорения обычно может быть обеспечено за счет увеличения числа процессоров, что приводит, как правило, к падению эффективности. И наоборот, повышение эффективности достигается во многих случаях при уменьшении числа процессоров (в предельном случае идеальная эффективность $E_p(n)=1$ легко обеспечивается при использовании одного процессора). Как результат, разработка методов параллельных вычислений часто предполагает выбор некоторого компромиссного варианта с учетом желаемых показателей ускорения и эффективности.

При выборе надлежащего параллельного способа решения задачи может оказаться полезной оценка стоимости вычислений, определяемой как произведение времени параллельного решения задачи и числа используемых процессоров $C_p=pT_p$.

В связи с этим можно определить понятие стоимостно-оптимального параллельного алгоритма как метода, стоимость которого является пропорциональной времени выполнения наилучшего последовательного алгоритма.

Оценка качества параллельных вычислений предполагает знание наилучших (максимально достижимых) значений показателей ускорения и эффективности, однако получение идеальных величин $S_p=p$ для ускорения и

E_p=1 для эффективности может быть обеспечено не для всех вычислительно трудоемких задач.

Для демонстрации ряда проблем, возникающих при разработке параллельных методов вычислений, можно рассмотреть сравнительно простую задачу нахождения частных сумм последовательности числовых значений:

$$S_k = \sum_{i=1}^{k} x_i \qquad \mathbf{1 \le k \le n,}$$

где n есть количество суммируемых значений (данная задача известна также под названием prefix sum problem).

Изучение возможных параллельных методов решения данной задачи начинается с более простого варианта ее постановки – с задачи вычисления общей суммы имеющегося набора значений (в таком виде задача суммирования является частным случаем общей задачи редукции)

$$S = \sum_{i=1}^{k} x_i$$

Традиционный, последовательный алгоритм для решения этой задачи состоит в последовательном суммировании элементов числового набора

S=0,

S=S+x1,...

Вычислительная схема данного алгоритма может быть представлена следующим образом (Рис. №№):

G1=(V1,R1),

где V1={v_{01},...,v_{0n}, v_{11},...,v_{1n}} есть множество операций (вершины v_{01},...,v_{0n} обозначают операции ввода, каждая вершина v_{1i}, 1_{in}, соответствует прибавлению значения x_i к накапливаемой сумме S), а

R1={(v_{0i},v_{1i}),(v_{1i},v_{1i+1}), 1≤ i ≤ n–1}

есть множество дуг, определяющих информационные зависимости операций.

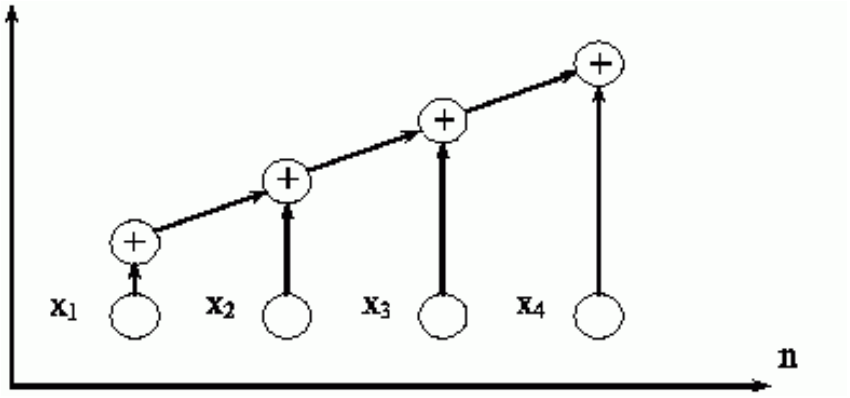


Рис. 2.3. Последовательная вычислительная схема алгоритма суммирования

Как можно заметить, данный «стандартный» алгоритм суммирования допускает только строго последовательное исполнение и не может быть распараллелен.

Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования (известный в литературе как каскадная схема) состоит в следующем (рис. 2.4):

– на первой итерации каскадной схемы все исходные данные разбиваются на пары, и для каждой пары вычисляется сумма их значений;

– далее все полученные суммы также разбиваются на пары, и снова выполняется суммирование значений пар и т. д.

Данная вычислительная схема может быть определена как граф (пусть $n=2^k$)

$$\mathbf{G_2(V_2,R_2),}$$

где $V_2=\{(v_{i1},...,v_{li}),\ 0 \le I \le k,\ 1 \le l_i \le 2^{-1}n\}$ есть вершины графа ($(v_{01},...v_{0n})$ – операции ввода, $(v_{1l},...,v_{1n/2})$ - операции суммирования первой итерации и т. д.), а множество дуг графа определяется соотношениями:

$$\mathbf{R_2=\{(v_{i-1,2j-1}v_{ij}),(v_{i-1,2j}v_{ij}),\ 1 \le i \le k,\ 1 \le j\ 2^{-i}n\}.}$$

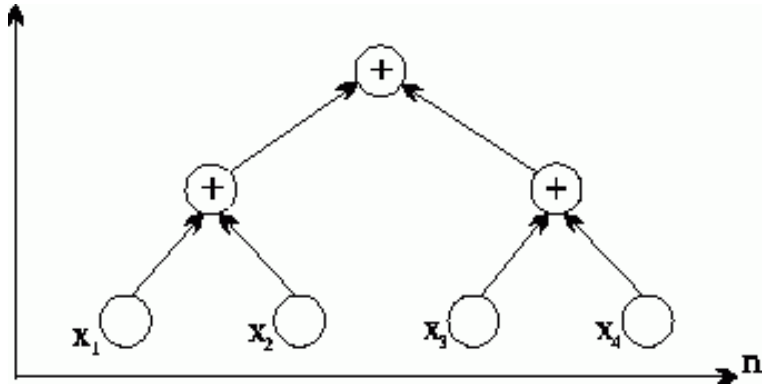


Рис. 2.4. Каскадная схема алгоритма суммирования

Как нетрудно оценить, количество итераций каскадной схемы оказывается равным величине

$$k=\log_2 n,$$

а общее количество операций суммирования

$$K_{посл}=n/2+n/4+...+1=n-1$$

совпадает с количеством операций последовательного варианта алгоритма суммирования. При параллельном исполнении отдельных итераций каскадной схемы общее количество параллельных операций суммирования является равным

$$K_{пар}=\log_2 n.$$

Поскольку считается, что время выполнения любых вычислительных операций является одинаковым и единичным, то $T_1=K_{посл}$, $T_p=K_{пар}$, поэтому показатели ускорения и эффективности каскадной схемы алгоритма суммирования можно оценить как

$$S_p=T_1/T_p=(n-1)/\log_2 n,$$

$$E_p=T_1/pT_p=(n-1)/(p\log_2 n)=(n-1)/((n/2)\log_2 n),$$

где $p=n/2$ есть необходимое для выполнения каскадной схемы количество процессоров.

Анализируя полученные характеристики, можно отметить, что время параллельного выполнения каскадной схемы совпадает с оценкой для паракомпьютера в теореме 2. Однако при этом эффективность использования

процессоров уменьшается при увеличении количества суммируемых значений

$$\lim E_p \to 0 \text{ при } n \to \infty.$$

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы. Для упрощения построения оценок можно предположить $n=2^k$, $k=2^S$. Тогда в новом варианте каскадной схемы все вычисления производятся в два последовательно выполняемых этапа суммирования (рис. 2.5):

– на первом этапе вычислений все суммируемые значения подразделяются на $(n/\log_2 n)$ групп, в каждой из которых содержится log2n элементов. Далее для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования; вычисления в каждой группе могут выполняться независимо друг от друга (т. е. параллельно – для этого необходимо наличие не менее $(n/\log_2 n)$ процессоров);

– на втором этапе для полученных $(n/\log_2 n)$ сумм отдельных групп применяется обычная каскадная схема.

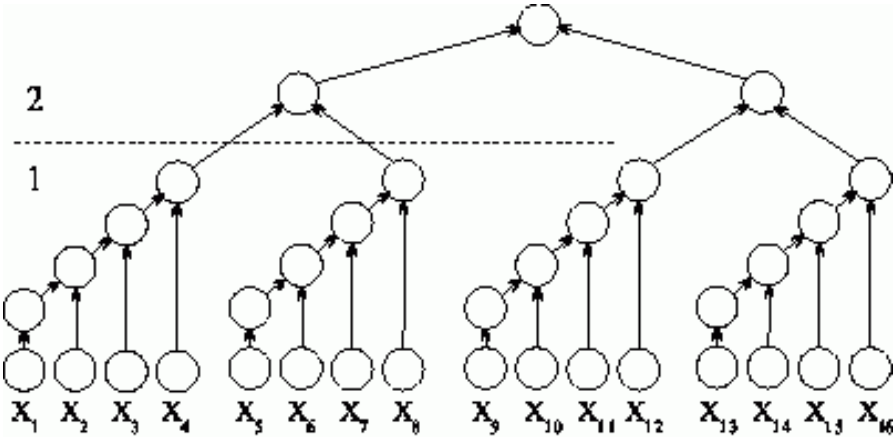


Рис. 2.5. Модифицированная каскадная схема суммирования

Тогда для выполнения первого этапа требуется $\log_2 n$ параллельных операций при использовании $p_1=(n/\log_2 n)$ процессоров.

Для выполнения второго этапа необходимо

$$\mathbf{\log_2(n/\log_2 n)\log_2 n}$$

параллельных операций для $p_2=(n/\log_2 n)/2$ процессоров. Как результат, данный способ суммирования характеризуется следующими показателями:

$$\mathbf{T_p=2\log_2 n,\ p=(n/\log_2 n).}$$

С учетом полученных оценок показатели ускорения и эффективности модифицированной каскадной схемы определяются соотношениями:

$$\mathbf{S_p=T_1/T_p=(n-1)/2\log_2 n,}$$

$$\mathbf{E_p=T_1/pT_p=(n-1)/(2(n/\log_2 n)\log_2 n)=(n-1)/2n.}$$

Сравнивая данные оценки с показателями обычной каскадной схемы, можно отметить, что ускорение для предложенного параллельного алгоритма уменьшилось в 2 раза, однако для эффективности нового метода суммирования можно получить асимптотически ненулевую оценку снизу

$$\mathbf{E_p = (n\text{-}1)/2n \geq 0{,}25 \lim E_p \to 0{,}5 \text{ при } n \to \infty.}$$

Можно отметить также, что данные значения показателей достигаются при количестве процессоров, определенном в теореме 5. Кроме того, необходимо подчеркнуть, что, в отличие от обычной каскадной схемы, модифицированный каскадный алгоритм является стоимостно-оптимальным, поскольку стоимость вычислений в этом случае

$$\mathbf{C_p=pT_p=(n/\log_2 n)(2\log_2 n)}$$

является пропорциональной времени выполнения последовательного алгоритма.

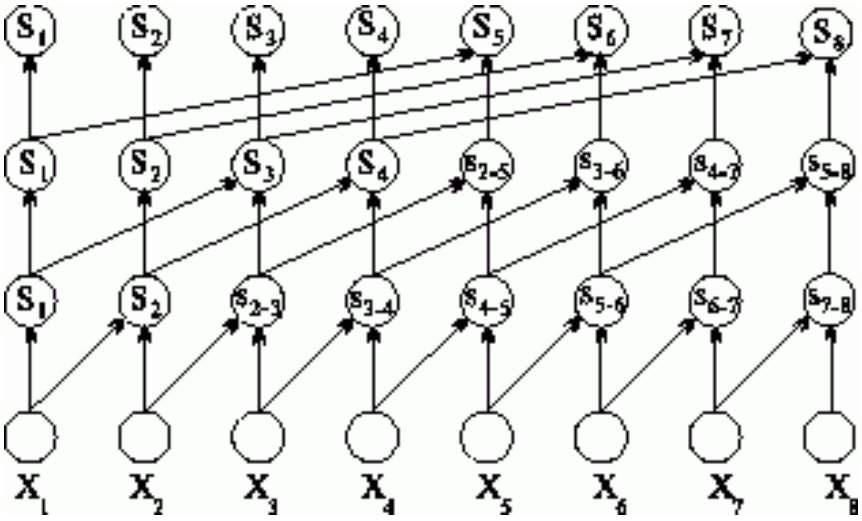
Вернемся к исходной задаче вычисления всех частных сумм последовательности значений и проведем анализ возможных способов последовательной и параллельной организации вычислений. Вычисление всех частных сумм на скалярном компьютере может быть получено при помощи обычного последовательного алгоритма суммирования, при том же количестве операций (!)

$$\mathbf{T_1=n.}$$

При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемым результатам; достижение эффективного распараллеливания требует привлечения новых подходов (может быть, даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения задач. Так, для рассматриваемой задачи нахождения всех частных сумм алгоритм, обеспечивающий получение результатов за $\log_2 n$ параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (рис. 5):

– перед началом вычислений создается копия S вектора суммируемых значений (S=x);

– далее на каждой итерации суммирования i, 1ilog2n, формируется вспомогательный вектор Q путем сдвига вправо вектора S на 2i-1 позиций (освобождающиеся при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов S и Q.



Величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности

Рис. 2.6. Схема параллельного алгоритма вычисления всех частных сумм

Всего параллельный алгоритм выполняется за log2n параллельных операций сложения. На каждой итерации алгоритма параллельно

выполняются n скалярных операций сложения и, таким образом, общее количество скалярных операций определяется величиной

$$\mathbf{K_{пар}=n\log_2 n,}$$

параллельный алгоритм содержит большее (!) количество операций по сравнению с последовательным способом суммирования. Необходимое количество процессоров определяется количеством суммируемых значений (p=n).

