ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА»

В.П. Гергель, В.А. Фурсов

ЛЕКЦИИ ПО ПАРАЛЛЕЛЬНЫМ ВЫЧИСЛЕНИЯМ

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

САМАРА
Издательство СГАУ
2009

УДК СГАУ: 519.6(075)
ББК 22.19
Г 375

Рецензенты: д-р техн. наук, проф. Ю. Я. Б о л д ы р е в
канд. техн. наук, доц. С.Б. П о п о в

Гергель В.П.
Г 375 Лекции по параллельным вычислениям: учеб. пособие / *В.П. Гергель, В.А.Фурсов.* – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2009. – 164 с.

ISBN 978-5-7883-0739-8

Излагаются основы построения параллельных алгоритмов, ориентированных для реализации на многопроцессорных вычислительных системах. Приводятся примеры распараллеливания алгоритмов для решения простейших задач. Для большинства примеров приводятся оценки достижимого ускорения и показателей эффективности (загрузки процессоров). Примеры завершаются построением временной диаграммы параллельного алгоритма, которая, по существу, является расписанием реализации параллельного алгоритма и исходной моделью для составления соответствующей программы.

Учебное пособие ориентировано в основном на начальную подготовку, например, студентов в рамках программ подготовки бакалавров, слушателей ФПКП и других категорий учащихся, приступающих к освоению технологий параллельных вычислений.

Учебное пособие может быть полезно также при проведении прикладных научных исследований, выполнении курсовых и дипломных проектов по физико-математическим и техническим направлениям подготовки.

УДК СГАУ: 519.6(075)
ББК 22.19

ISBN 978-5-7883-0739-8

ВВЕДЕНИЕ

В последние годы усиливается внимание к использованию высокопроизводительной вычислительной техники в научных исследованиях и образовании. Связано это с тем, что во многих областях знаний фундаментальные научные исследования связаны с необходимостью проведения масштабных численных экспериментов. Общая тенденция состоит в том, что происходит концентрация вычислительных ресурсов в центрах коллективного пользования и развитие инфраструктуры удаленного доступа с использованием средств телекоммуникаций. Это требует, чтобы большое число специалистов в разных областях владели современными технологиями работы как на многопроцессорных системах удаленных суперкомпьютерных центров, так и на персональных мини-кластерах.

В последние годы появились замечательные книги [2,3], которые позволили в какой-то степени снять проблему недостатка в учебных пособиях. Вместе с тем массовая подготовка специалистов, в особенности переход на двухступенчатую подготовку бакалавр-магистр (с крайне малым числом часов аудиторных занятий), потребовала издания более компактного и вместе с тем достаточно доступного для самостоятельной подготовки пособия. Другая важная отличительная черта настоящего учебного пособия состоит в следующем.

Подготовка задачи к решению на параллельном компьютере заключается в составлении плана вычислений и составлении кода. Для составления эффективного плана вычислений обычно требуется глубокое понимание существа задачи, а для написания соответствующего некоторому плану вычислений эффективного кода требуется глубокое знание особенностей построения вычислительной системы. В частности, требуется знание топологии связей, типа процессора, относительных объемов памяти кэшей различных уровней и др. К сожалению, специалистов, одинаково глубоко владеющих как предметом исследований в «своей» области знаний, так и знаниями в области высокопроизводительных вычислительных систем, не так много.

В последние годы предпринимаются усилия по расширению подготовки специалистов в области решения задач на многопроцессорных системах в рамках бакалавриата. Представляется, что решение этой проблемы более эффективно путем полной «развязки» параллельных вычислений и параллельного программирования. Настоящее учебное пособие является такой попыткой. В нем подчеркнуто не используются термины «программа» и «программирование», а анализ параллельных алгоритмов завершается составлением временной диаграммы его выполнения. Для того чтобы сделать диаграммы полностью понятными программисту, который не владеет существом прикладной задачи, авторы «модернизировали» обычно используемые диаграммы [1]. В частности, в них в явном виде введены нумерация процессоров и связи между процессами в виде номеров входных, выполняемых и выходных операторов.

Указанная особенность построения учебного пособия, кроме прочего, позволяет начать изучение вопросов организации параллельных вычислений до того, как изучен какой-либо алгоритмический язык. Это создает дополнительные удобства в реализации учебного плана. Такой подход может представлять интерес и для научных работников. Процесс создания параллельных приложений может быть более эффективным, если организовать своеобразный конвейер: математическая постановка задачи (1), структурный анализ алгоритма и построение плана вычислений (2), написание кода (3), отладка программы (4). Ясно, что методики, обеспечивающие возможность реализовывать каждый из указанных этапов независимо, обеспечат более быстрое и притом более качественное решение задачи.

Учебное пособие ориентировано в основном на начальную подготовку студентов в рамках программ подготовки бакалавров, слушателей ФПКП и других категорий учащихся, приступающих к освоению технологий параллельных вычислений, но может быть полезным также при выполнении курсовых и дипломных проектов, а также при проведении прикладных научных исследований, связанных с подготовкой параллельных приложений.

Лекция 1

История и значение вычислений

1.1. Предыстория ЭВМ и связанные с ней выводы

В последние годы усиливается внимание к использованию высокопроизводительной вычислительной техники. Возникает естественный вопрос: является ли это объективной потребностью общества или это лишь модное движение. Как долго продлится эта затянувшаяся гонка производительностей? Не является ли эта гонка своего рода соревнованием престижности, которое лишь истощает ресурсы общества. Для того чтобы ответить на эти вопросы, полезно «заглянуть» в историю вычислений.

Кратко перечислим некоторые наиболее известные, типы инструментов и машин, использовавшихся до эпохи ЭВМ и основанных на механических и электромеханических принципах [4].

Механические вычислители:

- Абак и счеты, логарифмическая линейка (1617).
- Вычислительная машина Паскаля (1642).
- Разностная и аналитическая машина Бэбиджа (1834).
- Арифмометр Лейбница (1673).
- Арифмометр Однера (1876) («Феликс»).

Электромеханические вычислители:

- Табулятор Холлерита(1887).
- Машинно-счетные станции.
- Проекты Конрада Цузе (Германия, 1938-1945).
- Проект MARK-1 (Эйкен, IBM, 1939-1944).
- Проекты Джорджа Стибица (Bell Laboratories, 1939-1947).

Что же двигало создателями этих машин и инструментов? Известно, что поистине революционный шаг: создание первой вычислительной машины,

юный Блез Паскаль совершил, движимый желанием облегчить арифметические вычисления отца – сборщика налогов. Конечно, для этого он должен был обладать необходимыми материальными ресурсами. Надо полагать, что семья сборщика налогов, по тем временам, была сравнительно обеспеченной, чтобы позволить сыну строить вычислительную машину.

Дальнейшие весьма важные усовершенствования вычислительной машины Паскаля были выполнены Лейбницем и Однером. Они также были продиктованы потребностями практики, в частности необходимостью совершенствования учета при сборе налогов. Значение, которое сыграли эти усовершенствования для ускорения вычислений, трудно переоценить. В СССР даже в 1969 году арифмометров «Феликс» было произведено 300000 шт.

Тем не менее, арифмометры не смогли обеспечить решение всех задач такой, например, актуальной задачи того времени – переписи населения. Например, итоги переписи 1880 года, когда население США составляло около 50 млн. человек, были получены только через 7,5 лет. В ответ на эти новые вызовы появляется «машина для переписи населения» – табулятор Холлерита. В табуляторе на каждый объект переписи заводилась 80-колоночная перфокарта, в которой с помощью перфоратора в определенных позициях делались отверстия, отвечающие определенным значениям признаков. Применение табуляторов в 1890 году позволило сократить сроки обработки результатов переписи до двух лет. После этого спрос на машины для переписи не только в США, но и в других странах резко возрос, что обеспечило успех становления компании IBM.

Следующим важным практическим шагом в развитии вычислений было создание электромеханической машины MARK-1. При создании этой машины Говард Эйкен опирался на идеи неосуществленного проекта Бэббиджа. Это служит лишним подтверждением того факта, что новые идеи в развитии вычислительной техники находят практическое воплощение при появлении спроса на высокопроизводительные вычисления. Действительно, одной из основных причин успеха проекта MARK-1 была его востребованность: машина создавалась в

интересах Военно-морского флота и использовалась для расчета артиллерийских таблиц. Таким образом, потребность плюс ресурсы (в создание машины MARK-1 компания IBM вложила 500000 долларов, проект также поддерживался правительством США и командованием Военно-морского флота) оказались решающими факторами в развитии высокопроизводительных, по тому времени, вычислений.

В настоящее время во многих областях знаний также возникает необходимость проведения масштабных вычислений, реализация которых возможна только на суперкомпьютерах. Если внимательно посмотреть на состояние современных наук, опирающихся на математику, нетрудно заметить следующую тенденцию. Ограничиваются возможности получения результатов только при помощи теоретического исследования. Связано это со сложностью аналитических моделей реальных процессов и явлений. При этом сравнимая, а чаще более высокая точность, может быть достигнута с использованием учитывающих большее число факторов, численных моделей. По-видимому, «развитие математики в определенный период времени отражает природу вычислительных средств, доступных в этот момент, в значительно большей степени, чем можно было бы предположить» [8, с.152].

Указанные тенденции вовсе не являются изобретением собственно математиков. Напротив, математики в данном случае пытаются найти рациональный ответ тем вызовам, которые возникли в современном обществе. Эти вызовы объективно существуют на нескольких уровнях.

1. Глобальные проблемы развития и сохранения цивилизаций (межгосударственный уровень). Примерами задач этого уровня являются моделирование климата, экологических процессов, прогнозирование космических явлений и др. Эти проблемы вполне осознаются сообществами ученых разных стран, а исследования поддерживаются развитыми странами и международными фондами.

2. Задачи поддержания высокого уровня национальной безопасности (государственный уровень). В данном случае высокопроизводительные вычис-

лительные системы являются основой инфраструктуры, обеспечивающей военно-экономическое преимущество и оперативное реагирование на возможные угрозы.

3. Задачи расширения рынка и повышения конкурентоспособности товаров и услуг (производственный уровень). Известно, для того чтобы захватить рынок, надо опередить конкурентов. Наряду с совершенствованием организации производства, в данном случае основной резерв – это сокращение сроков проектирования и отработки новых образцов за счет моделирования на высокопроизводительных ЭВМ.

Таким образом, гонка производительностей ЭВМ является одним из необходимых факторов успешного развития постиндустриального общества.