С учетом полученных соотношений показатели ускорения и эффективности параллельного алгоритма вычисления всех частных сумм оцениваются следующим образом:

$$\mathbf{S_p=T_1/T_p=n/\log_2 n,}$$

$$\mathbf{E_p=T_1/pT_p=n/(p\log_2 n)=n/(n\log_2 n)=1/\log_2 n.}$$

Как следует из построенных оценок, эффективность алгоритма также уменьшается при увеличении числа суммируемых значений, и при необходимости повышения величины этого показателя может оказаться полезной модификация алгоритма, как и в случае с обычной каскадной схемой.

Рассмотренная схема проектирования и реализации параллельных вычислений дает способ понимания параллельных алгоритмов и программ. На стадии проектирования параллельный метод может быть представлен в виде графа «подзадачи – сообщения», который представляет собой не что иное, как укрупненное (агрегированное) представление графа информационных зависимостей графа «операции – операнды». Аналогично на стадии выполнения для описания параллельной программы может быть использована модель в виде графа «процессы – каналы», в которой вместо подзадач используется понятие процессов, а информационные зависимости заменяются каналами передачи сообщений. Дополнительно на этой модели может быть показано распределение процессов по процессорам вычислительной системы, если количество подзадач превышает число процессоров.

Использование двух моделей параллельных вычислений позволяет лучше разделить проблемы, которые проявляются при разработке параллельных методов. Первая модель – граф «подзадачи – сообщения» – позволяет сосредоточиться на вопросах выделения подзадач одинаковой вычислительной сложности, обеспечивая при этом низкий уровень информационной зависимости между подзадачами. Вторая модель – граф «процессы – каналы» – концентрирует внимание на вопросах распределения подзадач по процессорам, обеспечивая еще одну возможность снижения трудоемкости информационных взаимодействий между подзадачами за счет размещения на одних и тех же процессорах интенсивно взаимодействующих процессов. Кроме того, эта модель позволяет лучше анализировать эффективность разработанного параллельного метода и обеспечивает возможность более адекватного описания процесса выполнения параллельных вычислений.

В модели «процессы – каналы» используются следующие понятия:

– под процессом понимается выполняемая на процессоре программа, которая использует для своей работы часть локальной памяти процессора и содержит ряд операций приема/передачи данных для организации информационного взаимодействия с другими выполняемыми процессами параллельной программы;

– канал передачи данных с логической точки зрения может рассматриваться как очередь сообщений, в которую один или несколько процессов могут отправлять пересылаемые данные и из которой процесс-адресат может извлекать сообщения, отправляемые другими процессами.

В общем случае, можно считать, что каналы возникают динамически в момент выполнения первой операции приема/передачи с каналом. По степени общности канал может соответствовать одной или нескольким командам приема данных процесса-получателя; аналогично, при передаче сообщений канал может использоваться одной или несколькими командами передачи данных одного или нескольких процессов. Для снижения

сложности моделирования и анализа параллельных методов будем предполагать, что емкость каналов является неограниченной и, как результат, операции передачи данных выполняются практически без задержек простым копированием сообщений в канал. С другой стороны, операции приема сообщений могут приводить к задержкам (блокировкам), если запрашиваемые из канала данные еще не были отправлены процессами – источниками сообщений.

Следует отметить важное достоинство рассмотренной модели «процессы – каналы» – в ней проводится четкое разделение локальных (выполняемых на отдельном процессоре) вычислений и действий по организации информационного взаимодействия одновременно выполняемых процессов. Такой подход значительно снижает сложность анализа эффективности параллельных методов и существенно упрощает проблемы разработки параллельных программ.

2.2. Закон Амдала

Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены.

Имеется n процессоров и p(%) вычислений, которые не могут выполняться параллельно. Во сколько раз быстрее можно выполнить вычисления по сравнению с одним процессором?

Например, если n=10, p=50, а на одном процессоре все вычисления выполняются за время t. Тогда первая половина вычислений (50 %) будет выполнена за время t/(2*10) , а вторая – за время t/2. Общее время вычислений в этом случае составит t/2 + t/20=11*t/20, а ускорение по сравнению с одним процессором составит 20/11 раза. Если же n=10, p=25, и на одном процессоре все вычисления выполняются за время t. Тогда 75 % вычислений будут выполнены за время 3*t/(4*10), а оставшиеся 25 % – за время t/4. Общее время вычислений в этом случае составит

t/4 + 3*t/40=13*t/40, а ускорение по сравнению с одним процессором составит 40/13 раза.

Сам же закон Амдала (иногда также Закон Амдаля-Уэра) записывается следующим образом:

$$S<=1/(f+(1-f)/p),$$

где S – ускорение работы программы на p процессорах, а f – доля непараллельного кода в программе. И иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей.

Эта формула справедлива и при программировании в модели общей памяти, и в модели передачи сообщений. Несколько разный смысл вкладывается в понятие доля непараллельного кода. Для SMP систем (модель общей памяти) эту долю образуют те операторы, которые выполняет только главная нить программы. Для MPP систем (механизм передачи сообщений) непараллельная часть кода образуется за счет операторов, выполнение которых дублируется всеми процессорами. Оценить эту величину из анализа текста программы практически невозможно. Такую оценку могут дать только реальные просчеты на различном числе процессоров. Из закона Амдала следует, что P-кратное ускорение может быть достигнуто, только когда доля непараллельного кода равна 0. Очевидно, что добиться этого практически невозможно. Очень наглядно действие закона Амдала можно продемонстрировать, подсчитав ускорение работы программы на разных количествах процессоров.

Можно подсчитать такое ускорение для 2 процессоров, при этом доля последовательных вычислений составляет тоже 2 %.

Итак, дано f=2, p=2.

Подставим в нашу формулу значения f, p. Подсчитав, получим значение ускорения, которое равняется – 1.96. Ускорения для 2, 4, 8, 16, 32, 512, 2048 процессоров представлены в табл. 1.

Таблица 1

Ускорение работы программы для разного числа процессоров

Число процессоров	Доля последовательных вычислений %				
	50	25	10	5	2
	Ускорение работы программы				
2	1,33	1,6	1,82	1,9	1,96
4	1,6	2,28	3,07	3,48	3,77
8	1,78	2,91	4,71	5,93	7,02
16	1,88	3,36	6,4	9,14	12,31
32	1,94	3,66	7,8	12,55	19,75
512	1,99	3,97	9,83	19,28	45,63
2048	2	3,99	9,96	19,82	48,83

Из таблицы хорошо видно, что если, например, доля последовательного кода составляет 2 %, то более чем 50-кратное ускорение в принципе получить невозможно. С другой стороны, нецелесообразно запускать такую программу на 2048 процессорах с тем, чтобы получить 49-кратное ускорение. Тем не менее, такая задача достаточно эффективно будет выполняться на 16 процессорах, а в некоторых случаях потеря 37 % производительности при выполнении задачи на 32 процессорах может быть вполне приемлемой. В некотором смысле, закон Амдала устанавливает предельное число процессоров, на котором программа будет выполняться с приемлемой эффективностью в зависимости от доли непараллельного кода. Заметим, что эта формула не учитывает накладные расходы на обмены между процессорами, поэтому на практике ситуация может быть еще хуже.

Не следует забывать, что распараллеливание программы – это лишь одно из средств ускорения ее работы. Не меньший эффект, а иногда и больший, может дать оптимизация однопроцессорной программы. Чрезвычайную актуальность эта проблема приобрела в последнее время из-за большого разрыва в скорости работы кэш-памяти и основной памяти. К сожалению, зачастую этой проблеме не уделяется должного внимания. Это приводит к тому, что тратятся значительные усилия на распараллеливание заведомо неэффективных программ.

Подведем итоги, что же нам дает закон Амдала с практической точки зрения?

Закон Амдала позволяет подсчитать реальную скорость ускорения вычислений на многопроцессорных системах. То есть определить нужное количество процессоров, которое должно быть, для более эффективных вычислений.

Пусть f есть доля последовательных вычислений в применяемом алгоритме обработки данных, тогда в соответствии с законом Амдала ускорение процесса вычислений при использовании p процессоров ограничивается величиной

$$S_p \le \frac{1}{f + (1-f)/p} \le S^* = \frac{1}{f}.$$

Так, например, при наличии всего 10 % последовательных команд в выполняемых вычислениях эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных.

Закон Амдала характеризует одну из самых серьезных проблем в области параллельного программирования – алгоритмов без определенной доли последовательных команд практически не существует. Однако часто доля последовательных действий характеризует не возможность параллельного решения задач, а последовательные свойства применяемых алгоритмов. Поэтому доля последовательных вычислений может быть

существенно снижена при выборе более подходящих для распараллеливания методов.

Следует отметить также, что рассмотрение закона Амдала происходит в предположении, что доля последовательных расчетов f является постоянной величиной и не зависит от параметра n, определяющего вычислительную сложность решаемой задачи. Однако, для большого ряда задач доля f = f(n) является убывающей функцией от n и, в этом случае, ускорение для фиксированного числа процессоров может быть увеличено за счет увеличения вычислительной сложности решаемой задачи. Данное замечание может быть сформулировано как утверждение, что ускорение Sp=Sp(n) является возрастающей функцией от параметра n (данное утверждение часто именуется эффект Амдала).

2.3. Закон Густафсона

Оптимистичный взгляд на закон Амдала дает закон Густафсона-Барсиса. Вместо вопроса об ускорении на n процессорах можно рассмотреть вопрос о замедлении вычислений при переходе на один процессор.

Можно оценить максимально достижимое ускорение исходя из имеющейся доли последовательных расчетов в выполняемых параллельных вычислениях:

$$g = \frac{\tau(n)}{\tau(n) + \pi(n)/p},$$

где **τ(n)** и **π (n)** есть времена последовательной и параллельной частей

выполняемых вычислений соответственно, т. е.

$$T_1 = \tau(n) + \pi(n),\ T_p = \tau(n) + \pi(n)/p.$$

С учетом введенной величины g можно получить

$$\tau(n)=g\cdot(\tau(n)+\pi(n)/p),\ \pi(n)=(1-g)p\cdot(\tau(n)+\pi(n)/p),$$

что позволяет построить оценку для ускорения

$$S_p=\frac{T_1}{T_p}=\frac{\tau(n)+\pi(n)}{\tau(n)+\pi(n)/p}=\frac{(\tau(n)+\pi(n)/p)(g+(1-g)p)}{\tau(n)+\pi(n)/p},$$

которая после упрощения приводится к виду закона Густавсона-Барсиса:

$$\mathbf{Sp = g+(1-g)p = p+(1-p)g.}$$

Аналогично за f принимается доля последовательной части программы. Тогда получим закон масштабируемого ускорения

$$\mathbf{S(n)=n+(1–n)*f}$$

и линейное ускорение в зависимости от числа процессоров.

При рассмотрении закона Густавсона-Барсиса следует учитывать еще один важный момент. С увеличением числа используемых процессоров темп уменьшения времени параллельного решения задач может падать (после превышения определенного порога). Однако за счет уменьшения времени вычислений сложность решаемых задач может быть увеличена. Оценку получаемого при этом ускорения можно определить при помощи сформулированных закономерностей. Такая аналитическая оценка тем более полезна, поскольку решение таких более сложных вариантов задач на одном процессоре может оказаться достаточно трудоемким и даже невозможным, например, в силу нехватки оперативной памяти. С учетом указанных обстоятельств оценку ускорения, получаемую в соответствии с законом Густавсона-Барсиса, еще называют ускорением масштабирования, поскольку данная характеристика может показать, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач.

Целью применения параллельных вычислений во многих случаях является не только уменьшение времени выполнения расчетов, но и обеспечение возможности решения более сложных вариантов задач (таких

постановок, решение которых не представляется возможным при использовании однопроцессорных вычислительных систем). Способность параллельного алгоритма эффективно использовать процессоры при повышении сложности вычислений является важной характеристикой выполняемых расчетов. Поэтому параллельный алгоритм называют масштабируемым, если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении постоянного уровня эффективности использования процессоров. Возможный способ характеристики свойств масштабируемости состоит в следующем.

Оценим накладные расходы, которые имеют место при выполнении параллельного алгоритма:

$$T_0 = pT_P - T_1.$$

Накладные расходы появляются за счет необходимости организации взаимодействия процессоров, выполнения некоторых дополнительных действий, синхронизации параллельных вычислений и т. п. Используя введенное обозначение, можно получить новые выражения для времени параллельного решения задачи и соответствующего ускорения:

$$Tp = (T1 + T0) / p,\ Sp = T1 / Tp = pT1 / (T1 + T0).$$

Применяя полученные соотношения, эффективность использования процессоров можно выразить как:

$$E_P = S_P / p = T_1 / (T_1 + T_0) = 1 / (1 + T_0 / T_1).$$

Последнее выражение показывает, что если сложность решаемой задачи является фиксированной (T1=const), то при росте числа процессоров эффективность, как правило, будет убывать за счет роста накладных расходов T0. При фиксации числа процессоров эффективность их использования можно улучшить путем повышения сложности решаемой задачи T1 (предполагается, что при росте параметра сложности n накладные расходы T0 увеличиваются медленнее, чем объем вычислений T1). Как результат, при увеличении числа процессоров в большинстве случаев можно

обеспечить определенный уровень эффективности при помощи соответствующего повышения сложности решаемых задач. Поэтому важной характеристикой параллельных вычислений становится соотношение необходимых темпов роста сложности расчетов и числа используемых процессоров.

Пусть E=const есть желаемый уровень эффективности выполняемых вычислений. Из выражения для эффективности можно получить:

$$T_0 / T_1 = (1 - E) / E \text{ или } T_1 = KT_0, \text{ где } K = E / (1 - E).$$

Порождаемую последним соотношением зависимость n=F(p) между сложностью решаемой задачи и числом процессоров обычно называют функцией изоэффективности.

Таким образом, законы Амдала и Густафсона в идентичных условиях дают различные значения ускорения. Где же ошибка?

Каковы области применения этих законов?

Густафсон заметил, что, работая на многопроцессорных системах, пользователи склонны к изменению своего поведения. Теперь снижение общего времени исполнения программы уступает объему решаемой задачи. Такое изменение цели обусловливает переход от закона Амдала к закону Густафсона. Например, на 100 процессорах программа выполняется 20 минут. При переходе на систему с 1000 процессорами можно достичь времени исполнения порядка 2 минут. Однако для получения большей точности решения имеет смысл увеличить на порядок объем решаемой задачи (например, решить систему уравнений в частных производных на более тонкой сетке). То есть при сохранении общего времени исполнения пользователи стремятся получить более точный результат.

Увеличение объема решаемой задачи приводит к увеличению доли параллельной части, так как последовательная часть (ввод/вывод, менеджмент потоков, точки синхронизации и т. п.) не изменяется. Таким образом, уменьшение доли f приводит к перспективным значениям ускорения.

3. Разработка параллельного алгоритма

Использование современных высокопроизводительных вычислительных средств параллельного действия для обработки большого количества быстро поступающей информации определило распараллеливание алгоритмов как одно из основных направлений на современном этапе развития науки и техники. С одной стороны, важной задачей является автоматическое распараллеливание алгоритмов обработки информации. От решения этой задачи во многом зависит успех внедрения многопроцессорных систем в практику вычислений. С другой стороны, при разработке высокопроизводительных вычислительных средств параллельного действия требуется априори определить основные свойства внутреннего параллелизма алгоритмов и закономерности, которым подчиняется структура параллельных процессов обработки информации. Определение классов алгоритмов, допускающих распараллеливание на заданном уровне и реализуемых за заданное время, является главной проблемой, от решения которой зависит успешная и целенаправленная разработка высокопроизводительных вычислительных средств для решения задач в реальном времени. Это позволит оптимально приблизить структуру задачи к структуре разрабатываемой вычислительной системы. Такое соответствие может быть полностью достигнуто лишь тогда, когда известны основные законы, определяющие внутреннюю структуру параллельных алгоритмов.

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему.

Действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

– выполнить анализ имеющихся вычислительных схем и осуществить их разделение (декомпозицию) на части (подзадачи), которые могут быть реализованы в значительной степени независимо друг от друга;

– выделить для сформированного набора подзадач информационные взаимодействия, которые должны осуществляться в ходе решения исходной поставленной задачи;

– определить необходимую (или доступную) для решения задачи вычислительную систему и выполнить распределение имеющего набора подзадач между процессорами системы.

При самом общем рассмотрении понятно, что объем вычислений для каждого используемого процессора должен быть примерно одинаков – это позволит обеспечить равномерную вычислительную загрузку (балансировку) процессоров. Кроме того, также понятно, что распределение подзадач между процессорами должно быть выполнено таким образом, чтобы количество информационных связей (коммуникационных взаимодействий) между подзадачами было минимальным.

После выполнения всех перечисленных этапов проектирования можно оценить эффективность разрабатываемых параллельных методов: для этого обычно определяются значения показателей качества порождаемых параллельных вычислений (ускорение, эффективность, масштабируемость). По результатам проведенного анализа может оказаться необходимым повторение отдельных (в предельном случае всех) этапов разработки – следует отметить, что возврат к предшествующим шагам разработки может происходить на любой стадии проектирования параллельных вычислительных схем.

Поэтому часто выполняемым дополнительным действием в приведенной выше схеме проектирования является корректировка состава сформированного множества задач после определения имеющегося количества процессоров – подзадачи могу быть укрупнены (агрегированы) при наличии малого числа процессоров или, наоборот, детализированы в противном случае. В целом, данные действия могут быть определены как масштабирование разрабатываемого алгоритма и выделены в качестве отдельного этапа проектирования параллельных вычислений.

Чтобы применить получаемый в конечном итоге параллельный алгоритм, необходимо выполнить разработку программ для решения сформированного набора подзадач и разместить разработанные программы по процессорам в соответствии с выбранной схемой распределения подзадач. Для проведения вычислений программы запускаются на выполнение (программы на стадии выполнения обычно именуются процессами), для реализации информационных взаимодействий программы должны иметь в своем распоряжении средства обмена данными (каналы передачи сообщений).

Следует отметить, что каждый процессор обычно выделяется для решения единственной подзадачи, однако при наличии большого количества подзадач или использовании ограниченного числа процессоров это правило может не соблюдаться и, в результате, на процессорах может выполняться одновременно несколько программ (процессов). В частности, при разработке и начальной проверке параллельной программы для выполнения всех процессов может использоваться один процессор (при расположении на одном процессоре процессы выполняются в режиме разделения времени).

Рассмотрев внимательно разработанную схему проектирования и реализации параллельных вычислений, можно отметить, что данный подход в значительной степени ориентирован на вычислительные системы с распределенной памятью, когда необходимые информационные взаимодействия реализуются при помощи передачи сообщений по каналам связи между процессорами. Тем не менее, данная схема может быть применена без потери эффективности параллельных вычислений и для разработки параллельных алгоритмов для систем с общей памятью – в этом случае механизмы передачи сообщений для обеспечения информационных взаимодействий должны быть заменены операциями доступа к общим (разделяемым) переменным.

3.1. Этапы разработки параллельных алгоритмов

Исходя из вышесказанного, процесс разработки параллельного алгоритма можно разбить на четыре этапа (рис. 3.1):

– декомпозиция, или разделение вычислений на независимые части;

– проектирование коммутаций, или выделение информационных зависимостей;

– масштабирование набора подзадач;

– планирование вычислений, или распределение подзадач между процессорами.

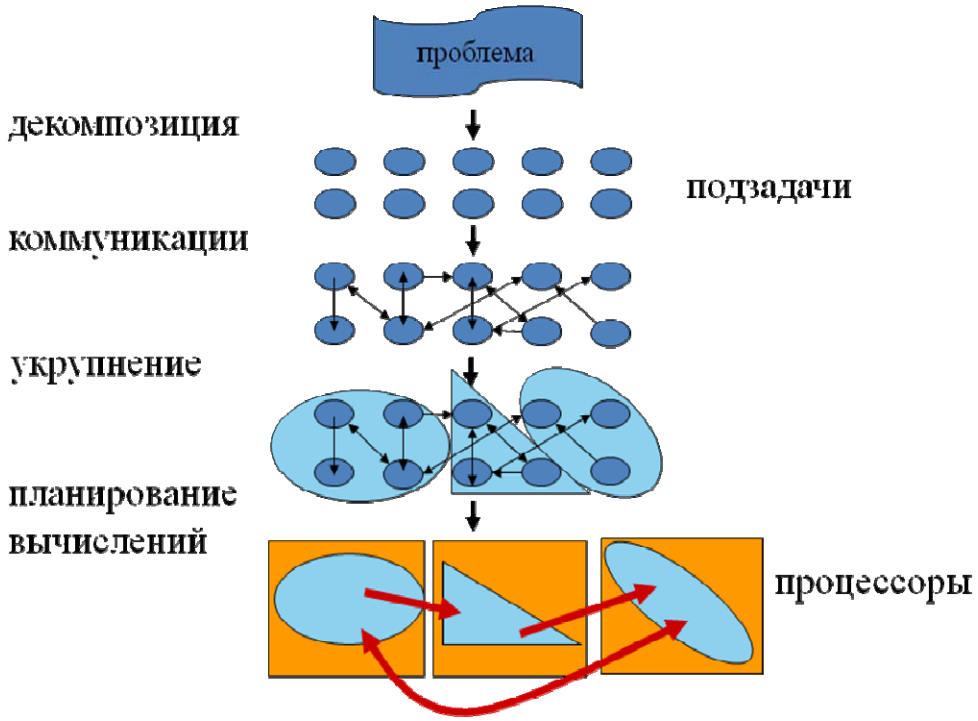


Рис. 3.1. Этапы разработки параллельных алгоритмов

3.1.1. Разделение вычислений на независимые части

Выбор способа разделения вычислений на независимые части основывается на анализе вычислительной схемы решения исходной задачи.

Сначала определяется степень возможного распараллеливания задачи. Чем меньше размер подзадач, получаемых в результате декомпозиции, чем больше их количество, тем более гибким оказывается параллельный алгоритм, тем легче обеспечить равномерную нагрузку процессоров.

Сегментироваться могут как вычислительные алгоритмы, так и данные. В этой связи существуют два метода декомпозиции. Метод декомпозиции данных и функциональной декомпозиции.

В методе декомпозиции данных, данные разбиваются на фрагменты приблизительно одинакового размера. С фрагментами данных связываются операции их обработки, из которых и формируются подзадачи и определяются необходимые пересылки данных. Пересечение частей, на которые разбивается программа, должны быть минимальны. Это позволяет избежать дублирования вычислений. Схема разбиения в дальнейшем может уточняться. В частности, если это необходимо для уменьшения числа обменов, допускается увеличение степени перекрытия фрагментов задачи. Например, в инженерных и научных расчетах часто используются различные сеточные методы. В этом случае трехмерная сетка определяет набор данных, элементы которого соответствуют узлам решетки.

Требования, которым должен удовлетворять выбираемый подход, обычно состоят в обеспечении равного объема вычислений в выделяемых подзадачах и минимума информационных зависимостей между этими подзадачами (при прочих равных условиях нужно отдавать предпочтение редким операциям передачи сообщений большего размера по сравнению с частыми пересылками данных небольшого объема).

В общем случае, проведение анализа и выделение задач представляет собой достаточно сложную проблему – ситуацию помогает разрешить существование двух часто встречающихся типов вычислительных схем (рис. 3.2).

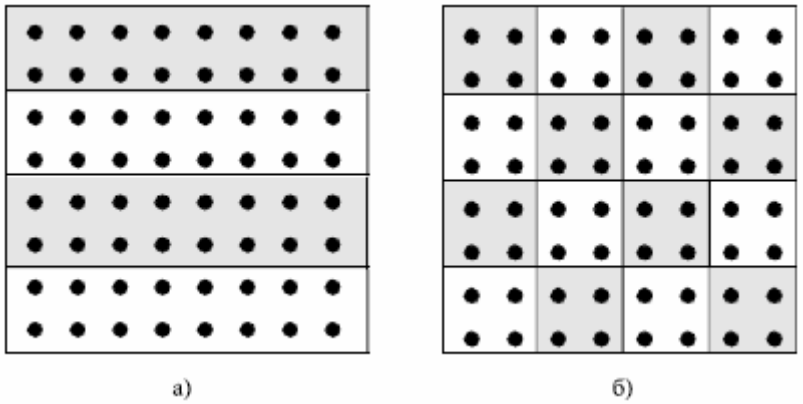


Рис. 3.2. Разделение данных матрицы: а) ленточная схема, б) блочная схема

При параллелизме по данным вычисления сводятся к выполнению однотипной обработки большого набора данных. К такому классу задач относятся, например, матричные вычисления, численные методы решения уравнений в частных производных и др. Так, например, для решения задачи поиска максимального элемента в матрице, при формировании подзадач, исходная матрица может быть разделена на отдельные строки (или последовательные группы строк) – так называемая ленточная схема разделения данных (рис. 3.2), либо на прямоугольные наборы элементов – блочная схема разделения данных. Для большого количества решаемых задач разделение вычислений по данным приводит к порождению одно-, двух- и трехмерных наборов подзадач, в которых информационные связи существуют только между ближайшими соседями (такие схемы обычно именуются сетками или решетками). Таким образом, декомпозиция может быть одномерной (крупноблочное разбиение), двумерной и трехмерной (рис. 3.3).

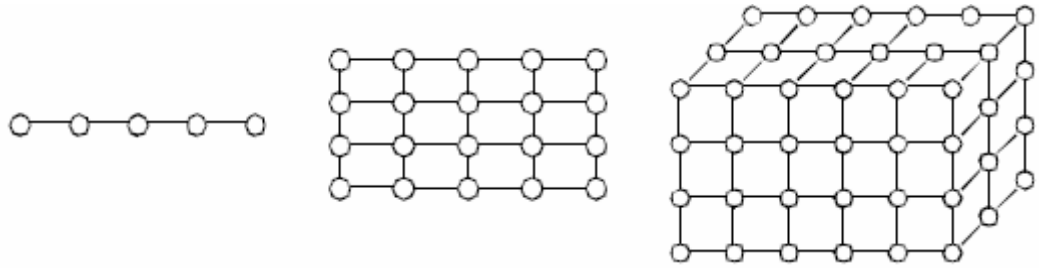


Рис. 3.3. Регулярные одно-, двух- и трехмерные структуры базовых подзадач после декомпозиции данных

Для другой части задач вычисления могут состоять в выполнении разных операций над одним и тем же набором данных – в этом случае говорят о существовании функционального параллелизма (в качестве примеров можно привести задачи обработки последовательности запросов к информационным базам данных, вычисления с одновременным применением разных алгоритмов расчета и т. п.). Очень часто функциональная декомпозиция может быть использована для организации конвейерной обработки данных (так, например, при выполнении каких-либо преобразований данных вычисления могут быть сведены к функциональной последовательности ввода, обработки и сохранения данных).

В методе функциональной декомпозиции сначала сегментируется вычислительный алгоритм, а затем уже под эту схему подгоняется декомпозиция данных. Однако при этом схема может оказаться такой, что потребует многочисленные дополнительные пересылки данных. Однако данный метод полезен, если нет структур данных очевидных для распараллеливания.

Для того чтобы декомпозиция была эффективной, необходимо придерживаться следующих рекомендаций:

– количество подзадач должно примерно на порядок превосходить число процессоров (для обеспечения гибкости на следующих этапах разработки);

– следует избегать лишних вычислений и пересылок данных, так как в противном случае возникает сложность масштабирования и снижается эффективность при решении задач большого объема;

– подзадачи должны быть примерно одинакового размера, для сбалансированности нагрузки;

– желательно, чтобы с увеличением объема задачи количество подзадач не возрастало.