1.2 История ЭВМ

1.2.1. Некоторые зарубежные решения

Юридический приоритет создания первой ЭВМ принадлежит Джону Атанасову (Atanasoff, John; 1903-1995). В 1939 г. он с аспирантом Клиффордом Берри (Berry, Clifford Edward; 1918-1963) приступил к постройке машины, предназначенной для решения системы алгебраических уравнений с 30 неизвестными (ABC — Atanasoff-Berry Calculator). Проект не был завершен (в этом разделе используются сведения, приведенные в [4]).

Первая работающая ЭВМ ENIAC (Electronic Numerical Integrator And Calculator) была создана в 1945 г. в Пенсильванском университете. Длина 26 м, высота 6 м, масса 30 т. 18 000 ламп, 1500 реле, потребляемая мощность 150 кВт.

В годы Второй мировой войны под руководством выдающегося математика Алана Тьюринга была также построена специализированная электронная вычислительная машина Colossus. Она насчитывала 2000 радиоламп и обрабатывала 25000 симв./с

В 1953 г. к производству ЭВМ общего назначения подключилась фирма IBM, выпустив серийную IBM-701. Быстродействие около 10000 оп./с, ОЗУ 2К

36-разрядных слов. Следующим, поистине революционным, шагом было создание вычислительной системы (mainframes) IBM-360 (апрель, 1964). Важнейшие особенности IBM System/360 следующие:

- микросхемная элементная база;
- микропрограммное управление;
- внешняя память на магнитных дисках;
- дисплейные терминалы;
- открытая масштабируемая архитектура.

Важной отличительной чертой системы IBM-360 являлась возможность создавать комплексы «под заказ» в широком диапазоне производительностей. Таким образом, отпала необходимость создавать майнфреймы. Это побудило производителей ЭВМ искать новые ниши как в направлении миниатюризации (мини-ЭВМ), так и в области создания супер-ЭВМ.

Первая супер-ЭВМ CDC-6600 была разработана Сеймуром Креем (Cray, Seimour; 1925-1996) и построена в фирме Control Data Corporation (1963 г.) Разрядность 64 бита, быстродействие 3 млн. оп./с.

Значительный скачок произошел также и в создании рынка мини-ЭВМ. Наиболее известные решения в этом направлении: мини-ЭВМ PDP: -5 (1963), -8 (1965) -11 (1970): первые модели ПК компаний MITS и Apple (Altair-1975, Apple-II, 1977); IBM-совместимые ПК (IBM PC XT - 1983, PC AT – 1984); Apple: Macintosh (1984) .

Появление на рынке сравнительно дешевых и достаточно высокопроизводительных персональных компьютеров, с одной стороны, и развитие высокопроизводительных сетевых технологий, с другой стороны, обусловило появление высокопроизводительных вычислительных систем – кластеров. Круг замкнулся.

Основа этих достижений, вне всякого сомнения, была создана успехами в развитии элементной базы. Ниже приводится наглядная иллюстрация закона Мура, согласно которому количество элементов на одном кристалле удваивается каждые полтора года. Несмотря на то, что многими исследователями

ощущается предел, обусловленный физическими законами, этот закон пока выполняется.

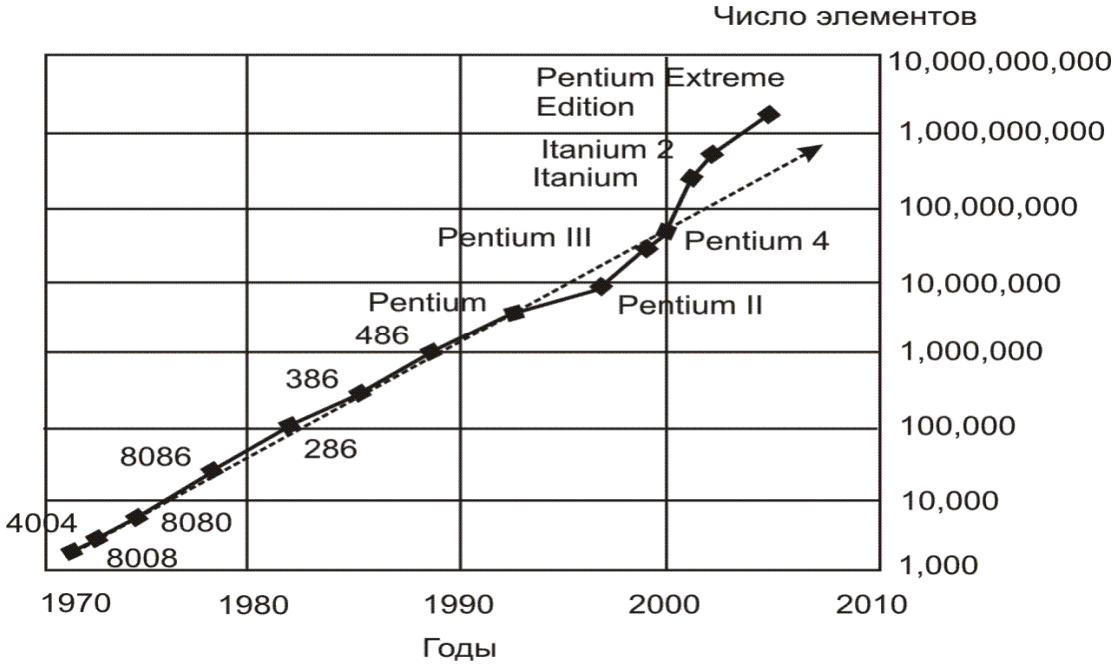


Рис. 1.1. Закон Мура

1.2.2 Развитие отечественных ЭВМ

В развитии отечественной вычислительной техники можно выделить следующие важнейшие этапы: зарождение (1948 - 1952 годы); расцвет (1950-е – 1960-е годы); подражание (1970-е – 1980-е годы); кризис (1990-е годы).

Этап зарождения интересен и поучителен для нынешнего времени. Началось с того, что в 1950 году академик М.А. Лаврентьев написал письмо И.В. Сталину, в котором обратил внимание на важность вычислительных машин для обороны страны. В результате в удивительно короткие сроки были подготовлены соответствующие постановления правительства и началась разработка ЭВМ одновременно в Академии наук СССР и Министерстве машиностроения и приборостроения.

В результате уже в 1951 году появилась первая отечественная ЭВМ МЭСМ (1951 г., Киев), гл. конструктор С.А. Лебедев. Она имела следующие характеристики: 6000 электронных ламп, быстродействие 50 оп./с, ОЗУ 94 16-разрядных слов, потребляемая мощность 15 кВт, занимаемая площадь – 60 кв.м.

Первая полномасштабная отечественная ЭВМ БЭСМ (1952 г., Москва, ИТМ и ВТ), гл. конструктор С.А. Лебедев. Характеристики: 5000 ламп, быстродействие 8000 оп./с, ОЗУ 1К 39-разрядных слов, ПЗУ 1К слов, потребляемая мощность 30 кВт, занимаемая площадь 100 кв. м.

Первая отечественная серийная ЭВМ «Стрела» (1953 г.), гл. конструктор Ю.Я. Базилевский, зам. гл. конструктора Б.И. Рамеев. Характеристики: 6200 ламп, 60000 полупроводниковых диодов. Быстродействие 2000 оп./с, ОЗУ на потенциалоскопах (43-разрядные слова), потребляемая мощность 150 кВт, занимаемая площадь 300 кв. м. С 1953 до 1956 г. выпущено 7 экземпляров этих машин.

Серийная ЭВМ общего назначения М-20 (1958 г.), гл. конструктор С.А.Лебедев, имела следующие характеристики: 2600 ламп, ОЗУ 4К 45-разрядных слов, быстродействие 20 000 оп./с (в то время самое большое в Европе).

Серийная ЭВМ малого класса Урал -1 (1957 г., НИИММ, г. Пенза): 700 электронных ламп, ОЗУ на магнитном барабане (1024 36-разрядных слов), быстродействие 100 оп./с. Семейство полупроводниковых ЭВМ среднего класса Урал -11,- 14, -16 (1964-1969 годы), гл. конструктор Б.И. Рамеев, имело унифицированную архитектуру, быстродействие от 45 до 100 тыс. оп./с.

ЭВМ БЭСМ-6 (1968 г.) наиболее мощная из отечественных машин 2-го поколения, гл. конструктор С.А. Лебедев. Характеристики: 60 тыс. транзисторов, 180 тыс. диодов, быстродействие 1 млн оп./с, ОЗУ от 32К до128К 48-разрядных слов. Производилась до 1987 г, всего выпущено 355 экз.

Машина инженерных расчетов МИР-1 (1966 г., ИК АН УССР, г. Киев), гл. конструктор В.М.Глушков. Машина имела оригинальное многоступенчатое программное управление. Куплена фирмой IBM (1967 г.).

Полупроводниковая ЭВМ «Минск-32» (1968 г.) – последняя из семейства машин «Минск». Гл. конструктор В.В. Пржиялковский, характеристики: 30-35 тыс. оп./с, ОЗУ 64К 38-разрядных слов. До 1975 г. выпущено 2889 экземпляров.

Для сопоставления темпов развития вычислительной техники в России и за рубежом ниже приводится таблица, в которую включены некоторые наиболее успешные решения, которые оказали большое влияние на последующие этапы развития.

Таблица 1.1 Сопоставление отечественных и зарубежных ЭВМ

Год	ЭВМ в СССР	Зарубежный образец
1951	МЭСМ 50 оп./с, ОЗУ 94 16-разрядных слова	UNIVAC-1 2000 оп./с, ОЗУ 1000 слов по 12 десятичных разрядов
1952	БЭСМ 5000 ламп, 8000 оп./с, ОЗУ 1К 39-разрядных слов, ПЗУ 1К слов	IBM-701. 10000 оп./с, ОЗУ 2К 36-разрядных слов (1952)
1953	«Стрела» 6200 ламп, 60000 п/п диодов. 2000 оп./с	IBM-702
1958	М-20 2600 ламп, ОЗУ 4К 45-разрядных слов 20 000 оп./с	IBM-7030 Stretch (1959 г.), 500 тыс. оп./с, ОЗУ до 256К 64-битовых слов
1964	Урал -11,- 14, -16 45 до 100 тыс. оп./с.	IBM System/360 (1964). Можно собрать машину любой мощности Появление чипов с числом 10 (1964) и 100 (1970) элементов Создание чипов с числом транзисторов до 65 тыс. (1975)
1968	БЭСМ-6 60 тыс. транзисторов, 180 тыс. диодов,1 млн оп./с, ОЗУ от 32К до128К 48-разрядных слов	
1968	«Минск-32» 30-35 тыс. оп./с, ОЗУ 64К 38-разрядных слов	

Таблица 1.1. показывает, что до начала этапа копирования, хотя и имело место отставание, оно не было катастрофическим и носило характер динамической системы. Заметим, что успехи в развитии вычислительной техники нельзя рассматривать изолированно. Они напрямую связаны с потребностями развития экономики СССР послевоенного периода, а достижения в других областях были и причиной и следствием.