Важный вопрос при выделении подзадач состоит в выборе нужного уровня декомпозиции вычислений. Формирование максимально возможного

количества подзадач обеспечивает использование предельно достижимого уровня параллелизма решаемой задачи, однако затрудняет анализ параллельных вычислений. Применение при декомпозиции вычислений только достаточно «крупных» подзадач приводит к ясной схеме параллельных вычислений, однако может затруднить эффективное использование достаточно большого количества процессоров. Возможное разумное сочетание этих двух подходов может состоять в применении в качестве конструктивных элементов декомпозиции только тех подзадач, для которых методы параллельных вычислений являются известными. Так, например, при анализе задачи матричного умножения в качестве подзадач можно использовать методы скалярного произведения векторов или алгоритмы матрично-векторного произведения. Подобный промежуточный способ декомпозиции вычислений позволит обеспечить и простоту представления вычислительных схем, и эффективность параллельных расчетов. Выбираемые подзадачи при таком подходе именуются далее базовыми, которые могут быть элементарными (неделимыми), если не допускают дальнейшего разделения, или составными – в противном случае.

Размер блоков, из которых состоит параллельная программа, может быть разным. В зависимости от размера блоков алгоритм может иметь различную «зернистость». Ее мерой в простейшем случае может служить количество операций блоке. Выделяют три степени «зернистости»: мелкоблочный, среднеблочный и крупноблочный параллелизм.

Параллелизм на уровне команд – мелкоблочный его масштаб менее 20 команд на блок. Количество параллельно выполняемых подзадач – от 1 до нескольких тысяч, причем средний масштаб параллелизма составляет около 5 команд на блок.

Следующий уровень – это параллелизм на уровне циклов. Обычно цикл содержит не более 500 команд. Если итерации цикла независимы, они могут выполняться, например, с помощью конвейера или на векторном процессоре. Это тоже мелкозернистый параллелизм.

Параллелизм на уровне процедур – среднеблочный. Размер блока до 2000 команд. При этом следует учитывать возможные межпроцедурные зависимости.

Параллелизм на уровне задач – крупноблочный. Он соответствует выполнению независимых программ на параллельном компьютере. Требуется поддержка операционной системы.

Добиться эффективной работы параллельной программы можно, сбалансировав «зернистость» алгоритма и затраты времени на обмен данными. Различные подсистемы параллельного компьютера характеризуются различным временем задержки, которые определяет степень масштабируемости системы.

Части программы могут выполняться параллельно, только если они независимы.

Независимость можно рассматривать с четырех сторон:

– независимость по данным – данные, обрабатываемые одной частью программы, не модифицируются другой;

– независимость по управлению – порядок выполнения частей программы может быть определен только во время выполнения программы, в противном случае предполагается последовательное выполнение (рис. 3.4.);

For i:=1 to n do
Begin
A[i]:=b[i];
If a[i]>c then a[i]:=1;
End
 	
Зависима по управлению
	For i:=1 to n do
	If a[i]>c then a[i]:=1;
	
	Независима от управления

Рис. 3.4. Определение зависимости по управлению

– независимость по ресурсам – наличие достаточного количества памяти, функциональных узлов и т. д.;

– независимость по выводу – не производится запись в одну переменную или файл.

Декомпозиция должна быть такой, чтобы время обработки данных превосходило время пересылки. Можно считать, что сегментирование – это искусство.

Для рассматриваемой учебной задачи достаточный уровень декомпозиции может состоять, например, в разделении матрицы на множество отдельных строк и получении на этой основе набора подзадач поиска максимальных значений в отдельных строках; порождаемая при этом структура информационных связей соответствует линейному графу (рис. 3.5).

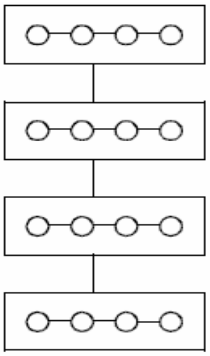


Рис. 3.5. Структура информационной задачи нахождения максимального значения в матрице

Для оценки корректности этапа разделения вычислений на независимые части можно воспользоваться следующим списком вопросов:

– Выполненная декомпозиция не увеличивает объем вычислений и необходимый объем памяти?

– Возможна ли при выбранном способе декомпозиции равномерная загрузка всех имеющихся процессоров?

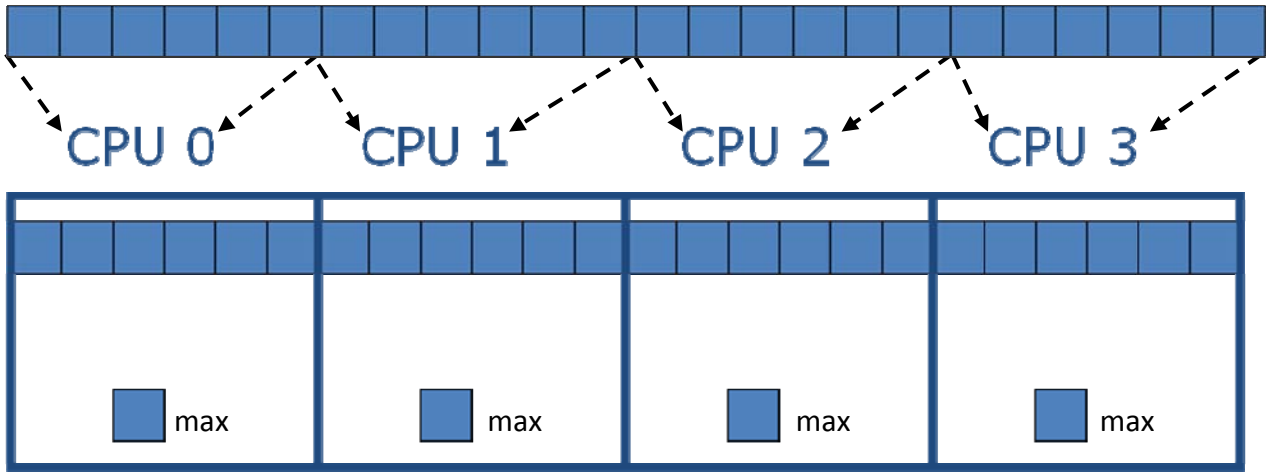
– Достаточно ли выделенных частей процесса вычислений для эффективной загрузки имеющихся процессоров (с учетом возможности увеличения их количества)?

При увеличении количества процессорных ядер декомпозиция по данным позволяет увеличить количество решаемых проблем. То есть

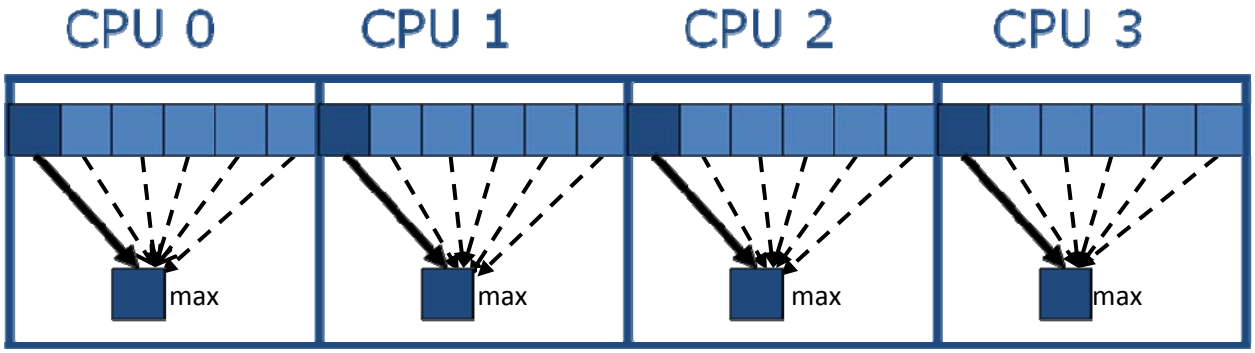
благодаря декомпозиции можно выполнить больше работы за одно и то же время. Для иллюстрации можно рассмотреть пример поиска максимального (минимального) элемента в массиве.

Пример: Декомпозиция по данным. Поиск максимального элемента массива при наличии четырех процессоров.

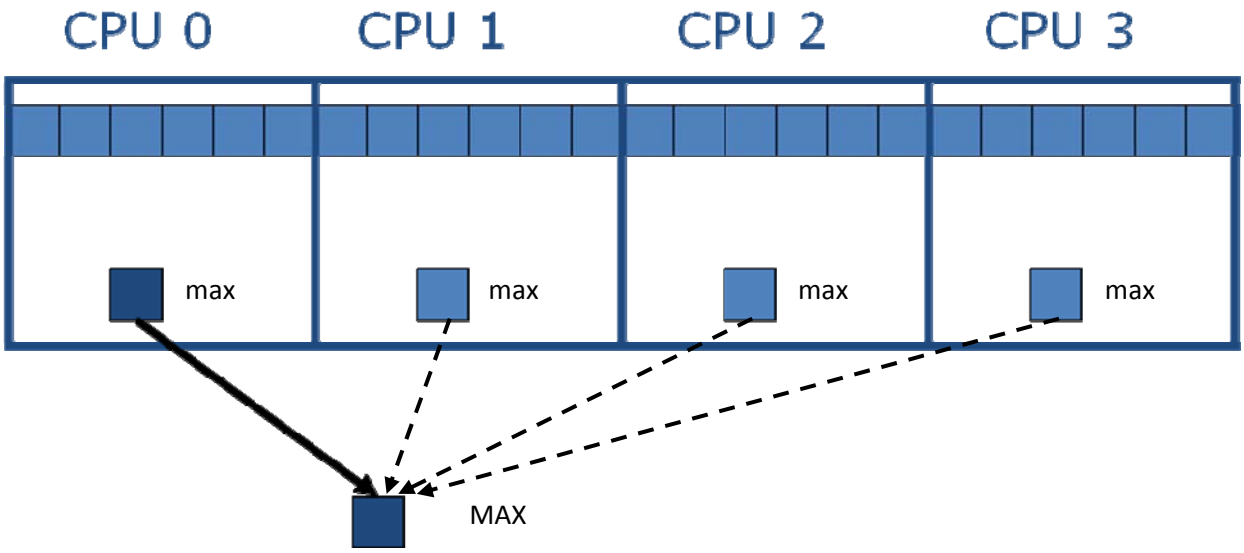
Весь массив разбивается на четыре части и распределяется по процессорам (рис. 3.6 а). На каждом из процессоров выполняется поиск максимального элемента в соответствующей части массива с использованием последовательного алгоритма поиска (рис. 3.6 б). После этого формируется промежуточный массив из четырех элементов, содержащий максимальные элементы каждой из четырех частей массива. Поиск максимального элемента этого массива выполняется на одном из процессоров, при этом также используется последовательный алгоритм (рис. 3.6 в).



а) Распределение элементов массива по процессорам



б) Поиск максимального элемента в каждой части массива



в) Поиск максимального элемента в массиве из четырех элементов

Рис. 3.6. Поиск максимального элемента в массиве на четырех процессорах

3.1.2. Выделение информационных зависимостей

При наличии вычислительной схемы решения задачи после выделения базовых подзадач определение информационных зависимостей между ними обычно не вызывает больших затруднений. При этом, однако, следует отметить, что на самом деле этапы выделения подзадач и информационных зависимостей достаточно сложно поддаются разделению. Выделение подзадач должно происходить с учетом возникающих информационных связей, после анализа объема и частоты необходимых информационных обменов между подзадачами может потребоваться повторение этапа разделения вычислений.

При проведении анализа информационных зависимостей между подзадачами следует различать:

– локальные и глобальные схемы передачи данных – для локальных схем в каждый момент времени передачи данных выполняются только между небольшим числом подзадач (располагаемых, как правило, на соседних процессорах), для глобальных – в процессе коммуникации принимают участие все подзадачи;

– структурные и произвольные способы взаимодействия – для структурных способов организация взаимодействий приводит к формированию некоторых стандартных схем коммуникации (например, в виде кольца, прямоугольной решетки и т. д.), для произвольных – схема выполняемых операций передач данных не является однородной;

– статические или динамические схемы передачи данных – для статических схем моменты и участники информационного взаимодействия фиксируются на этапах проектирования и разработки параллельных программ, для динамического варианта – структура операции передачи данных определяется в ходе выполняемых вычислений;

– синхронные и асинхронные способы взаимодействия – для синхронных способов операции передачи данных выполняются только при готовности всех участников взаимодействия и завершаются только после

полного окончания всех коммуникационных действий, при асинхронном выполнении участники взаимодействия могут не дожидаться полного завершения действий по передаче данных. Для представленных способов взаимодействия достаточно сложно выделить предпочтительные формы организации передачи данных: синхронный вариант, как правило, более прост для применения, в то время как асинхронный способ часто позволяет существенно снизить временные задержки, вызванные операциями информационного взаимодействия.

При проектировании информационных зависимостей следует учитывать, что количество коммутаций у подзадач должно быть примерно одинаковым, т. к. иначе приложение становится слабо масштабируемым. Там где возможно следует использовать локальную коммутацию Коммутации должны быть по возможности параллельными.

При проектировании информационных зависимостей необходимо отслеживать вероятность тупиковых ситуаций, связанные с неправильной последовательностью обмена данными.

Пусть например имеются три подзадачи ПЗ1, ПЗ2, ПЗ3, которые вырабатывают сообщения М1, М2 и М3 соответственно. Задача П1 является «потребителем» сообщения М3, задача П2 является «потребителем» сообщения М1, задача П3 является «потребителем» сообщения М2. То есть имеет место следующая схема

П1: ….

Ждать сообщения (П3, М3)

Послать сообщение (П2, М2)

…

П2: ….

Ждать сообщения (П1, М1)

Послать сообщение (П3, М3)

…

П1: ….

Ждать сообщения (П2, М2)

Послать сообщение (П1, М1)

…

Решить проблему тупиков можно правильным планированием и использованием не блокирующего приема данных, сочетающим прием данных с их отправкой.

В случае блокирующего приема/передачи передающий процесс блокируется на время приема сообщения. Не блокирующий прием/передача завершается немедленно, т. е. для системы не важно прибыло сообщение или нет. Убедиться в этом можно только с помощью функций проверки получения. Обычно вызов таких функций помещается в цикле, который повторяется до тех пор пока тест не вернет значение «истина».

Обмен сообщениями может быть реализован по-разному: с помощью потоков, межпроцессорных коммуникаций (IPC), TCP-сокетов и т. д. Одним из самых распространенных способов программирования коммутаций – использование библиотек, реализующих обмен сообщениями. Например, PVM, MPI, которые позволяют создавать параллельные программы для различных платформ. Так, программы, написанные для кластера, могут выполняться на суперкомпьютере.

Метод RPC, позволяющий одному процессу вызывать процедуры другого процесса и получать результаты ее выполнения, CORBA, DCOM и т. д.

Для оценки правильности этапа выделения информационных зависимостей можно воспользоваться контрольным списком вопросов, предложенным:

– Соответствует ли вычислительная сложность подзадач интенсивности их информационных взаимодействий?

– Является ли одинаковой интенсивность информационных взаимодействий для разных подзадач?

– Является ли схема информационного взаимодействия локальной?

– Не препятствует ли выявленная информационная зависимость параллельному решению подзадач?

3.1.3. Масштабирование набора подзадач

Масштабирование разработанной вычислительной схемы параллельных вычислений проводится в случае, если количество имеющихся подзадач отличается от числа планируемых к использованию процессоров. Для сокращения количества подзадач необходимо выполнить укрупнение (агрегацию) вычислений. Применяемые здесь правила совпадают с рекомендациями начального этапа выделения подзадач: определяемые подзадачи, как и ранее, должны иметь одинаковую вычислительную сложность, а объем и интенсивность информационных взаимодействий между подзадачами должны оставаться на минимально возможном уровне. Как результат, первыми претендентами на объединение являются подзадачи с высокой степенью информационной взаимозависимости.

При недостаточном количестве имеющихся подзадач для загрузки всех доступных к использованию процессоров необходимо выполнить детализацию (декомпозицию) вычислений. Как правило, проведение подобной декомпозиции не вызывает каких-либо затруднений, если для базовых задач методы параллельных вычислений являются известными.

Выполнение этапа масштабирования вычислений должно свестись, в конечном итоге, к разработке правил агрегации и декомпозиции подзадач, которые должны параметрически зависеть от числа процессоров, применяемых для вычислений.

Для рассматриваемой задачи поиска максимального значения агрегация вычислений может состоять в объединении отдельных строк в группы (ленточная схема разделения матрицы – см. рис. 4.3 а), при декомпозиции подзадач строки исходной матрицы могут разбиваться на несколько частей (блоков).

Список контрольных вопросов для оценки правильности этапа масштабирования, выглядит следующим образом:

– Не ухудшится ли локальность вычислений после масштабирования имеющегося набора подзадач?

– Имеют ли подзадачи после масштабирования одинаковую вычислительную и коммуникационную сложность?

– Соответствует ли количество задач числу имеющихся процессоров?

– Зависят ли параметрически правила масштабирования от количества процессоров?

3.1.4. Распределение подзадач между процессорами

Распределение подзадач между процессорами является завершающим этапом разработки параллельного метода. Надо отметить, что управление распределением нагрузки для процессоров возможно только для вычислительных систем с распределенной памятью, для мультипроцессоров (систем с общей памятью) распределение нагрузки обычно выполняется операционной системой автоматически. Кроме того, данный этап распределения подзадач между процессорами является избыточным, если количество подзадач совпадает с числом имеющихся процессоров, а топология сети передачи данных вычислительной системы представляет собой полный граф (т. е. все процессоры связаны между собой прямыми линиями связи).

Основной показатель успешности выполнения данного этапа – эффективность использования процессоров, определяемая как относительная доля времени, в течение которого процессоры использовались для вычислений, связанных с решением исходной задачи. Пути достижения хороших результатов в этом направлении остаются прежними: как и ранее, необходимо обеспечить равномерное распределение вычислительной нагрузки между процессорами и минимизировать количество сообщений, передаваемых между ними. Точно так же как и на предшествующих этапах

проектирования, оптимальное решение проблемы распределения подзадач между процессорами основывается на анализе информационной связности графа «подзадачи – сообщения». Так, в частности, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных.

Следует отметить, что требование минимизации информационных обменов между процессорами может противоречить условию равномерной загрузки. Можно разместить все подзадачи на одном процессоре и полностью устранить межпроцессорную передачу сообщений, однако понятно, что загрузка большинства процессоров в этом случае будет минимальной.

Для задачи поиска максимального значения распределение подзадач между процессорами не вызывает каких-либо затруднений – достаточно лишь обеспечить размещение подзадач, между которыми имеются информационные связи, на процессорах, для которых существуют прямые каналы передачи данных. Поскольку структура информационной связей данной задачи имеет вид линейного графа, выполнение этого требования может быть обеспечено практически при любой топологии сети вычислительной системы.

Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений может изменяться в ходе решения задачи. Причиной этого могут быть, например, неоднородные сетки при решении уравнений в частных производных, разреженность матриц и т. п. Кроме того, используемые на этапах проектирования оценки вычислительной сложности решения подзадач, могут иметь приближенный характер, и, наконец, количество подзадач может изменяться в ходе вычислений. В таких ситуациях может потребоваться перераспределение базовых подзадач между процессорами уже непосредственно в ходе выполнения параллельной программы (или, как обычно говорят, придется выполнить динамическую балансировку вычислительной нагрузки). Данные

вопросы являются одними из наиболее сложных (и наиболее интересных) в области параллельных вычислений – к сожалению, их рассмотрение выходит за рамки нашего учебного материала.

При планировании вычислений обычно используют алгоритмы двух групп:

– алгоритмы планирования выполнения задач task scheduling (основаны на функциональной декомпозиции, при которой может создаваться множество коротко живущих подзадач);

– эвристические алгоритмы сбалансированной загрузки процессоров.

Планирование выполнения задач заключается в организации доступа к ресурсам. Порядок предоставления доступа определяется используемым для этого алгоритмом. В качестве примера можно привести планирование по принципу кругового обслуживания (round robin), когда несколько процессов обслуживаются по очереди, получая одинаковые порции процессорного времени. Часто используют метод сбалансированной нагрузки (load balancing) процессоров вычислительной системы, который основан на учете текущей загрузки каждого процессора, иногда может предусматриваться перенос процесса с одного процессора на другой.

Согласно принципу планирования выполнения задач формируется пул задач, в который включаются вновь созданные задачи и из которого задачи выбираются для выполнения на освободившемся процессоре. Стратегия размещения задач на процессоре представляет собой компромисс между требованиями максимальной независимости выполнения задач и глобальным учетом состояния вычислений. Чаще всего применяется стратегии master/slave, иерархические и децентрализованные.

В качестве примера можно дать краткую характеристику широко используемого способа динамического управления распределением вычислительной нагрузки, обычно именуемого схемой «менеджер – исполнитель» (the manager-worker scheme). При использовании данного подхода предполагается, что подзадачи могут возникать и завершаться в ходе

вычислений, при этом информационные взаимодействия между подзадачами либо полностью отсутствуют, либо минимальны. В соответствии с рассматриваемой схемой для управления распределением нагрузки в системе выделяется отдельный процессор-менеджер, которому доступна информация обо всех имеющихся подзадачах. Остальные процессоры системы являются исполнителями, которые для получения вычислительной нагрузки обращаются к процессору-менеджеру. Порождаемые в ходе вычислений новые подзадачи передаются обратно процессору-менеджеру и могут быть получены для решения при последующих обращениях процессоров-исполнителей. Завершение вычислений происходит в момент, когда процессоры-исполнители завершили решение всех переданных им подзадач, а процессор-менеджер не имеет каких-либо вычислительных работ для выполнения.

Перечень контрольных вопросов для проверки этапа распределения подзадач состоит в следующем:

– Не приводит ли распределение нескольких задач на один процессор к росту дополнительных вычислительных затрат?

– Существует ли необходимость динамической балансировки вычислений?

– Не является ли процессор-менеджер «узким» местом при использовании схемы «менеджер – исполнитель»?

3.2. Распараллеливание арифметических вычислений и параллельная сортировка

Проблема распараллеливания арифметических выражений – одна из наиболее изученных в параллельном программировании.

С точки зрения структуры, вычисление выражения – это ациклический процесс, моделируемый ориентированным ациклическим графом или иными словами, деревом.

Основная цель распараллеливания выражения – разработать дерево вычислений минимальной высоты (т. е. выражение должно вычисляться за минимально возможное время или число шагов алгоритма при условии реализации на каждом ярусе максимального числа независимых операций). Распараллеливание выражения осуществляется в рамках модели MIMD с неограниченным параллелизмом.

Задача распараллеливания выражений – построить алгоритм, который по каждому выражению E дает эквивалентное ему E' с минимальной высотой дерева вычислений.

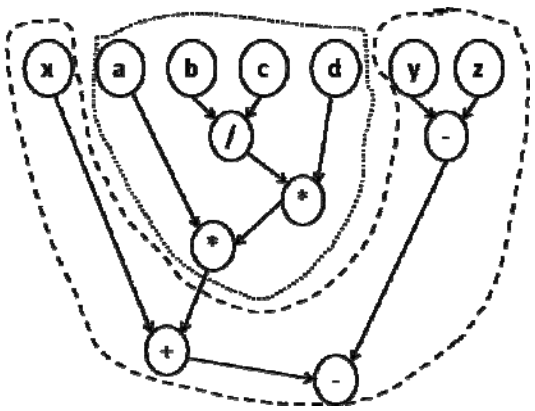
Два выражения E и E' называются эквивалентными, если выражение E преобразуется в E' и наоборот путем использования конечного числа раз законов ассоциативности, коммутативности и дистрибутивности.

Дано выражение E = (x + (a · ((b / c) · d))) – (y – z) (рис. 3.7 а).

Дерево вычислений одного из возможных эквивалентных для E выражений E'= ((a · b) / (c / d)) – ((y – z) – x) (рис. 3.7 б).

Таким образом, выражение E' имеет лучшие характеристики параллельности, чем выражение E. Как можно заметить по приведенному примеру, для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений, и построены, соответственно, разные вычислительные модели. Разные схемы вычислений обладают различными возможностями для распараллеливания, и тем самым, при построении модели вычислений может быть поставлена задача выбора наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.

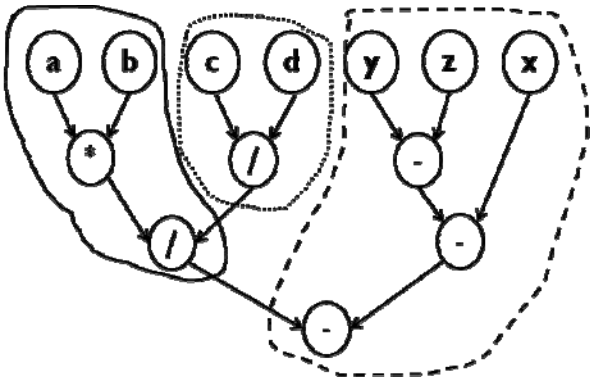
E=(x+(a*((b/c)*d)))-(y-z)



Количество используемых процессоров: 2
количество тактов: 5
число операций: 6

а)

E'=((a*b)/(c/d))-((y-z)-x)



Количество используемых процессоров: 3
количество тактов: 3
число операций: 6

б)

Рис. 3.7. Дерево вычислений

При распараллеливании циклов одной из важных проблем является выявление зависимости по данным. Прежде всего, сюда включается выявление зависимостей по данным между операторами программы. Такая зависимость возникает, например, в случае, если один оператор использует значение переменной, вычисленное некоторым другим оператором. Независимые операторы могут быть выполнены параллельно на разных процессорах. Обычно этот принцип применяется к циклам: если нет зависимостей по данным между разными витками цикла, то витки могут быть выполнены параллельно разными процессорами. Решение этой проблемы является важнейшим этапом анализа. Один из методов анализа основан на исследовании исходного текста программ и построении информационного графа.

Можно рассмотреть два простых примера.

На рис. 3.8 приведен пример цикла и информационный граф, где отсутствует зависимость по данным. В этом случае можно выполнять параллельно все итерации цикла.

```
for (i = 0; i < 3; i++)
a[i] = b[i] / 2.0;
```

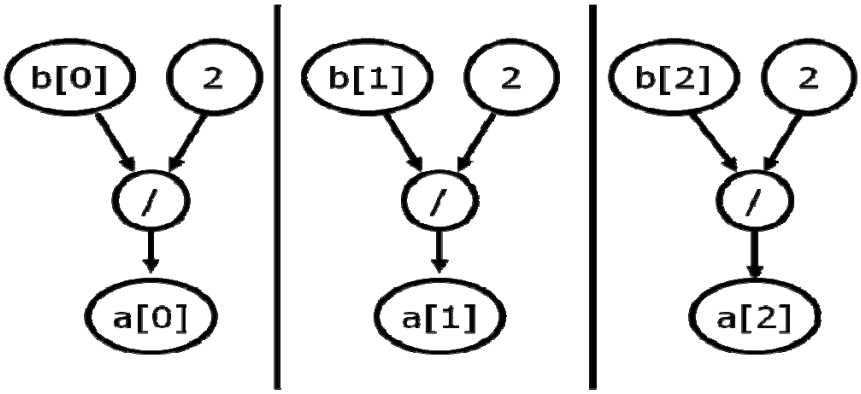


Рис. 3.8. Цикл с отсутствием зависимости по данным

На рис. 3.9 значение элемента массива зависит от значения предыдущего элемента, что видно на информационном графе. Следовательно, в этом случае возможно только последовательное выполнение итераций

```
for (i = 1; i < 4; i++)
a[i] = a[i-1] * b[i];
```

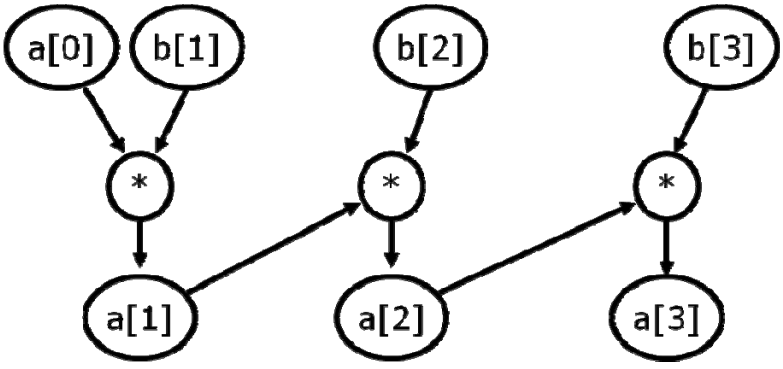


Рис. 3.9. Цикл с зависимостью по данным

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений в порядке возрастания или убывания.