Напомним некоторые из этих достижений:

- август 1949 г. - успешное испытание в СССР атомной бомбы;
- 4 ноября 1957 г. – в СССР запущен первый искусственный спутник Земли;
- 12 сентября 1959 г . - запуск космического аппарата «Луна-2», достигшего поверхности Луны;

- 12 апреля 1961 г . - Юрий Гагарин на космическом корабле «Восток» совершил первый в мире полёт в космос;
- 31 января 1966 г - космический аппарат «Луна-9» впервые в мире осуществил мягкую посадку на Луну;
- 12 июня 1967 г. – космический аппарат «Венера-4» впервые осуществил плавный спуск в атмосфере Венеры.

Нетрудно заметить, что период успехов в развитии вычислительной техники (50-е годы) удивительным образом совпадает со впечатляющими успехами СССР в других областях: космических исследованиях, развитии атомной энергетики и др.

1.2.3 Этап создания единой системы (ЕС) ЭВМ

В конце 1960-х годов советское руководство приняло решение о прекращении производства оригинальных отечественных ЭВМ и развертывании работ по созданию Единой системы ЭВМ (ЕС ЭВМ) социалистических стран на базе архитектуры IBM System/360, а также Системы малых машин (СМ ЭВМ) на базе архитектуры Hewlett Packard и PDP-11. Всего за период с 1970 по 1990 год было произведено более 15000 вычислительных машин разной производительности ряда ЕС ЭВМ.

Этот этап имел как некоторые положительные, так и отрицательные последствия. В качестве положительных результатов следует отметить следующие: удалось несколько сократить технологическое отставание; пользователи могли использовать уже существовавшие к тому времени довольно значительные библиотеки для ЭВМ. Кроме того, наличие переводной литературы позволило в короткие сроки организовать выпуск специалистов.

Вместе с тем это имело также значительные отрицательные последствия. Несмотря на большую затратность проекта, скачка так и не произошло, поскольку технологическая база оставалась слабой, но самое главное были разрушены сложившиеся научные школы. Некоторые отрицательные последствия,

например, зависимость от заимствованного программного обеспечения остро ощущается до сих пор. Это существенно сдерживает создание и реализацию сложных систем во многих областях.

1.3 Новая стратегическая инициатива США и перспективы развития суперкомпьютерной отрасли в России

Успехи США в разработке ЭВМ и программного обеспечения, конечно, не были случайными. Они явились следствием микропроцессорной революции, которая произошла в результате беспрецедентных по масштабу инвестиций правительства США в электронное машиностроение. Следствием этого является то, что до сих пор ведущие производители вычислительной техники (IBM, HP, Intel) сосредоточены в США. Тем не менее именно США выступили с новой стратегической инициативой в области вычислительной техники.

В мае 2005 года Консультативный комитет по информационным технологиям при Президенте США представил Джорджу Бушу аналитический доклад под названием: «Вычислительная наука: обеспечение конкурентоспособности Америки» [11]. В этом докладе содержатся результаты анализа потенциальных возможностей развития науки, промышленности и экономики, их связи с достижениями в области информатики (Computational Science). По мнению авторов, это одна из наиболее важных областей, которая является существенным фактором инновационного развития общества в XXI веке. Авторы доклада подчеркивают, что именно эта область в настоящее время является важнейшим фактором для обеспечения научного лидерства, конкурентоспособности и национальной безопасности США.

Под Computational Science (вычислительными науками) авторы доклада понимают:

1. Модели, вычислительные алгоритмы и моделирующие программы для решения научных (биология, физика, социальные исследования и др.), технических и гуманитарных проблем.

2. Оптимизация и развитие аппаратных, программных и сетевых средств обработки данных, обеспечивающих решение указанных выше вычислительных задач.
3. Компьютерная инфраструктура для поддержки научных исследований и решения технических задач.

В докладе показано, что развитие «вычислительной науки» создает уникальные возможности для проведения научных исследований, в т.ч. для исследования процессов и явлений в реальном масштабе времени с использованием высокопроизводительных суперкомпьютеров. Предполагается, что именно вычислительная наука, наряду с теоретическими исследованиями и физическим экспериментом, будет являться главной основой всей научной методологии XXI века (рис. 1.2).

Рис. 1.2. Иллюстрация интегрирующей роли вычислительных наук

Основные меры, направленные на поддержание национальной безопасности, которые были сформулированы в документе:

1. Выработка перспективных направлений и поддержание высокого уровня исследований в области вычислительных наук.

2. Создание инфраструктуры для поисковых исследований.

3. Создание национального репозитария данных и программных продуктов.

4. Создание и поддержка национальных высокопроизводительных вычислительных центров.

Новая американская стратегическая компьютерная инициатива содержит серьезный анализ новых вызовов XXI века мировому сообществу. Эти вызовы затронут все без исключения страны мира и повлекут за собой новый этап конкурентной борьбы в области науки, образования и информационных технологий. По-видимому, вычислительные науки, понимаемые в широком смысле этого слова, т.е. включающие как собственно теорию и методы алгоритмизации вычислительных процессов, так и вопросы создания аппаратно-программного обеспечения, будут играть ведущую интегрирующую роль в этом процессе.

К сожалению, Россия отстает от основных лидеров – стран, где установлены самые мощные суперкомпьютеры. Поэтому весьма своевременной представляется задача, поставленная президентом Дмитрием Медведевым на заседании комиссии по модернизации и технологическому развитию российской экономики в г. Сарове Нижегородской области: «для технологического и инновационного развития Россия должна сократить отставание от лидеров в сфере вычислительных технологий. Прежде всего это касается создания суперкомпьютеров и так называемых grid-технологий» (Российская газета - Федеральный выпуск №4962 (138) от 29 июля 2009 г.). В частности, в этом городе во Всероссийском НИИ экспериментальной физики в 2011 году должен появиться первый отечественный суперкомпьютер, способный выполнять одновременно квадриллион операций.

Дмитрий Медведев очертил круг основных направлений работы в сфере суперкомпьютеров:

«Есть пять задач, которые надо поставить и решить.

- Первая – определить приоритетное направление использования суперкомпьютерных и grid-технологий в области обеспечения национальной безопасности и социально-экономического развития страны.

- Вторая задача заключается в подъеме уровня отечественной электронной компонентной базы до потребностей производства суперкомпьютеров.
- Третье очевидное условие – сформировать нормативно-правовую базу применения суперкомпьютеров.
- Четвертое – создать условия для построения grid-сетей прежде всего в научно-образовательной сфере.
- Пятое – организовать специальную систему подготовки специалистов ведущих вузов страны»

Президент России подчеркнул, что «применение суперкомпьютеров во всем мире осуществляется при решающей финансовой и организационной поддержке государства, и Россия готова идти по общепринятому пути». При этом важно также одновременно развивать технологическую культуру разработки новых образцов техники с использованием современной суперкомпьютерной техники. «Мы должны всячески стимулировать ее востребованность не потому, что это модная тема, а просто потому, что по-другому не создать конкурентоспособную продукцию» – считает глава государства.

Опираясь на опыт передовых стран, в России предполагается создание иерархических вычислительных комплексов – так называемых grid-систем. В основе grid-систем должно находиться большое количество относительно небольших компьютеров с очень широкой географией, связанных в высокопроизводительные сети. А вершиной пирамиды станет мощная ЭВМ для проведения наиболее сложных фундаментальных исследований. Для того чтобы ликвидировать отставание, предполагается развитие сотрудничества с США и другими странами по вопросам развития высокопроизводительных вычислений. Использование имеющегося опыта, в том числе зарубежных стран, в деле создания своей суперкомпьютерной отрасли имеет большое значение для реализации стратегии национальной безопасности и развития концепции долгосрочного социально-экономического развития России.

1.4 Проблемы высокопроизводительных вычислений и задачи курса

В последние годы во всем мире усиливается внимание к использованию высокопроизводительной вычислительной техники. С одной стороны, растет число пользователей персональных миникластеров, с другой стороны, происходит концентрация мощных вычислительных ресурсов в центрах коллективного пользования и развитие инфраструктуры удаленного доступа с использованием средств телекоммуникаций.

С другой стороны, в настоящее время всеми осознается факт, что дальнейшее повышение производительности компьютеров только за счет улучшения характеристик элементов электроники достигло предела, определяемого физическими законами. Дальнейшее повышение производительности возможно лишь за счет распараллеливания процессов обработки информации. Однако оказалось, что построение параллельных вычислительных процессов, обеспечивающих достижение максимальной производительности, является важной самостоятельной проблемой.

Сокращение времени выполнения большого объема работ путем разбиения на отдельные работы, которые могут выполняться *независимо* и *одновременно,* используется во многих отраслях. В области вычислительной техники это достигается путем увеличения числа одновременно работающих процессоров. Вместе с тем известно, что перенос обычной программы на многопроцессорную вычислительную систему может не дать ожидаемого выигрыша в производительности. Более того, в результате такого переноса программа может работать медленнее.

Опыт показывает, что написать эффективную параллельную программу или приспособить уже имеющуюся последовательную программу для параллельных вычислений чрезвычайно трудно. Связано это с тем, что переход к использованию многопроцессорных систем характеризуется принципиально новым содержанием. В данном случае важнейшим оказывается структурный анализ алгоритма, выявление его внутреннего параллелизма. Этот анализ требует

глубокого понимания существа задачи. Часто же проблема заключается в том, что программист не вполне ясно понимает алгоритм решения задачи, а математик не может поправить программу, т.к. не обладает достаточной квалификацией в области программирования.

Эта трудность может быть преодолена, если использовать модель представления алгоритма, которая понятна как математику, так и программисту. При этом математик и программист могут работать в значительной степени независимо, взаимодействуя лишь в рамках используемой модели.

Представляется, что такое «разделение труда» позволит существенно сократить сроки решения сложных задач. Это освобождает прикладного математика высокой квалификации от необходимости написания кодов для решения своих задач. Математик может выполнить структурный анализ алгоритма, выявить имеющийся в нем параллелизм и представить в виде модели, понятной программисту. С другой стороны, программист, не вникая в существо задачи, используя лишь модель параллельных вычислений, может написать весьма эффективный код.

Именно поэтому центральная идея настоящего пособия состоит в том, чтобы изучать вопросы построения параллельных алгоритмов и вопросы программирования независимо. В частности, цель настоящего курса лекций дать основные сведения в области параллельных вычислений, касающиеся вопросов анализа эффективности параллельной структуры алгоритма. При этом основное внимание будет уделено построению моделей и схем параллельных вычислений, а также анализу достижимых ускорения и эффективности для различных прикладных вычислительных задач. Изучение этих вопросов, следуя изложенной выше точке зрения, будет доводиться до описания параллельного алгоритма в виде блок-схем, графов и диаграмм, которые понятны программисту и могут использоваться им для написания параллельной программы.

Лекция 2

Архитектура параллельных вычислительных систем

2.1 Введение

Представьте себе на минуту, что вы музыкант и ваша задача подготовить некоторое музыкальное произведение для исполнения оркестром (на музыкальном языке это называется написать партитуру для оркестра). Что для этого надо знать? По-видимому, надо знать, какие группы инструментов существуют (духовые, струнные, ударные и др.). Затем надо знать, какие особенности звучания имеют отдельные виды инструментов внутри каждой группы и, наконец, как они называются (ведь каждая партия должна быть адресована конкретному инструменту). Композитор, пишущий партитуру для оркестра, может впоследствии и не дирижировать оркестром. Вполне вероятно, это будут делать другие музыканты, притом с различными стилями дирижирования, всякий раз придающими произведению своеобразную окраску.

Этот музыкальный пример мы привели для того, чтобы еще раз подчеркнуть основную идею настоящего учебного пособия: рассмотреть проблемы параллельных вычислений, не затрагивая вопросы программирования. Мы рассматриваем процесс подготовки параллельного решения задачи на многопроцессорной системе как отдельный этап подготовки описания (партитуры) параллельного алгоритма не некотором языке блок-схем и/или графов. Написание параллельной программы рассматривается как завершающий этап (дирижирование оркестром), обеспечивающий эффективную реализацию задуманного алгоритма (произведения), возможно, с учетом его конкретных особенностей (состава, квалификации музыкантов и др.).

Другими словами, при описании параллельных алгоритмов не обязательно знать конкретный язык программирования, на котором будет реализована программа. Однако надо хорошо знать особенности вычислительной системы, на которой будет реализован алгоритм (типы используемых вычислительных уз-

лов, производительность и др.). Если у разработчика есть выбор, можно поставить задачу построения наиболее эффективного параллельного алгоритма, подобрав типы вычислителей, наиболее полно реализующие его особенности. Для этого необходимо ясно представлять потенциальные возможности различных архитектур.

Таким образом, изучение возможных типов архитектур, характеристик и способов организации вычислительной системы, на которой предполагается реализация разрабатываемого параллельного алгоритма, является необходимым этапом. Настоящая лекция содержит краткий обзор наиболее популярных архитектур в том минимальном объеме, который может потребоваться при разработке параллельного алгоритма.

2.2 Классификация компьютерных систем

Существуют различные классификации, преследующие разные цели. При разработке параллельного алгоритма наиболее важно знать тип оперативной памяти, т.к. она определяет способ взаимодействия между частями параллельной программы. В зависимости от организации подсистем оперативной памяти параллельные компьютеры можно разделить на следующие два класса.

Системы с разделяемой памятью (*мультипроцессоры*), у которых имеется одна виртуальная память, а все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти (uniform memory access или UMA). По этому принципу строятся векторные параллельные процессоры (parallel vector processor или PVP) и симметричные мультипроцессоры (symmetric multiprocessor или SMP).

Системы с распределенной памятью (*мультикомпьютеры*), у которых каждый процессор имеет свою локальную оперативную память, а у других процессоров доступ к этой памяти отсутствует.

При работе на компьютере с распределенной памятью необходимо создавать копии исходных данных на каждом процессоре. В случае системы с разде-

ляемой памятью достаточно один раз задать соответствующую структуру данных и разместить ее в оперативной памяти.

Указанные два типа организации памяти могут быть реализованы в различных архитектурах. Рассмотрим различные классификации параллельных компьютеров, указывая там, где это имеет значение, способ организации оперативной памяти.

Исторически наиболее ранней является классификация *М. Флинна* (1966). Классификация основана на понятии *потока*, под которым понимается последовательность команд или данных, обрабатываемых процессором. На основе числа потоков команд и потоков данных выделяют четыре класса архитектур:

- SISD (Single Instruction stream/Single Data stream) – один поток команд и один поток данных;
- SIMD (Single Instruction stream/Multiple Data stream) – один поток команд и множество потоков данных;
- MISD (Multiple Instruction stream/Single Data stream) – множество потоков команд и один поток данных;
- MIMD (Multiple Instruction stream/Multiple Data stream) – множество потоков команд и множество потоков данных.

В настоящее время подавляющее число «серьезных» компьютеров реализуется в классе MIMD-архитектур. При этом рассматривают следующие основные подклассы.

Векторно-конвейерные компьютеры, в которых используется набор векторных команд, обеспечивающих выполнение операций с массивами независимых данных за один такт. Типичным представителем данного направления является линия «классических» векторно-конвейерных компьютеров CRAY.

Массово-параллельные (чаще называемые также *массивно-параллельные*) компьютеры с *распределенной* памятью. В данном случае микропроцессоры, имеющие каждый свою локальную память, соединяются посредством некоторой коммуникационной среды. Достоинство этой архитектуры – возможность

наращивать производительности путем добавления процессоров. Недостаток – большие накладные расходы на межпроцессорное взаимодействие.

Симметричные мультипроцессоры (SMP) состоят из совокупности процессоров, имеющих разделяемую общую память с единым адресным пространством и функционирующих под управлением одной операционной системы. Недостаток – число процессоров, имеющих доступ к общей памяти, нельзя сделать большим. Существует предел наращивания числа процессоров, превышение которого ведет к быстрому росту потерь на межпроцессорный обмен данными.

Кластеры образуются из вычислительных модулей любого из рассмотренных выше типов, объединенных системой связи или посредством разделяемой внешней памяти. Могут использоваться как специализированные, так и универсальные сетевые технологии. Это направление, по существу, является комбинацией предыдущих трех.

Еще раз подчеркнем, что наиболее важным при разработке параллельного алгоритма является деление на компьютеры с общей и распределенной памятью. Для компьютеров с общей памятью пользователю не нужно заботиться о распределении данных, достаточно предусмотреть лишь затраты на выбор необходимых данных из этой памяти. При реализации параллельного алгоритма на компьютерах с распределенной памятью необходимо продумать рациональную, с точки зрения потерь на обмен данными, схему их размещения. Далее дадим более подробную характеристику каждого из указанных выше подклассов компьютеров.

2.3 Детализация архитектур по достижимой степени параллелизма

Выше мы рассмотрели основные классы параллельных компьютеров, отличия между которыми следует учитывать в первую очередь при построении параллельных алгоритмов. Поскольку подавляющее число архитектур реализуется в классе MIMD, требуется более детальная классификация, которая, кроме прочего, позволяла бы также давать оценку достижимой степени параллелизма.

Одна из таких систематизаций MIMD-компьютеров дана *Р. Хокни* [2]. Основная идея классификации состоит в том, что множественный поток команд может быть обработан либо по конвейерной схеме в режиме разделения времени, либо каждый поток обрабатывается своим устройством.

В соответствии с этим различают следующие MIMD – компьютеры:

- конвейерные;
- переключаемые (с обей памятью и распределенной памятью);
- сети, реализованные в виде: регулярной решетки, гиперкуба, иерархической структуры и изменяемой конфигурации.

Следующая классификация [2] *Т. Фенга* позволяет также строить оценки достижимой степени параллелизма. Она основана на двух характеристиках:

- число n бит в машинном слове, обрабатываемых параллельно;
- число слов m, обрабатываемых одновременно вычислительной системой.

Произведение $P=m\times n$, определяющее интегральную характеристику параллельности архитектуры, называют *максимальной степенью параллелизма* вычислительной системы. Введение этой *единой числовой метрики* для всех типов компьютеров позволяет сравнивать любые два компьютера между собой. Однако в данном случае не делается акцент на том, за счет чего компьютер может одновременно обрабатывать более одного слова.

С точки зрения указанной классификации возможны следующие варианты построения компьютера:

- разрядно-последовательные, пословно-последовательные (n=1, m=1);
- разрядно-параллельные, пословно-последовательные (n>1, m=1);
- разрядно-последовательные, пословно-параллельные (n=1, m>1);
- разрядно-параллельные, пословно-параллельные (n>1, m>1).

Подавляющее большинство вычислительных систем принадлежит к этому, последнему, классу.

Классификация *В. Хендлера* [2]. В основе этой классификации явное описание параллельной и конвейерной обработки. При этом различают три уровня обработки данных:

- уровень выполнения программы;
- уровень выполнения команд;
- уровень битовой обработки.

На каждом уровне допускается возможность конвейерной обработки. Таким образом, в общем случае каждый компьютер может быть охарактеризован следующими шестью числами:

k - число процессоров;

k' - глубина макроконвейера;

d - число АЛУ в каждом процессоре;

d' - глубина конвейера из функциональных устройств АЛУ;

w - число разрядов в слове, обрабатываемых в АЛУ параллельно;

w' - число ступеней в конвейере функциональных устройств каждого АЛУ.

Имеет место связь классификации *Хендлера* с классификацией *Фенга*: для получения максимальной степени параллелизма в смысле *Фенга* необходимо вычислить произведение указанных выше шести величин.

В классификации *Д. Скилликорна* [2] архитектуру любого компьютера предлагается рассматривать как абстрактную структуру, состоящую из четырех компонентов:

- *процессор команд* (IP – Instruction Procesor) – интерпретатор команд;
- *процессор данных* (DP – Data Procesor) – устройство обработки данных;
- *устройство памяти* (IM – Instruction Memory, DM – Data Memory);
- *переключатель* – абстрактное устройство, обеспечивающее связь между процессорами и памятью.

Рассматривается четыре типа переключателей:

- 1–1 – связывает пару функциональных устройств;
- *n–n* – реализует попарную связь каждого устройства из одного множества с соответствующим ему устройством из другого множества;
- *1–n* – соединяет одно выделенное устройство со всеми функциональными устройствами из некоторого набора;

- $n \times n$ – каждое функциональное устройство одного множества может быть связано с любым устройством из некоторого набора.

Заметим, что приведенные в настоящем разделе типы классификаций претендуют на более высокий уровень формализации количественных оценок параллелизма и поэтому могут быть полезными при проведении исследований, связанных с применением моделей вычислительных систем достаточно высокого уровня абстрактности.

2.4 Векторно-конвейерные компьютеры

Появление термина *суперкомпьютер* связано с созданием в середине шестидесятых годов фирмой CDC (Сеймуром Крэем) высокопроизводительного компьютера с новой *векторной* архитектурой. Основная идея, положенная в основу этой архитектуры, заключалась в распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. Эта идея оказалась плодотворной и нашла воплощение на разных уровнях функционирования компьютера.