Существует множество различных алгоритмов сортировки, но самый простой из них это сортировка пузырьком. К сожалению, он также один из самых медленных. Избавиться от этого недостатка помогает алгоритм параллельной пузырьковой сортировки.

Последовательный алгоритм пузырьковой сортировки сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности (a_1, a_2, ..., a_n) алгоритм сначала выполняет n-1 базовых операций «сравнения-обмена» для последовательных пар элементов (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n). В результате после первой итерации алгоритма самый большой элемент перемещается («всплывает») в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности (a'_1, a'_2, ..., a'_{n-1}).

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод чет-нечетной перестановки (the odd-even transposition method). Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода: в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

(a1, a2), (a3, a4), ..., (an-1,an) (при четном n) (рис. 3.10 а),

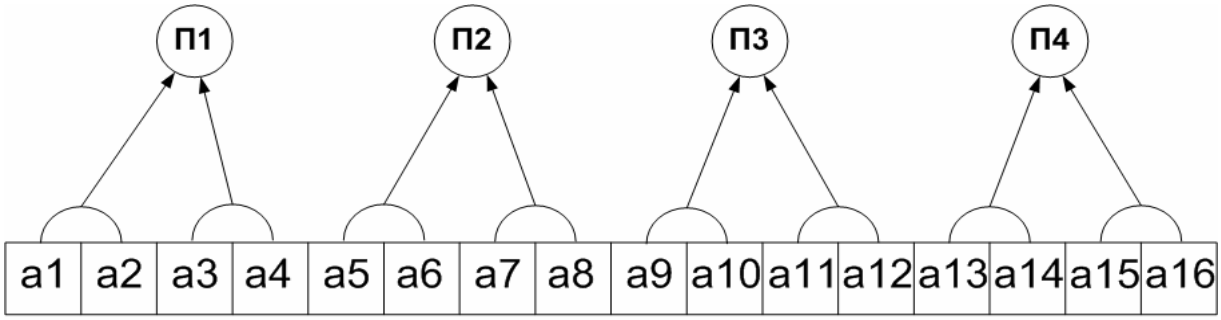
а на четных итерациях обрабатываются элементы

(a2, a3), (a4, a5), ..., (an-2,an-1) (рис. 3.10 б).

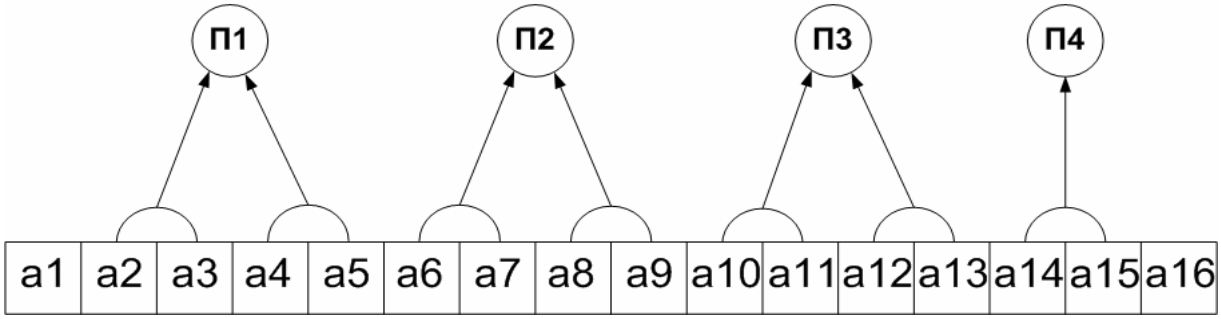
После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным. В общем случае выполнение параллельного метода может быть прекращено, если в течение каких-либо двух последовательных итераций сортировки состояние упорядочиваемого набора данных не было изменено.

```
// Последовательный алгоритм чет-нечетной перестановки
void OddEvenSort(double A[], int n) {
  for (int i = 1; i < n; i++) {
    if (i % 2 == 1) {    // нечетная итерация
      for (int j = 0; j < n/2 - 2; j++)
        compare_exchange(A[2*j + 1], A[2*j + 2]);
      if (n % 2 == 1) // сравнение последней пары при нечетном n
        compare_exchange(A[n - 2], A[n - 1]);
    }
    else    // четная итерация
      for (int j = 1; j < n/2 - 1; j++)
        compare_exchange(A[2*j], A[2*j + 1]);
  }
}
```

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. В случае p<n, когда количество процессоров меньше числа упорядочиваемых значений, процессоры содержат блоки данных размера n/p и в качестве базовой подзадачи может быть использована операция «сравнить и разделить».



а) Нечетная итерация



б) Четная итерация

Рис. 3.10. Алгоритм чет-нечетной сортировки

```
// Параллельный алгоритм чет-нечетной перестановки
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();     // номер процесса
  int np = GetProcNum();    // количество процессов
  for (int i = 0; i < np; i++ ) {
    if (i % 2 == 1) {    // нечетная итерация
      if (id % 2 == 1) {    // нечетный номер процесса
        // Сравнение-обмен с процессом — соседом справа
        if (id < np – 1) compare_split_min(id + 1);
      }
      else
        // Сравнение-обмен с процессом — соседом слева
        if (id > 0) compare_split_max(id – 1);
```

```
    }
    else {    // четная итерация
      if(id % 2 == 0) {    // четный номер процесса
        // Сравнение-обмен с процессом — соседом справа
        if (id < np – 1) compare_split_min(id + 1);
      }
      else
        // Сравнение-обмен с процессом — соседом слева
        compare_split_max(id – 1);
    }
  }
}
```

4. Издержки и выигрыш при реализации параллельных и векторных вычислений

4.1. Способы векторизации и распараллеливания программ

Целью программиста не должно быть получение правильного результата вычислений любой ценой, но получение правильного результата наибыстрейшим, оптимальным способом. Если программа предназначена для однократного использования, то лучше написать ее как можно проще, не оптимизируя ее быстродействие и используемую память, чтобы потратить минимум усилий на тестирование и отладку. Если программа предназначена для частого использования или время ее работы будет гораздо больше времени ее написания и отладки, то не следует жалеть труда на оптимизацию ее быстродействия. Но в результате программа, оптимальная для одного типа ЭВМ, окажется совсем не оптимальной для машин других типов.

Методы программирования для скалярных, векторных и параллельных ЭВМ коренным образом отличаются друг от друга. Конечно, любая программа, написанная на языке высокого уровня, может быть оттранслирована в коды любой ЭВМ и исполнена как на скалярной, так и на векторной или параллельной машинах и результаты работы всех программ скорее всего будут близкими. Эту оговорку надо принять во внимание. Дело в том, что результат действий с вещественными числами может изменяться при изменении последовательности операций, т. к. в компьютерной арифметике закон сочетаний не всегда выполняется:

(A+B)+C не всегда равно A+(B+C).

Самый простой вариант попробовать ускорить имеющуюся программу – это воспользоваться встроенными в транслятор (обычно с ФОРТРАНа или Си) средствами векторизации или распараллеливания. При этом никаких изменений в программу вносить не придется. Однако вероятность существенного ускорения (в разы или десятки раз) невелика. Трансляторы с ФОРТРАНа и Си векторизуют и распараллеливают

программы очень аккуратно и при любых сомнениях в независимости обрабатываемых данных оптимизация не проводится. Поэтому, кстати, и не приходится ожидать ошибок от компиляторов, если программист явно не указывает компилятору выполнить векторную или параллельную оптимизацию какой-либо части программы.

Второй этап работы с такой программой – анализ затрачиваемого времени разными частями программы и определение наиболее ресурсопотребляющих частей. Последующие усилия должны быть направлены именно на оптимизацию этих частей. В программах наиболее затратными являются циклы и усилия компилятора направлены прежде всего на векторизацию и распараллеливание циклов. Диагностика компилятора поможет установить причины, мешающие векторизовать и распараллелить циклы. Возможно, что простыми действиями удастся устранить эти причины. Это может быть простое исправление стиля программы, перестановка местами операторов (цикла и условных), разделение одного цикла на несколько, удаление из критических частей программы лишних операторов (типа операторов отладочной печати). Небольшие усилия могут дать здесь весьма существенный выигрыш в быстродействии.

Третий этап – замена алгоритма вычислений в наиболее критичных частях программы. Способы написания оптимальных (с точки зрения быстродействия) программ существенно отличаются в двух парадигмах программирования – в последовательной и в параллельной (векторной). Поэтому программа, оптимальная для скалярного процессора, с большой вероятностью не может быть векторизована или распараллелена. В то же время специальным образом написанная программа для векторных или параллельных ЭВМ будет исполняться на скалярных машинах довольно медленно. Замена алгоритма в наиболее критических частях программы может привести к серьезному ускорению программы при относительно небольших потраченных усилиях. Дополнительные возможности предоставляют специальные векторные и параллельные библиотеки

подпрограмм. Используя библиотечные функции, которые оптимизированы для конкретной ЭВМ, можно упростить себе задачу по написанию и отладке программы. Единственный недостаток данного подхода состоит в том, что программа может стать не переносимой на другие машины (даже того же класса), если на них не окажется аналогичной библиотеки.

Написание программы «с нуля» одинаково сложно (или одинаково просто) для машин любых типов. Этот способ является идеальным для разработки эффективных, высокопроизводительных векторных или параллельных программ. Начинать надо с изучения специфики программирования для векторных и параллельных ЭВМ, изучения алгоритмов, которые наиболее эффективно реализуются на ЭВМ данных типов. После этого надо проанализировать поставленную задачу и определить возможность применения векторизуемых и распараллеливаемых алгоритмов для решения конкретной задачи. Возможно, что придется переформулировать какие-то части задачи, чтобы они решались с применением векторных или параллельных алгоритмов. Программа, специально написанная для векторных или параллельных ЭВМ, даст наибольшее ускорение при ее векторизации и распараллеливании.

Программист должен быть готов к поддержанию одновременно нескольких (двух или даже трех) версий программы, предназначенных для работы на ЭВМ разного типа. Это, наверно, самая существенная из всех «издержек», связанная с одновременным использованием скалярных, векторных и параллельных машин.

4.2. Применение разных языков программирования

Перенос готовой программы и оптимизация ее для исполнения на машине другого типа может потребовать весьма существенных усилий по исправлению стиля написания программы. Опыт показывает, что проще всего адаптировать к исполнению на векторных или параллельных машинах программы на ФОРТРАНе. Конструкции ФОРТРАНа для организации

циклов и для работы с массивами менее разнообразны, чем в Си, а именно циклы по обработке массивов и являются главными объектами оптимизации при векторизации или распараллеливанию программ. Более того, в ФОРТРАНе-90 встроенные функции по работе с массивами и арифметические операции с массивами имеют довольно эффективную реализацию в виде стандартных библиотечных векторных или параллельных функций или даже в виде генерируемого компилятором объектного кода (in line).

Больше всего трудностей появится при переносе программ на Си, оптимальным образом написанных для скалярных процессоров. Следующие программы на Си показывают два принципиально разных подхода к операциям с массивами. Традиционный Си'шный подход к копированию массивов является недопустимым при написании векторных или параллельных программ. Приведенный ниже цикл не может быть векторизован или распараллелен из-за того, что значение ссылок на элементы массивов вычисляются рекуррентно (зависят от значений на предыдущем шаге):

```
float x[100], a[100];
register float *xx = x, *aa = a;
register int i;
for( i=0; i<100; i++){ *xx++ = *aa++ }.
```

В то время как другой цикл, написанный совсем не в традициях Си, может быть распараллелен и векторизован:

```
float x[100], a[100];
register int i;
for( i=0; i<100; i++){ x[i] = a[i]; }.
```

Сравните: ФОРТРАН-77 позволяет единственным способом записать цикл и этот способ совпадает со вторым, нетрадиционным способом в Си:

```
real x(100), a(100)
do i=1, 100
```

```
    x(i) = a(i)
  enddo.
```

ФОРТРАН-90 разрешает применить векторную операцию присваивания, что упрощает анализ программы транслятором:

```
  real x(100), a(100)
  x = a ! векторное присваивание.
```

Приведенный пример показывает, что существенным фактором при адаптации готовых программ к работе на параллельных и векторных ЭВМ является не только стиль написания программы, но и сам использованный язык программирования.

4.2.1. Сходство алгоритмов – параллелизм данных

Из архитектуры векторных и параллельных компьютеров следует самое важное свойство алгоритмов, которые могут быть эффективно исполнены на таких ЭВМ: эти алгоритмы должны включать одинаковые действия над некоторыми наборами данных (массивами), при этом результаты действий над любой частью данных не должны зависеть от результатов действий над другой их частью. Иначе говоря, последовательность вычисления величин не должна играть роль в данном алгоритме. Это есть параллелизм данных. Простейшим примером такого алгоритма может служить следующий фрагмент программы:

```
  real x(100), a(100), b(100)
  do i=1, 100
    x(i) = real(i)**2 + 3.0
    a(i) = b(i) + x(i)
  enddo
```

или в другой интерпретации:

```
  real x(100), a(100), b(100)
  do i=100, 1, -1
    x(i) = real(i)**2 + 3.0
```

```
    a(i) = b(i) + x(i)
  enddo.
```

Предположим, что i=5, тогда тело цикла примет вид:

```
x(5) = 5.0**2 + 3.0
a(5) = b(5) + x(5).
```

Порядок исполнения этих двух операторов не может быть изменен, но 5-ые элементы массивов a и x могут быть вычислены независимо от 4-го, 6-го и других элементов. Приведенная программа удовлетворяет основному требованию к векторизуемым алгоритмам – для любого значения i вычисления можно проводить независимо.

Предположим, что имеется векторная ЭВМ с векторными регистрами длиной 100 элементов. Тогда весь цикл по всем элементам массива записывается как линейная (не циклическая) последовательность команд, а каждая команда оперирует со всеми 100 элементами массива. Выигрыш в увеличении быстродействия программ (по отношению к быстродействию не векторизованных программ на той же ЭВМ) может ожидаться равным числу элементов в массиве.

Теперь представим, что у нас есть 100-процессорная параллельная ЭВМ и каждый процессор имеет прямой доступ ко всем элементам массивов. Тогда для каждого скалярного процессора можно написать программу для вычисления значений a(i) и x(i) для одного конкретного i, совпадающего с номером процессора (процессор должен знать свой номер). В принципе такая программа также, как и векторная, будет линейной. Так как у нас в распоряжении 100 процессоров, то после исполнения каждым процессором своих команд все 100 элементов массивов a и x будут вычислены. Очевидно, что это будет в 100 раз быстрее, чем вычисление всех элементов одним процессором.

4.2.2. Различие алгоритмов – параллелизм действий

Можно проследить аналогию в векторной и параллельной реализациях предыдущей программы. Каждая машинная команда вызывает действия сразу над 100 числами: в векторной программе явно выполняются операции над всеми элементами регистра, в параллельной программе каждый из 100 процессоров выполняет более или менее синхронно одинаковые машинные команды, оперирует со своими собственными регистрами и в результате выполняются действия одновременно со 100 числами.

Справедливо следующее утверждение: алгоритм, который можно векторизовать, можно и распараллелить. Обратное утверждение не всегда верно.

В многопроцессорной ЭВМ каждый процессор исполняет свой поток команд. В общем случае для каждого из процессоров параллельной ЭВМ можно составить свою программу, не повторяющую программы для других процессоров. Например, скалярная величина y есть сумма двух функций:

$$y = F(x) + G(x).$$

Тогда один процессор может считать значение F(x), а второй – G(x). После счета достаточно сложить полученные два числа, чтобы получить требуемое значение y. Такой параллелизм действий может быть достигнут (и так поступают наиболее часто) в единой программе для обоих процессоров:

```
if( номер_процессора .eq. 1 ) then
  y1 = F(x)
else if( номер_процессора .eq. 2 ) then
  y2 = G(x)
endif
ждать завершения работы обоих процессоров
if( номер_процессора .eq. 1 ) then
  y = y1 + y2
endif.
```

Параллельность действий (т. е. применение параллельных ЭВМ) дает большое преимущество в программировании перед чистой параллельностью данных (применением векторных ЭВМ). Даже простой вызов подпрограммы приводит к невозможности векторизации цикла, в то время, как распараллеливание возможно:

```
real a(100)
do i=1, 100
   call proc( a(i) )
enddo.
```

4.2.3. Предельное быстродействие векторных программ

Векторный процессор выполняет математические операции сразу над всеми элементами векторного регистра. Если число элементов регистра равно 128, то операция над всеми 128 числами выполняется в векторном режиме так же быстро, как над одним числом в скалярном режиме. Это и есть теоретический предел повышения быстродействия программ при их векторизации. Однако необходимо учесть, что любая векторная операция требует больше машинных тактов для своего исполнения, чем такая же скалярная операция. С другой стороны циклическое N-кратное исполнение скалярной команды для обработки массива требует исполнения еще нескольких команд, организующих собственно цикл. В результате исполнение векторной команды может оказаться эффективнее более, чем в 128 раз. Для простоты можно считать, что предельное повышение эффективности векторных программ равно числу элементов в векторном регистре ЭВМ. Чаще всего это 128 или 256.

4.2.4 Две части программы – скалярная и векторная

В любой программе существуют две части – векторизуемая и не векторизуемая. Например, алгоритмы построения последовательностей, заданных рекуррентным отношением, нельзя векторизовать – каждый последующий элемент зависит от предыдущих и, соответственно, не может

быть вычислен ни ранее, ни одновременно с предыдущими. Другие невекторизуемые части программ – ввод/вывод, вызов подпрограмм или функций, организация циклов, разветвленные алгоритмы, работа со скалярными величинами. Это довольно широкий класс подзадач, он гораздо шире класса векторизуемых алгоритмов. При векторизации программ на самом деле ускоряется выполнение только части программы (большей или меньшей). Поэтому каждую программу можно представить такой упрощенной диаграммой, показанной на рис. 4.1 (если собрать все векторизуемые и все скалярные части в единые блоки):

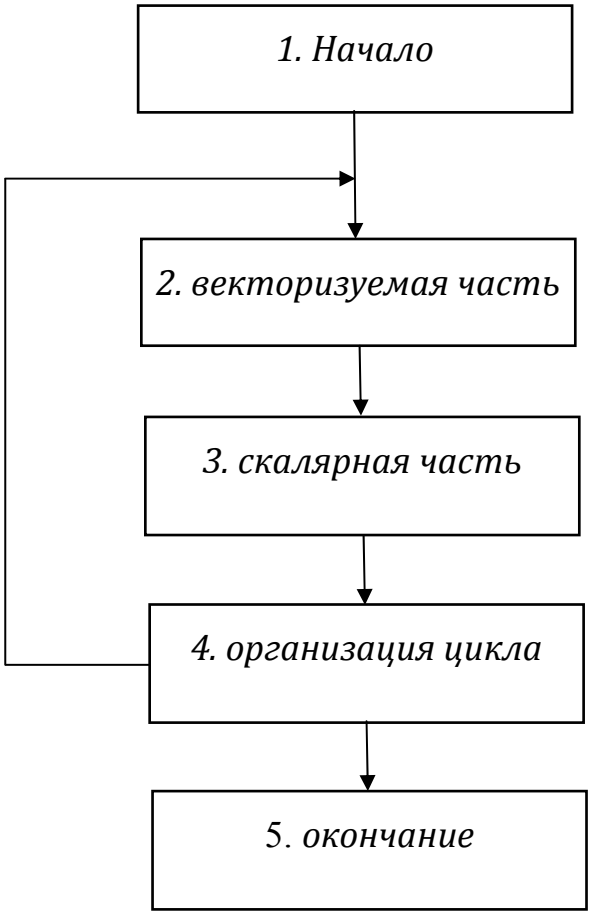


Рис. 4.1. Общий случай схемы программы

Для начала исполним программу в полностью скалярном варианте (т. е. умышленно не будем векторизовать исполняемый код). Обозначим время исполнения каждой из частей программы через T1...T5. Тогда полное время работы невекторизуемых частей программы будет равно

$$\mathbf{T_{скал} = T1 + T3 + T4 + T5,}$$

а полное время работы программы будет

T' = Tскал + T2.

Далее перетранслируем программу в режиме векторизации. Единственная часть программы, которая ускорит свое выполнение – это вторая часть. Предположим, что достигнутое ускорения работы этой части составляет N раз. Тогда полное время работы всей программы составит

T" = Tскал + T2/N.

А выигрыш в эффективности работы всей программы будет

P = T' / T" = (Tскал + T2)/ (Tскал + T2/N),

что совсем не равно N. Положим, что Tскал = 1с, T2 = 99с, а N=100 (очень хороший показатель для 128-элементных векторных процессоров). В нашем варианте эффективность P будет всего (1+99)/(1+99/100)=50 или 40 % от предельно возможных 128 раз, а при Tскал = 2с значение P будет (2+98)/(2+98/100)=34 (27 %). Хотя само по себе увеличение быстродействия программы в 50 или даже в 30 раз при ее векторизации является очень большим (вычисления будут занимать 1 сутки вместо 1 месяца), но предельно возможное ускорение в 128 (или 256) раз не может быть достигнуто на векторных ЭВМ. Практика показывает, что хорошим показателем увеличения быстродействия P можно считать уже значения 6–10, что соответствует времени исполнения скалярной части всего 10–15 % от полного времени исполнения программы (T2 составляет 85–90 %).

4.3. Параллельные ЭВМ и параллельные программы

При работе на параллельных ЭВМ пользователь имеет возможность запускать программу или на всех процессорах сразу, или на ограниченном их числе. Поскольку все процессоры в параллельных ЭВМ одинаковые (в составе параллельной ЭВМ могут работать еще и специализированные процессоры ввода/вывода, но на них счет не проводится), то можно ожидать, что программа будет выполняться во столько раз быстрее, сколько процессоров будут проводить вычисления. Но это только в идеальном случае, на самом деле это далеко не так.

4.3.1. Три части программы – параллельная, последовательная и обмен данными

Как и в векторных программах, в любом параллельном алгоритме присутствуют параллельная и последовательная части. В отличие от векторизации, внутренние циклы, ветвящиеся алгоритмы, вызовы подпрограмм и функций не являются препятствием для распараллеливания программы. При распараллеливании программы могут быть оптимизированы внешние, самые всеобъемлющие циклы. Однако любые рекуррентные вычисления, ввод/вывод, вычисления, понижающие размерность массивов (вплоть до скаляра), не могут быть (полностью) распараллелены.

Исполнение разных частей программы разными процессорами или, если быть точнее, разными процессами вносит дополнительный обязательный фрагмент в программу, а именно, обмен данными между процессами. Современные параллельные ЭВМ исполняют разные копии одинаковой программы в качестве отдельных задач, т. е. процессов. Каждый процесс может иметь свои локальные данные и глобальные данные, к которым есть доступ у всех процессов. Результаты, сосчитанные в одних процессах, в определенные моменты должны передаваться в другой или другие процессы для дальнейшей работы. Это процесс обмена данными. Фрагмент программы, который будет исполняться на 4-х процессорной ЭВМ:

```
      real x(4)
1     do i=1,4
        x(i) = func(i)
      enddo
2     s = 0.0
3     do i=1,4
        s = s + x(i)
      enddo.
```

Все 4 элемента массива x можно вычислить параллельно в цикле с меткой 1. При этом вообще цикл не понадобится, т. к. у нас число процессов

будет равно числу искомых элементов массива. Переменную s должен инициализовать только один процесс – это последовательный фрагмент программы. Цикл с меткой 3 должен также выполняться одним процессом. Для этого надо сначала передать этому процессу все значения x(i), i=1..4, из других процессов. Этот цикл не сможет начаться раньше, чем будет вычислен и передан последний (не по номеру, а по времени) элемент массива x. Таким образом, главный процесс (проводящий суммирование) будет ожидать завершение передачи элементов массива всеми остальными процессами.

Время обмена данными зависит от архитектуры параллельной ЭВМ. Оно может быть равно нулю для многопроцессорных рабочих станций с общей оперативной памятью и организацией распараллеливания в пределах одного процесса или составлять значительную величину при обмене в кластерах ЭВМ, связанных компьютерной сетью.

Выводы: объем данных, предназначенных для обмена, должен быть по возможности меньше, а последовательная часть программы должна быть как можно быстрее. Часто это удается совмещать путем проведения частичных вычислений в параллельном режиме (например, вычисление частичных сумм) с последующей передачей промежуточных результатов в главный процесс (например, для вычисления полной суммы).

4.3.2. Синхронизация процессов, равномерность загрузки процессов

Еще один важный фактор, влияющий на ускорение работы параллельных программ, есть равномерность загрузки процессов. При обсуждении предыдущей программы было сказано, что главный процесс перед началом исполнения цикла 3 должен получить все элементы массива x. Даже если собственно время обмена данными будет равно нулю, то все равно цикл не сможет начаться до окончания вычисления последнего (не по номеру, а по времени) из элементов x. За этим следит одна из важнейших

частей параллельного алгоритма, которая часто называется «барьером» и осуществляет синхронизацию процессов.

Если предположить, что время вычисления x(i) будет равно 1, 2, 3 и 4 секундам для соответствующих i, тогда самое последнее значение x(4) будет получено через 4 секунды после начала вычислений, а цикл 3 не сможет начаться ранее этого времени.

Если к концу предыдущей программы дописать такой распараллеливаемый фрагмент:

```
4    do i=1,4
        x(i) = x(i)/s
     enddo
```

то, несмотря на незагруженность трех процессов исполнением цикла 3, они не смогут продолжить работу до его (цикла) окончания и рассылки главным процессом значения s во все процессы. Перед циклом 4 неявно запрограммирована синхронизация всех процессов, которая может привести к их простою. Предположим, что главным процессом у нас является третий. Тогда первый процесс после завершения вычисления x(1) (на это у него уйдет 1 секунда) перейдет в режим ожидания значения s: 3 с для завершения вычисления x(4) плюс время обмена элементами массива плюс время вычисления суммы третьим процессом и, наконец, плюс время получения s.

Важный вывод из сказанного выше – программист должен распределить вычислительную работу как можно более равномерно между всеми процессами.