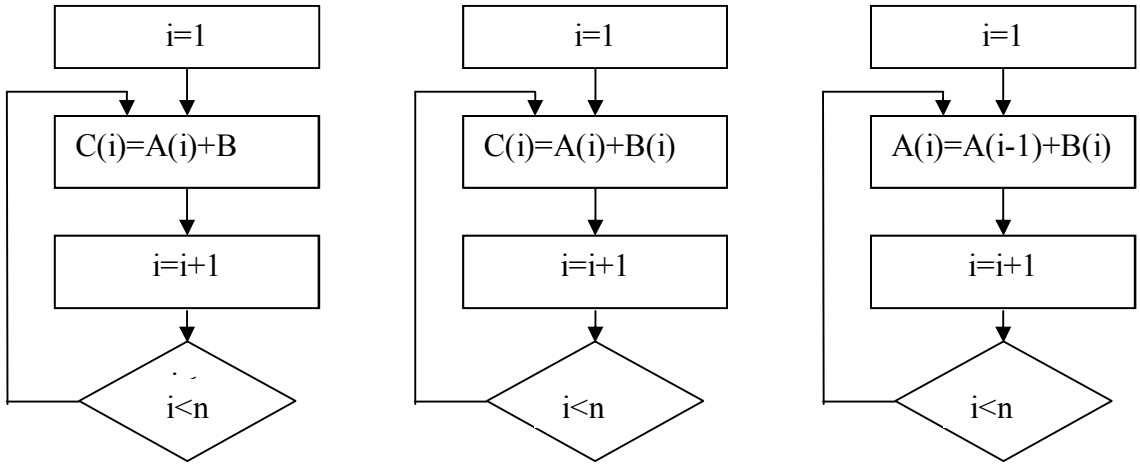
Классическим представителем мира суперкомпьютеров является первый векторно-конвейерный компьютер Cray-1 (1976). Основные особенности архитектуры этого класса компьютеров следующие.

- *Конвейеризация выполнения команд*.
- *Независимость функциональных устройств*, т.е. несколько операций могут выполняться одновременно.
- *Векторная обработка* (набор данных обрабатывается одной командой).
- *Зацепление функциональных устройств* (выполнение нескольких векторных операций в режиме «макроконвейера»).
- *Многопроцессорная обработка* (наличие независимых процессоров позволяет выполнять несколько независимых программ).

Эффективность векторно-конвейерных компьютеров существенным образом зависит от наличия одинаковых и независимых операций. В качестве при-

мера рассмотрим несколько фрагментов вычислений в виде блок-схем, показанных на рисунке 2.1, а, б, в.

Поскольку в системе команд векторно-конвейерных компьютеров обычно есть векторные команды, в которых аргументы могут быть как скалярами, так и векторами, векторизация фрагментов, показанных на рис. 2.1, а и б, не вызовет проблем. В то же время фрагмент, показанный на рис. 2.1, в, невозможно векторизовать, поскольку вычисление *i*-го элемента массива *A* не может начаться , пока не будет вычислен предыдущий элемент. В данном примере имеет место *зависимость между операциями*, которая будет препятствовать векторизации. Это надо иметь в виду при выполнении программы на компьютере векторно-конвейерной архитектуры.



а) б) в)

Рис. 2.1 Примеры векторизуемых и невекторизуемых алгоритмов

В качестве примера предположим, что половина некоторой программы – это сугубо последовательные вычисления, которые нельзя векторизовать. Тогда, даже в случае мгновенного выполнения второй половины программы за счет идеальной векторизации, ускорения работы всей программы более чем в два раза мы не получим. Подробно вопросы достижимого ускорения, определяемого законом Амдала, будут рассмотрены в лекции 4.

2.5 Вычислительные системы с распределенной памятью (мультикомпьютеры)

Как уже указывалось выше, вычислительные узлы этого класса (массивно-параллельных) компьютеров объединяются друг с другом посредством коммуникационной среды. Каждый узел имеет один или несколько процессоров и свою собственную локальную память. Распределенность памяти означает, что каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Доступ к памяти других узлов осуществляется посредством либо специально проектируемой для данной вычислительной системы, либо стандартной коммуникационной среды.

Преимущества этой архитектуры – низкое значение отношения цена/производительность и возможность практически неограниченно наращивать число процессоров. Различия компьютеров данного класса сводятся к различиям в организации коммуникационной среды. Известны архитектуры, в которых процессоры расположены в узлах прямоугольной решетки. Иногда взаимодействие идет через иерархическую систему коммутаторов, обеспечивающих возможность связи каждого узла с каждым. Используется также топология трехмерного тора, т.е. каждый узел имеет шесть непосредственных соседей вне зависимости от того, где он расположен.

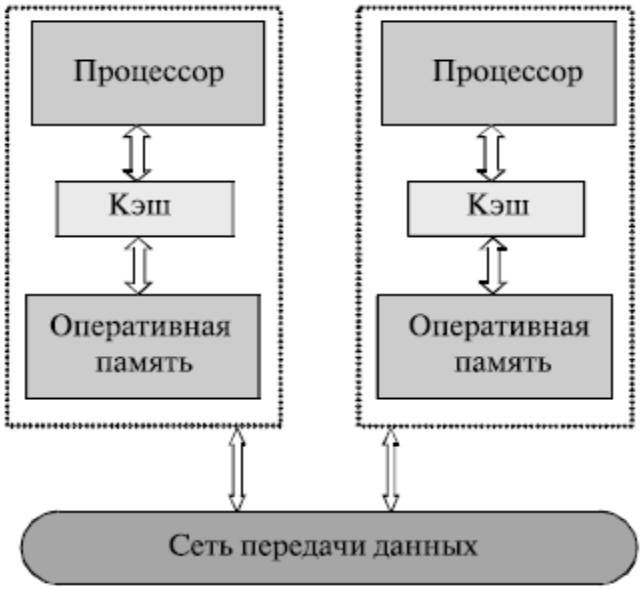


Рис. 2.2 Архитектура многопроцессорных систем с распределенной памятью

На рис. 2.2 показана общая схема связей основных элементов системы в архитектуре многопроцессорных систем с распределенной памятью.

2.6 Параллельные компьютеры с общей памятью (мультипроцессоры)

Организация параллельных вычислений для компьютеров этого класса значительно проще, чем для систем с распределенной памятью. В данном случае не надо думать о распределении массивов. Однако компьютеры этого класса имеют небольшое число процессоров и очень высокую стоимость. Поэтому обычно используются различные решения, позволяющие увеличить число процессоров, но сохранить возможность работы в рамках единого адресного пространства.

В частности общая память может быть физически распределенной, однако все процессоры имеют доступ к памяти любого процессора. Достигается это применением специальных программно-аппаратных средств. Основная проблема, которую при этом решают – обеспечение когерентности кэш-памяти отдельных процессоров. Реализация мероприятий по обеспечению когерентности кэшей позволяет значительно увеличить число параллельно работающих процессоров по сравнению с SMP-компьютером. 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 процессоров (cache-only memory architecture или COMA);
- системы, в которых обеспечивается когерентность локальных кэшей разных процессоров (cache-coherent NUMA или CC-NUMA);
- 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.

На рис. 2.3. приведены некоторые типовые схемы связей элементов в мультипроцессорных системах.

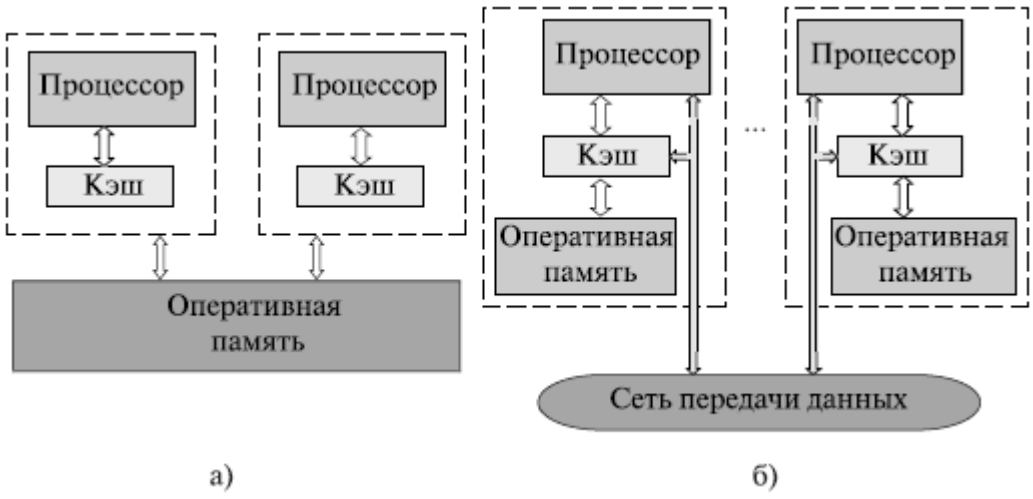


Рис. 2.3 Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным (б) доступом к памяти

Использование распределенной общей памяти (distributed shared memory или DSM) упрощает проблемы создания мультипроцессоров (известны примеры систем с несколькими тысячами процессоров). Однако при построении параллельных алгоритмов в данном случае необходимо учитывать, что время доступа к локальной и удаленной памяти может различаться на несколько порядков. Для обеспечения эффективности алгоритма в этом случае следует в явном виде планировать распределение данных и схему обмена данными между процессорами таким образом, чтобы минимизировать обращения к удаленной памяти.

В заключение обратим внимание на существенные различия векторных и массивно-параллельных архитектур. В векторной программе явно выполняются операции над всеми элементами регистра, в параллельной программе каждый из процессоров выполняет более или менее синхронно машинные команды, оперируя со своими собственными регистрами. В обоих случаях действия выполняются одновременно, однако каждый из процессоров параллельной ЭВМ может реализовывать свой алгоритм, отличающийся от алгоритмов других процессоров.

Указанное отличие является весьма существенным. Справедливо следующее утверждение: алгоритм, который можно векторизовать, можно и распарал-

лелить. Обратное утверждение не всегда верно. Например, для невекторизуемого фрагмента алгоритма, показанного на рис. 2.1, в, нетрудно организовать конвейерную схему вычислений на массивно-параллельном компьютере.

2.7 Кластеры

Кластеры являются одним из направлений развития компьютеров с массовым параллелизмом. Кластерные проекты связаны с появлением на рынке недорогих микропроцессоров и коммуникационных решений. В результате появилась реальная возможность создавать установки «суперкомпьютерного» класса из составных частей массового производства.