5. Распараллеливающие компиляторы

Средства автоматизированного распараллеливания следует использовать на первом этапе разработки параллельной программы, если у нее уже существует работающий последовательный аналог. В этом случае на первом этапе разработки параллельной программы рекомендуется применить средства автоматизированного распараллеливания к исходной последовательной программе.

Так называемые «высокопроизводительные ФОРТРАН и Си» (high-performance FORTRAN and C - HPF and HPC) являются новыми стандартами на компиляторы для параллельных суперкомпьютеров. Эти языки полностью совместимы с «обычными» ФОРТРАН-77 и Си/Си++. Обычная программа может быть без каких-либо изменений оттранслирована для супер-ЭВМ и исполнена на любом числе процессоров. Однако такой простейший подход приведет к тому, что каждый из процессов на супер-ЭВМ будет полностью от начала и до конца исполнять всю программу без какого-либо реального распараллеливания.

Распараллеливающий компилятор сам находит параллельность в последовательной программе и создает корректно синхронизированную параллельную программу, которая содержит последовательный код и библиотечные вызовы.

Автоматическое распараллеливание позволяет:

- удешевить и ускорить разработку параллельных программ;
- снизить требования к квалификации программистов;
- повысить надежность разрабатываемых программ, поскольку объем исходного текста последовательной программы на порядок меньше, чем параллельной, а ее логическая структура проще;
- упростить перенос многих разработанных программ с последовательных компьютеров на параллельные;
- эффективнее проектировать программно-аппаратные комплексы с учетом новых методов распараллеливания.

Основная задача распараллеливающего компилятора – извлечь как можно больше скрытого параллеллизма из участков повторяемости последовательной программы, определяющих основное время ее выполнения: циклов и рекурсивных процедур. Это требует изменения порядка исполнения операторов в участках повторяемости последовательной программы, которые до выполнения алгоритмов распараллеливания, в случае отсутствия рекурсивных процедур, можно привести к виду гнезд DO-циклов.

Алгоритмы распараллеливания циклов – это весьма важный класс реструктурирующих преобразований. Они особенно привлекательны тем, что их применение не требует знания целевой архитектуры. Эти алгоритмы можно рассматривать как завершающую стадию машинно-независимой части процесса генерации параллельного кода распараллеливающим компилятором. Распараллеливание циклов позволяет обнаружить параллелизм и выявить те зависимости, которые ответственны за изначально последовательное» состояние некоторых операций исходной программы.

For i_1=1, r_1 Do
For i_2=1, r_2 Do
...
For i_n=1, r_n Do $T(i_1, i_2,\ldots,i_n)$

} циклический фрагмент программы (T – тело цикла)

Множество целочисленных векторов:

$I = \{(i_1, i_2, \ldots, i_n) \mid 1 \le i_k \le r_k, k \in [1, n]\}$ – пространство итераций гнезда циклов.

Задача распараллеливания гнезда циклов ставится как задача разбиения множества I на подмножества $I_1, I_2, \ldots, I_S$ такие, что вычисления тела цикла $T(i_1, i_2, \ldots, i_n)$ в каждом из них могут быть выполнены одновременно (естественно, с сохранением информационных связей исходного цикла).

Распараллеливание циклических фрагментов программ выполняется следующим образом (рис. 5.1):

– на этапе синтаксического анализа проверяется выполнение некоторых ограничений на тела циклов (линейность индексных выражений,

отсутствие операторов и обращений к подпрограммам и т. п. – см. далее), кроме того, оценивается время выполнения программы;

– на этапе распараллеливания циклов проверяется выполнение всех необходимых ограничений на тела циклов (в частности, условия Рассела – см. далее), здесь же для каждого цикла производится выбор метода распараллеливания;

– на этапе оценки качества распараллеливания производится анализ ускорения параллельной программы по сравнению с исходной последовательной программой;

– на этапе генерации параллельной программы генерируется программа на параллельном ЯВУ;

– на этапе генерации машинного кода компилятор генерирует код для используемой параллельной вычислительной системы.

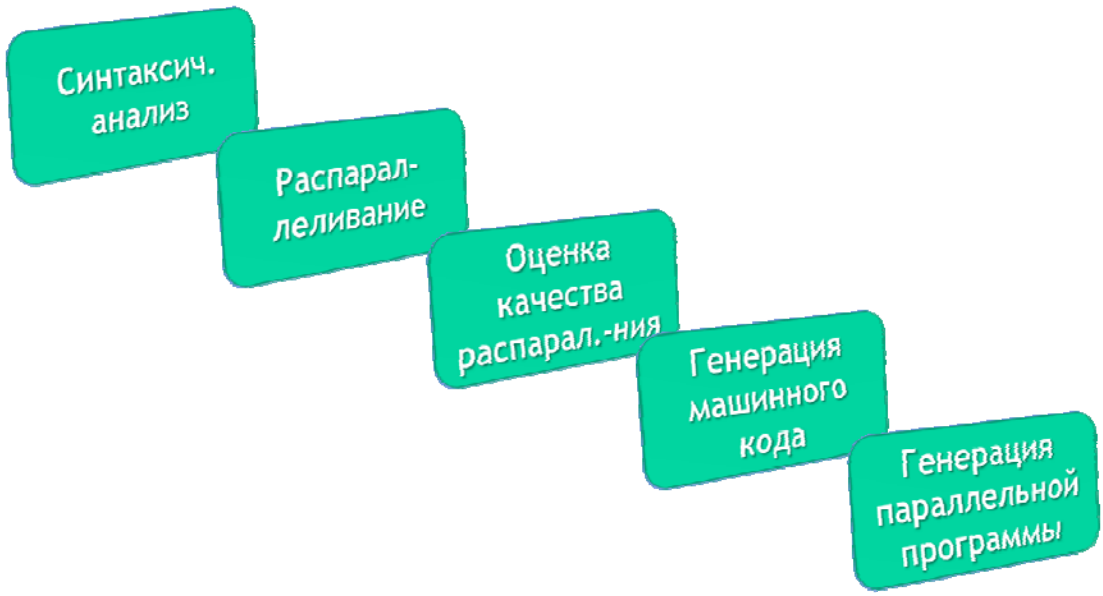


Рис. 5.1. Общий алгоритм работы распараллеливающего компилятора

Методы распараллеливания применимы только при выполнении ряда ограничений на операторы тела цикла. Эти ограничения зависят от типа ЭВМ и метода распараллеливания.

К основным ограничениям на операторы тела цикла относятся следующие:

– тело цикла не содержит условных и безусловных переходов вне тела;

– внутри тела цикла передача управления осуществляется только вперед;

– тело цикла не содержит операторов ввода-вывода и обращений к подпрограммам;

– все индексные выражения линейны;

– отсутствует косвенная адресация вида X(N(I));

– индексы не изменяются в теле цикла;

– в теле цикла не используется простая неиндексированная переменная раньше, чем этой переменной в теле цикла присваивается некоторое значение (условие Рассела);

– тело цикла не содержит условных и безусловных переходов вне тела;

– внутри тела цикла передача управления осуществляется только вперед;

– тело цикла не содержит операторов ввода-вывода и обращений к подпрограммам;

– все индексные выражения линейны;

– отсутствует косвенная адресация вида X(N(I));

– индексы не изменяются в теле цикла;

– в теле цикла не используется простая неиндексированная переменная раньше, чем этой переменной в теле цикла присваивается некоторое значение (условие Рассела), например,

```
FOR i=1, N
  S:= S + X(i).
```

При выполнении приведенного фрагмента, начальное значение переменной S, должно быть присвоено вне цикла.

В табл. 3 приведены основные методы распараллеливания циклов.

Таблица 2

Методы распараллеливания циклов

Метод	**Описание**
Перестановка циклов	Внешний и внутренний циклы меняются местами
Локализация	Каждому процессу дается копия переменной
Распределение цикла	Один цикл расщепляется на два отдельных цикла
Слияние циклов	Два цикла объединяются в один
Разделение на полосы	Итерации одного цикла разделяются на два вложенных цикла
Развертка цикла	Тело цикла повторяется и выполняется меньше итераций
Развертка и сжатие	Комбинируются перестановка циклов, разделение на полосы и развертка
Разделение на блоки	Область итераций разбивается на прямоугольные блоки
Перекос цикла	Границы цикла изменяются, чтобы выделить параллельность

Распараллеливание ациклических участков основано на выявлении в исходной программе различных зависимостей по данным и анализа полученных результатов. Для этого строится ориентированный граф, вершины которого являются операторами исходной программы, а ребра обозначают зависимости по данным.

Особенности распараллеливания на системах с распределенной памятью:

– обмен данными между процессорами через коммуникационную систему требует значительного времени, поэтому вычислительная работа должна распределяться между процессорами крупными порциями;

– в отличие от многопроцессорных ЭВМ с общей памятью, на системах с распределенной памятью необходимо произвести не только распределение вычислений, но и распределение данных, наличие распределенных данных порождает проблему обеспечения эффективного доступа к удаленным данным;

– распределение вычислений и данных по процессорам вычислительной системы должно быть произведено таким образом, чтобы обеспечить сбалансированность загрузки процессоров.

Сложности распараллеливания:

– анализ зависимостей сложен для кода, использующего косвенную адресацию, указатели, рекурсию, вызовы функций по косвенности (например, виртуальные функции заранее неизвестного класса);

– циклы могут иметь неизвестное заранее количество итераций, усложняется выбор циклов, требующих распараллеливания;

– доступ к глобальным ресурсам тяжело координировать в терминах выделения памяти, ввода-вывода, разделяемых переменных.

Подходы для упрощения автоматического распараллеливания:

– дать программистам возможность добавлять к программе подсказки компилятору, чтобы влиять на процесс распараллеливания. Среди решений, можно указать High Performance Fortran для систем с распределенной памятью и OpenMP для систем с общей памятью;

– создать интерактивную систему компиляции, в работе которой принимал бы участие человек. Такие системы созданы в рамках подпроекта «SUIF Explorer» (проект SUIF – The Stanford University Intermediate Format compiler), в компиляторах Polaris и ParaWise (среда CAPTools);

– добавить в аппаратуру спекулятивную многопоточность (speculative multithreading).

Современные компиляторы с поддержкой автоматического распараллеливания:

– Oracle Solaris Studio (ранее Sun Studio);

– Intel C++ Compiler;

– GCC.

Ручное или автоматическое распараллеливание?

Спор, что лучше – ручное или автоматическое распараллеливание, такой же несостоятельный, как и выбор между ассемблером и языками высокого уровня – нужно и то и другое.

Разумеется, прикладная программа на ассемблере будет работать быстрее, но разработка займет слишком много времени, не говоря уже о модификации. За это время будет создано новое поколение процессоров, быстродействие которых погасит ассемблерные преимущества в ускорении.

Библиографический список

1. Воеводин, В. В. Вычислительная математика и структура алгоритмов [Текст]: учебник / В. В. Воеводин ; МГУ. – 2-е изд., стер. – Москва : Изд-во МГУ, 2010. – 166 с.

2. Гергель, В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем [Текст]: учеб. пособие / В. П. Гергель ; ННГУ. – Москва : Изд-во МГУ : Физматлит : Изд-во ННГУ, 2010. – 539, [4] с.

3. Илюшечкин, В. М. Операционные системы [Текст]: учеб. пособие / В. М. Илюшечкин. – Москва : Бином. Лаборатория знаний, 2009. – 109, [2] с. : ил.

4. Мельцов, В. Ю. Основы проектирования параллельных алгоритмов и программ [Электронный ресурс] : учеб. пособие / В. Ю. Мельцов, В. С. Князьков ; ВятГУ, научно-образоват. центр супервычислит. технологий и систем. – Киров, 2010.

5. Олифер, В. Г. Сетевые операционные системы [Текст]: учеб. / В. Г. Олифер, Н. А. Олифер. – Санкт-Петербург; Москва; Харьков : Питер, 2006. – 539 с. : ил.

6. Стивенс, У. UNIX: взаимодействие процессов [Текст] / У. Стивенс. – Санкт-Петербург: Питер, 2005. – 576 с.: ил.

7. Эндрюс, Г. Р.. Основы многопоточного, параллельного и распределенного программирования [Текст] / Г. Р. Эндрюс. – Москва: Вильямс, 2003. – 512 с. : ил.

Электронные ресурсы

8. Параллельные алгоритмы. Информатика и программирование: шаг за шагом [Электронный ресурс]: [сайт] / Каф. Информационных технологий Курганского гос. ун-та. – Режим доступа: http://it.kgsu.ru/ParalAlg/. – Загл. с экрана. – 21.01.2014.

9. Организация ЭВМ и систем. Дистанционное образование [Электронный ресурс] / Иркутский государственный технический ун-т; Дистанционное образование. – Режим доступа: http://paralichka85.px6.ru/1architecture/glava01_3.htm. – Загл. с экрана. – 21.01.2014.

10. Свойства параллельных алгоритмов [Электронный ресурс]: опубл. 8 окт. 2008 г., обновлено 31 окт. 2011 г. – Режим доступа: http://iproc.ru/parallel-programming/lection-3/. – Загл. с экрана. – 21.01.2014.

11. Гергель В. П. Теория и практика параллельных вычислений [Электронный ресурс]: [учеб. пособие] / В. П. Гергель; НОУ ИНТУИТ – Режим доступа: http://www.intuit.ru/department/calculate/paralltp/. – Загл. с экрана. – 21.01.2014.

12. Долинский М. Толкачев А. Обзор аппаратных и программных средств реализации параллельной обработки [электронный ресурс] – URL:http://www.kit-e.ru/articles/cad/2004_6_152.php.

Учебное издание

Должженкова Мария Львовна

Караваева Ольга Владимировна

ОСНОВЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

Подписано к использованию 16.01.2014. Заказ № 1365.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Вятский государственный университет».

610000, г. Киров, ул. Московская, 36, тел.: (8332) 64-23-56, http://vyatsu.ru