Один из первых кластерных проектов – Beowulf-кластеры. Первый кластер был собран в 1994 г. в центре NASA Goddard Space Flight Center (GSFC). Он включал 16 процессоров Intel 486DX4/100 МГц. На каждом узле было установлено по 16 Мбайт оперативной памяти и сетевые карты Ethernet. Чуть позже был собран кластер TheHIVE (Highly-parallrl Integrated Virtual Environment). Этот кластер включал 332 процессора и два выделенных хост-компьютера. Все узлы кластера работали под управлением Red Hat Linux.

В настоящее время известно огромное количество кластерных решений. Одно из существенных различий состоит в используемой сетевой технологии. При использовании массовых сетевых технологий, обладающих низкой стоимостью, как правило, возникают большие накладные расходы на передачу сообщений.

Для характеристики сетей в кластерных системах используют два параметра: латентность и пропускную способность. *Латентность* – это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если в параллельном алгоритме много коротких сообщений, то критической характеристикой является латентность. Если передача сообщений организована большими порциями, то более важной является пропускная способность каналов связи. Ука-

занные две характеристики могут оказывать огромное влияние на эффективность исполнения кода.

Если в компьютере не поддерживается возможность *асинхронной посылки сообщений* на фоне вычислений, то возникают неизбежные при этом накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов. Для повышения эффективности параллельной обработки на кластере необходимо добиваться *равномерной загрузки всех процессоров*. Если этого нет, то часть процессоров будет простаивать. В случае, кода вычислительная система неоднородна (гетерогенна), балансировка загрузки процессоров становится крайне трудной задачей.

В заключение еще раз зададимся вопросом: чем же все-таки кластеры отличаются от других компьютерых систем? Следуя [2] приведем следующее утверждение «Отличие понятия кластера от сети компьютеров (network of workstations) состоит в том, что для построения локальной компьютерной сети, как правило, используют более простые сети передачи данных, компьютеры сети обычно более рассредоточены, а пользователи могут применять их для выполнения каких-либо дополнительных работ». Впрочем, эта граница все чаще оказывается в значительной степени «размытой», в связи с бурным ростом пропускной способности сетей передачи данных.

2.8 Концепция GRID и метакомпьютинг

В принципе, любые вычислительные устройства можно считать параллельной вычислительной системой, если они работают одновременно и их можно использовать для решения одной задачи. Под это определение попадают и компьютеры в сети Интернет. Интернет можно рассматривать как самый мощный кластер – метакомпьютер. Процесс организации вычислений в такой вычислительной системе – *метакомпьютинг*. В отличие от традиционного компьютера метакомпьютер имеет некоторые, присущие только ему особенности:

- огромные вычислительные ресурсы (число процессоров, объем памяти и др.);
- присущая ему от природы распределенность ресурсов;
- возможность динамического изменения конфигурации (подключений);
- неоднородность;
- объединение ресурсов различных организаций, с различающейся политикой доступа (по принадлежности).

Из указанных особенностей следует, что метакомпьютер – это не только и не столько сами вычислительные устройства, сколько инфраструктура. В данном случае в комплексе должны рассматриваться модели программирования, распределение и диспетчеризация заданий, организация доступа, интерфейс с пользователем, безопасность, надежность, политика администрирования, средства и технологии распределенного хранения данных и др.

Необходимо подчеркнуть, что развитие идей метакомпьютинга связано с актуальными проблемами переработки огромных объемов информации. Характерной задачей является создание информационной инфраструктуры для поддержки экспериментов в физике высоких энергий в Европейском центре ядерных исследований (CERN). Для обработки большого объема данных целесообразно создание иерархической распределенной системы, включающей ряд связанных высокоскоростными телекоммуникационными каналами центров различного уровня. При этом центры могут быть удалены друг от друга на значительные расстояния. Основой для построения инфраструктуры указанного типа являются GRID-технологии [2].

Лекция 3

Модели вычислительных процессов и систем

3.1 Понятие графа алгоритма и его свойства

Для описания информационных зависимостей алгоритмов решения задач широко используют модель в виде ациклического ориентированного графа [1-3], называемую *графом алгоритма*. В этой модели множество операций алгоритма и существующие между операциями информационные зависимости описываются двойкой:

$$G = (V, E), \qquad (3.1)$$

где $V = \{1,...,|V|\}$ – множество вершин графа, представляющих операции алгоритма, а E – множество дуг графа, устанавливающих *частичный* порядок операций. Дуга $E_{i,j} = (i, j)$ принадлежит графу только в том случае, если операция j использует результат выполнения операции i. Свойство *ацикличности* графа алгоритма состоит в том, что никакая величина не может определяться через саму себя.

Описанная выше модель является *направленным графом*. Если дугам графа приписать веса $c_{ij}, \left(i, j = \overline{1, N}\right)$, отражающие интенсивность информационного обмена между i-й и j- ветвями программы, такой граф называется *взвешенным направленным графом*. В общем случае граф алгоритма есть мультиграф, т.е. две вершины могут быть связаны несколькими дугами [2]. При этом в качестве разных аргументов одной операции используется одна и та же величина. Количество вершин графа (не считая вершин ввода) далее будем обозначать $\left|\overline{V}\right|$. Путь максимальной длины в графе называют критическим.

Для ориентированного ациклического графа с n вершинами существует число $s<n$, для которого все вершины графа можно так пометить одним из индексов $1,2,\ldots,s$, что если дуга из вершины с индексом i идет в вершину с индексом j, то $i<j$. Покажем это [2].

Пометим любое число вершин графа, не имеющих предшествующих, единицей и удалим из графа эти вершины вместе с инцидентными им дугами. Оставшийся граф также ациклический. В нем любое число вершин, не имеющих предшествующих, пометим индексом 2 и удалим их. Продолжим этот процесс до исчерпания графа. Поскольку на каждом шаге помечается не менее одной вершины, число индексов не превысит число вершин графа.

Нетрудно заметить, что, например, для графов, показанных на рис. 3.1 и 3.2, в обоих случаях n=7, а число s при этом принимает значения 4 и 3 соответственно. Из схемы разметки, в частности, следует:

- никакие две вершины с одинаковым индексом не связаны дугой;
- минимально число индексов на единицу больше длины критического пути;
- для любого целого s , не превосходящего числа вершин, но большего длины критического пути, существует разметка, при которой используются все s индексов.

Граф, размеченный в соответствии с описанной схемой, называют *строгой параллельной формой графа* [2].

Существует строгая параллельная форма, в которой максимальная из длин путей, оканчивающихся в вершине с индексом k, равна k -1, и все входные вершины находятся в одной группе с индексом 1. Она называется *канонической*. Для заданного графа каноническая форма *единственна*.

Группа вершин с одинаковыми индексами называется *ярусом*, число вершин в группе – *шириной* яруса, а число ярусов – *высотой* параллельной формы. Параллельная форма минимальной высоты называется *максимальной*, т.к. в каждом ярусе такой формы максимальное число вершин.

Предположим теперь, что все операции алгоритма выполняются за одинаковое время, равное 1, каждая операция может начаться в момент готовности ее аргументов, а все операции, не имеющие предшествующих, могут выполняться одновременно (параллельно). Обозначим момент начала реализации алгоритма нулем, а каждой операции будем присваивать индекс, равный моменту оконча-

ния ее выполнения. Если эти индексы перенести на вершины графа алгоритма, то мы получим каноническую форму.

Ограничивая число операций, которые могут выполняться параллельно, можно получить отличающиеся строгие параллельные формы. В предельном случае, когда на каждом шаге вычислительного процесса может выполняться только одна операция, т.е. все ярусы имеют ширину, равную 1, будет получена так называемая *линейная* форма, т.е. граф упорядочивается *линейно*.

На рис. 3.1 и 3.2 приведены примеры ориентированных графов, описывающих алгоритмы нахождения суммы последовательности числовых значений

$$S = \sum_{i=1}^{n} x_i \,, \qquad (3.2)$$

где n – количество суммируемых значений. В частности, на рис. 3.1 показан ориентированный граф

$$G_S = (V_S, R_S) \qquad (3.3)$$

алгоритма последовательного суммирования элементов числового набора. Здесь $V_S = \{v_{01},...,v_{0,n},v_{11},...,v_{1,n}\}$ – множество операций (ввода – $v_{0,i}$ и суммирования – v_{1i}, $1 \le i \le n-1$), а $R_S = \{(v_{0,i}, v_{1,i}),(v_{1,i}, v_{1,i+1}),\ 1 \le i \le n-1\}$ – множество дуг, определяющих информационные зависимости операций. В данном случае операции ввода обозначены цифрами 1-4, а операции суммирования – цифрами 5-7. Нетрудно заметить, что этот граф является линейной формой и не допускает параллельную реализацию на многопроцессорной системе.

Параллельная реализация алгоритма суммирования возможна, например, в случае, когда алгоритм строится в виде каскадной схемы:

- на первой итерации каскадной схемы все исходные данные разбиваются на пары, и для каждой пары вычисляется сумма их значений;
- полученные суммы также разбиваются на пары, и снова выполняется суммирование значений пар и т.д.

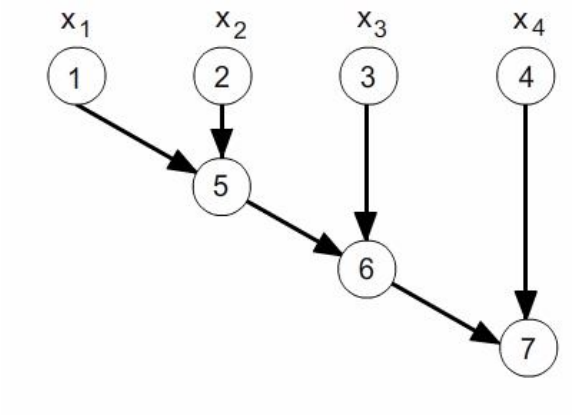


Рис. 3.1 Граф алгоритма последовательного суммирования

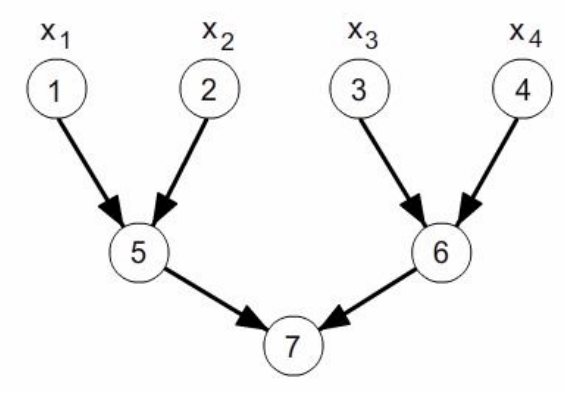


Рис. 3.2 Граф алгоритма каскадного суммирования

Соответствующий ориентированный граф каскадной схемы приведен на рис. 3.2. Он является одной из возможных строгих параллельных форм. Цифровые обозначения операций здесь те же, что и на рис. 3.1.

3.2 Проблема отображения

Приведенный выше граф каскадного суммирования отражает лишь принципиальную возможность параллельных вычислений. Эффективность реализации этих вычислений на конкретной системе зависит от того, насколько структура алгоритма соответствует архитектуре вычислительной системы. Решение этой задачи является содержанием центральной проблемы при планировании вычислительных ресурсов – *проблемы отображения* параллельного алгоритма и соответствующей ему программы на архитектуру мультипроцессорной ВС.

Постановка задачи следующая. Имеется некоторая многопроцессорная вычислительная система, на которой предполагается реализация некоторого параллельного алгоритма, требующего для себя решающее поле, т.е. подмножество процессоров. Параллельный алгоритм представляется графом алгоритма

$$G_p(V_p, E_p),$$

где V_p - множество вершин, соответствующее операциям алгоритма, E_p - множество дуг, представляющих информационные связи между ними, $N = |V_p|$ –

количество операций. Дугам графа приписываются веса $c_{ij},\left(i,j=\overline{1,N}\right)$, отражающие интенсивность информационного обмена между *i*-й и *j*- вершинами (операциями).

Вычислительная система также представляется в виде графа

$$G_s\left(P_s,E_s\right). \qquad (3.4)$$

Здесь P_s - множество процессоров, E_s - множество дуг, представляющих линии связи между процессорами. Пропускная способность линий связи характеризуется весами дуг графа $m_{ij},\left(i,j=\overline{1,N}\right),\ i,j\in E_s$.

Задается критерий оптимальности отображения $\varphi: V_p \to P_s$ графа параллельной задачи $G_p\left(A_p,E_p\right)$ на структуру вычислительной системы, заданной графом $G_s\left(P_s,E_s\right)$:

$$Q(\varphi)=Q\left\{\varphi: V_p \to P_s\right\}. \qquad (3.5)$$

Если отображение $\varphi: V_p \to P_s$ представляется матрицей $X=\left\{X_{ij}: i\in V_p, j\in P_s\right\}$ где $X_{ij}=1$, если $\varphi(i)=j$ и $X_{ij}=0$, если $\varphi(i)\neq j$, то критерий оптимальности отображения, при условии равной производительности всех процессоров системы, имеет вид

$$Q(X)=\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{p=1}^{N}\sum_{k=1}^{N} m_{ij}c_{kp}X_{ki}X_{pj} \to \min .$$

В рамках общей проблемы отображения обычно решается одна из двух задач:

1. Для данной параллельной программы выбрать решающее поле.

2. Распределить ветви параллельной программы по процессорам компьютера заданной архитектуры.

3.3 Модели сетей передачи данных между процессорами

Для точного отображения алгоритма на архитектуру конкретной вычислительной системы (ВС) граф алгоритма и граф ВС должны быть изоморфны. К

сожалению, выполнение этого требования на практике требует чрезвычайно больших усилий и может быть оправдано лишь для некоторых задач, которые решаются многократно (например, прогноз погоды) в течение длительного времени. Обычно решают задачу выбора приближенно подходящей архитектуры из некоторого множества типовых архитектур ВС.

Граф вычислительной системы определяется структурой линий коммутации между процессорами ВС (топологией сети передачи данных). К числу типовых топологий обычно относят следующие схемы коммуникации процессоров (рис. 3.3).

Полный граф (completely-connected graph или clique) – между любой парой процессоров существует линия связи. Такая топология обеспечивает минимальные затраты при передаче данных, однако оказывается сложной при большом количестве процессоров.

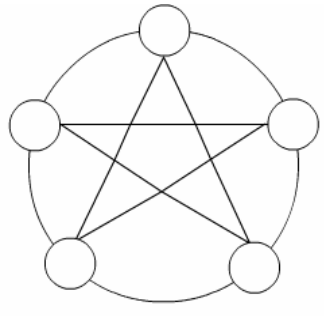
Линейка (linear array или farm) – все процессоры перенумерованы по порядку и каждый процессор, кроме первого и последнего, имеет линии связи только с двумя соседними. Достоинство – простая реализация, однако класс задач, которым эта схема соответствует, ограничен. Топология является подходящей для конвейерных вычислений.

Кольцо (ring) – получается из линейки процессоров соединением первого и последнего процессоров линейки.

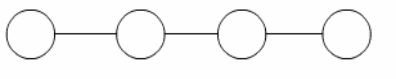
Звезда (star) – все процессоры имеют связь с одним (управляющим) процессором. Топология эффективна для централизованной схемы вычислений.

Решетка (mesh) – граф связей образует (двух- или трехмерную) сетку. Топология реализуется достаточно просто и может эффективно использоваться при решении сеточных задач, описываемых уравнениями в частных производных.

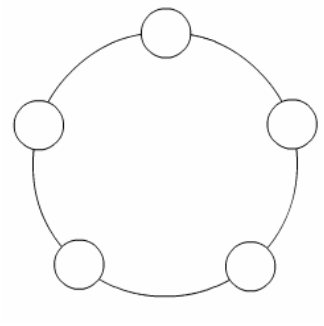
Гиперкуб (hypercube) – частный случай решетки, когда по каждой размерности сетки имеется только два процессора (при размерности N гиперкуб содержит 2^N процессоров).



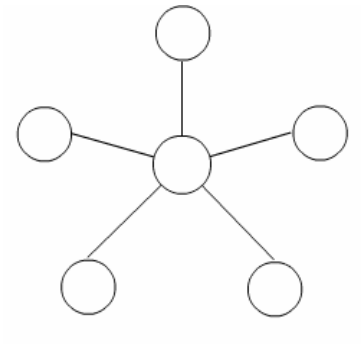
Полный граф



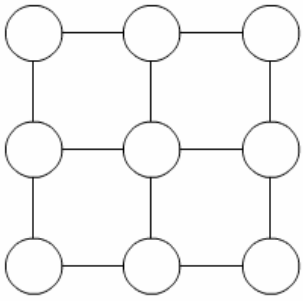
Линейка



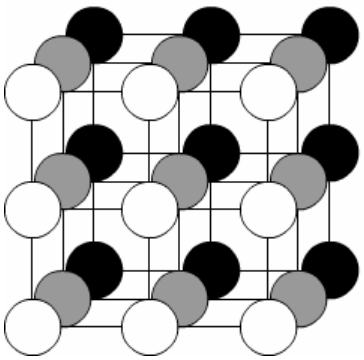
Кольцо



Звезда



2-мерная решетка



3-мерная решетка

Рис. 3.3. Примеры топологий многопроцессорных вычислительных систем

Гиперкуб обладает следующими свойствами:

- два процессора имеют соединение, если двоичные представления их номеров имеют только одну различающуюся позицию;
- в *N*-мерном гиперкубе каждый процессор связан ровно с *N* соседями;
- *N*-мерный гиперкуб может быть разделен на два (*N*–1)-мерных гиперкуба (всего возможно *N* таких разбиений);
- кратчайший путь между двумя любыми процессорами имеет длину, равную числу различающихся битовых значений в двоичных номерах процессоров (в теории информации это так называемое минимальное кодовое расстояние или расстояние Хэмминга).

Основные характеристики топологии сети:

- *диаметр* – максимальное (по всем возможным парам) расстояние между двумя процессорами сети, измеряемое по кратчайшему пути, эта величина обычно характеризует максимально время, необходимое для передачи данных между процессорами;
- *связность* (connectivity) – характеризует наличие разных маршрутов передачи данных между процессорами, показатель может быть определен, например, как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области;
- *стоимость* – общее количество линий передачи данных в многопроцессорной вычислительной системе.

3.4 Модели параллельных вычислений

Если подходящая архитектура вычислительной системы для решения заданной конкретной вычислительной задачи определена, для параллельной реализации алгоритма далее необходимо построить расписание. Для этого задается множество

$$H_s = \{(i, P_i, t_i) : i \in V\}, \qquad (3.6)$$

в котором каждой операции $i \in V$ ставится в соответствие номер используемого для ее выполнения процессора P_i и время начала выполнения операции t_i. Для того чтобы расписание было реализуемым, при задании множества H_s необходимо выполнение следующих очевидных требований:

1. Один и тот же процессор не может назначаться разным операциям в один и тот же момент: $\forall i, i \in V : t_i = t_j \Rightarrow P_i \neq P_j$.

2. К назначаемому моменту выполнения операции все необходимые данные уже должны быть вычислены: $\forall (i, j) \in R t_j \geq t_i + \tau$, где τ – время выполнения одной операции.

Граф вычислительной схемы алгоритма (3.1) совместно с расписанием (3.4) может рассматриваться как модель параллельного алгоритма:

$$A_s(G,H_s), \qquad (3.7)$$

реализуемого с использованием *s* процессоров.

Время выполнения параллельного алгоритма определяется максимальным значением времени, применяемым в расписании:

$$T_s(G,H_s)=\frac{max}{i\in V}(t_i+\tau). \qquad (3.8)$$

Для фиксированного графа вычислений целесообразно строить расписание, обеспечивающее минимальное время исполнения алгоритма:

$$T_s(G)=\frac{min}{\forall H_s}T_s(G,H_s). \qquad (3.9)$$

Часто в ходе построения параллельного вычислительного процесса с целью уменьшения времени реализации алгоритма перебираются также различные вычислительные схемы. Переход к каждой новой вычислительной схеме, как правило, ведет к изменению расписания. Такая постановка задачи, в общем виде, может быть сформулирована следующим образом:

$$T_s=\frac{min}{G}\frac{min}{H_s}T_s(G,H_s). \qquad (3.10)$$

Точное решение такой задачи возможно лишь в очень немногих простых случаях. Для большинства же реальных алгоритмов оценить хотя бы приближенно даже число возможных вариантов графа вычислительной схемы вряд ли удастся. Поэтому сформулированная задача может рассматриваться, как формальное представление целевой функции, которая может использоваться при последовательном просмотре вариантов построения параллельного вычислительного процесса на конкретной многопроцессорной вычислительной системе.

При этом в процессе перебора вариантов оценки $T_s(G,H_s)$, $T_s(G)$ и T_s могут использоваться при принятии решения о выборе конкретного графа и расписания вычислительного процесса.

Для анализа максимально возможного параллелизма в рассмотрение вводят оценку наиболее быстрого исполнения алгоритма

$$T_\infty = \frac{min}{s \geq 1} T_s . \qquad (3.11)$$

Оценку T_∞ можно рассматривать как минимально возможное время выполнения параллельного алгоритма при использовании неограниченного количества процессоров. Модель такой системы, называемой *паракомпьютером*, применяется при теоретическом анализе параллельных вычислений.

Для сравнительной оценки эффективности параллельного решения исследуемой вычислительной задачи время последовательного решения на одном процессоре – T_1 следует определять с учетом всех последовательных вариантов алгоритма:

$$T_1 = \frac{\min}{G}(G), \qquad (3.12)$$

где минимум берется по множеству всех возможных последовательных алгоритмов решения данной задачи. Для графа заданной вычислительной схемы

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

где $|\bar{V}|$ – количество вершин без вершин ввода.

3.5 Представление алгоритма в виде диаграммы расписания

Для удобства представления параллельных алгоритмов наряду с другими способами используют также временные диаграммы выполнения операторов при заданных значениях времени начала (окончания) их выполнения. В диаграммах, использовавшихся в работе [1], операторы обозначаются прямоугольниками с длиной, равной времени выполнения соответствующего оператора. Для указания связей между прямоугольниками-операторами используются стрелки, которые соответствуют дугам в графе алгоритма. Когда связей становится много, из-за множества стрелок проследить эти связи становится трудно.

Для того чтобы диаграмма оставалась наглядной при любом количестве связей и однозначно отражала расписание параллельного алгоритма введем следующие дополнительные правила. Ось ординат разобьем на интервалы, каждый из которых соответствует одному из параллельно работающих процессоров. В каждом интервале будем размещать только те прямоугольники-операторы, которые закреплены за соответствующим этому интервалу процессором.

Операторы на диаграмме будем обозначать в центре прямоугольников цифрой в кружке, а связи между операторами – цифрами слева и справа от этого кружка. В левой части прямоугольника-оператора будем записывать номера предшествующих по информационной связи операторов, а в правой части – номера операторов, следующих за данным оператором.

Поскольку каждый прямоугольник диаграммы размещается в интервале «своего» процессора, описанный способ их нумерации отражает также связь между процессорами. Поэтому этот способ представления является исчерпывающим наглядным представлением расписания параллельного алгоритма и может быть удобным программисту для написания параллельной программы. При этом информация о номерах связанных операторов может использоваться для оценки объема передаваемых данных как между процессами, так и между процессорами.

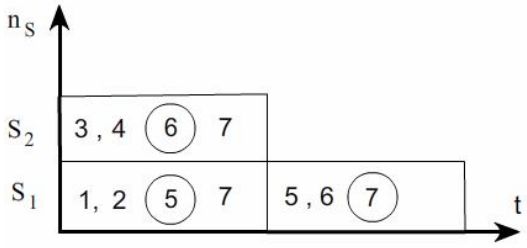


Рис. 3.4. Пример временной диаграммы алгоритма каскадного суммирования четырех чисел

На рис. 3.4 приведен пример диаграммы расписания, составленной для приведенного на рис. 3.2 параллельного алгоритма каскадного суммирования 4 чисел на двух процессорах.

3.6 Сети Петри

Часто представляет интерес задача сравнения вычислительных процессов. Для этого осуществляют их моделирование. Удобным математическим аппаратом для построения формальных моделей вычислительных процессов являются сети Петри. Различают сети типа *N* и сети типа *PN*. Ниже кратко рассматриваются сети типа *PN*, как более общий случай.

Сетью Петри *PN* называют четверку объектов

(P,T,F,M_0),

где P – непустое конечное множество мест, T – непустое конечное множество переходов: $P \cap T = \varnothing$, F – функция инцидентности, такая, что каждое место инцидентно какому-нибудь переходу, каждый переход инцидентен какому-нибудь месту и

$$P \times T \cup T \times P \to N_0,$$

где N_0 – множество натуральных чисел, а M_0 – некоторая начальная разметка.

Для каждого места p_i, $i = \overline{1,n}$ вводится разметка $m_i = M(p_i)$. Совокупность разметок всех мест $P = \{p_1, p_2, ..., p_n\}$ образует разметку сети $M = \{m_1, m_2, ..., m_n\}$. Разметку каждого места будем обозначать точками, помещенными в это место (рис. 3.5). Разметка изменяется после срабатывания перехода. Переход срабатывает в случае, когда число точек в каждом входном для данного перехода t_j месте p_i больше или равно количеству стрелок, идущих из этого места в данный переход: $M(p_i) \geq F(p_i, t_j)$.

Процесс срабатывания переходов и смены разметки называют работой сети Петри. На рис. 3.5 показана смена разметки сети Петри в случае, когда условие срабатывания выполнено, т.е. функция инцидентности $F(p_1, t_1) = 1$. На рис. 3.6 приведен пример разметки, при которой переход t_1 не может сработать, т.к. $F(p_2, t_1) = 3$, в то время как m_2=2.

а б

Рис. 3.5. Смена разметки сети Петри: *а* – до срабатывания перехода t_1, *б* – после срабатывания перехода t_1

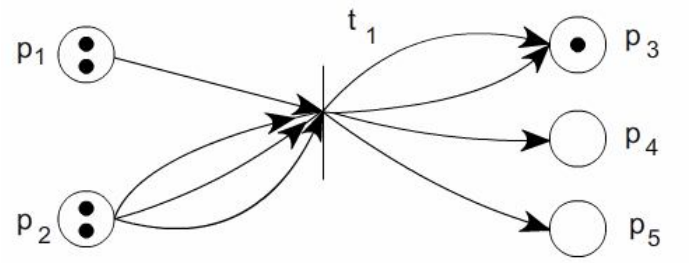


Рис. 3.6. Разметка сети Петри, при которой переход t_1 не может сработать

Говорят, что разметка M_2 непосредственно следует за разметкой M_1, если существует переход $t \in T$, срабатывание которого изменяет разметку M_1 на M_2. Процесс смены разметок записывают в виде

$$M_0[t_1 > M_1[t_2 > M_2[t_3 > M_3 ... [t_q > M_q .$$

При этом разметка M_q считается достижимой от разметки M_0. Множество достижимых разметок от разметки M_0 обозначается как $R_0 = (PN, M_0)$.

Сети Петри являются подходящим аппаратом для моделирования вычислительных процессов с целью выявления типичных ошибок, возникающих при написании параллельных программ, например, блокировка и отталкивание. Блокировка и отталкивание возникают в ситуации, когда несколько задач, получая доступ к общему ресурсу, не могут его поделить между собой. При этом вычислительный процесс затрудняется или даже полностью парализуется.

На рис. 3.7 приведен пример представления сетью Петри алгоритма, в котором два взаимодействующих процесса используют общие ресурсы (p_1; p_2). В данном случае, если срабатывают переходы t_{13} и t_{23}, то возникает ситуация блокировки. Вычисления в обоих процессах не могут начаться до тех пор, пока не сработают переходы t_{12} и t_{22}. Но они не могут сработать до тех пор, пока не сработают переходы t_{16} и t_{26}, ожидающие завершения вычислений, которые не могут начаться.

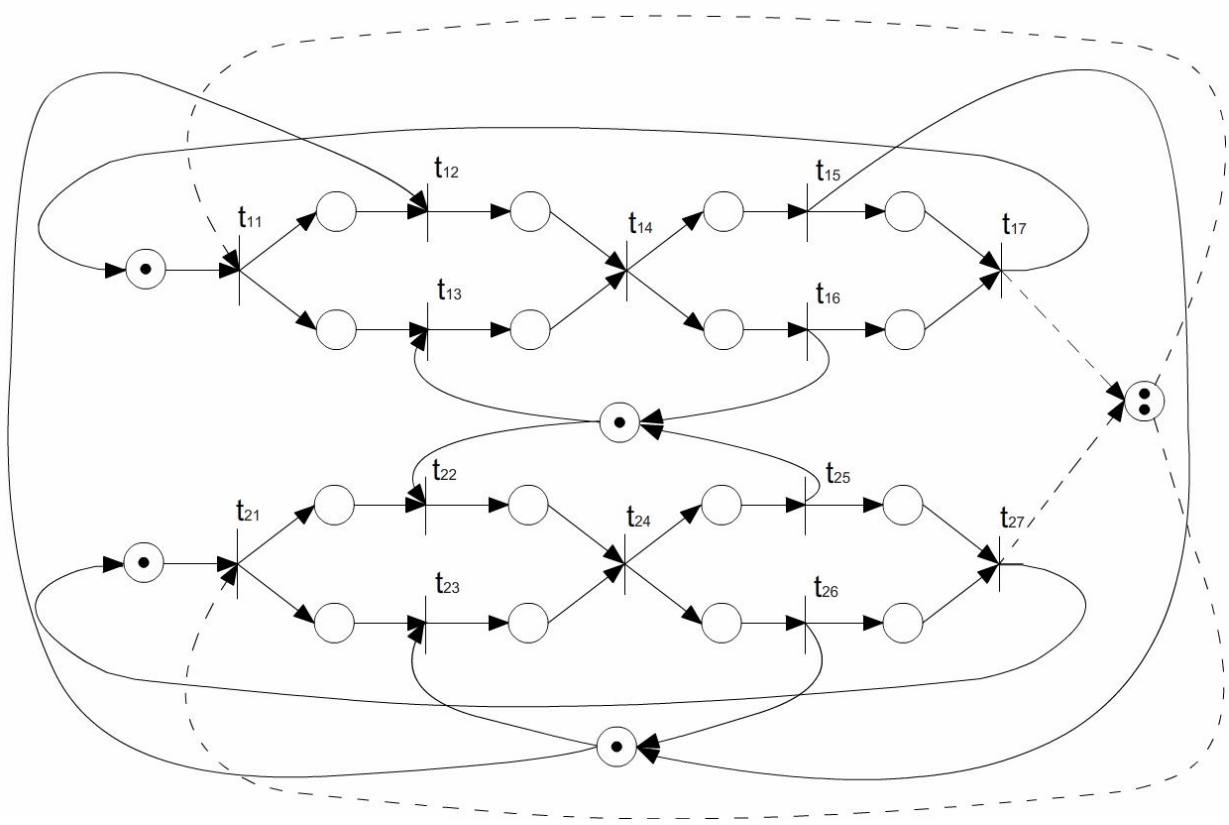


Рис. 3.7 Пример сети Петри двух взаимодействующих процессов, использующих общие ресурсы

Простейший способ устранения блокировки в данной ситуации – добавление одного места с двумя точками (добавленные связи обозначены пунктиром). Нетрудно заметить, что при этом один (любой) из указанных двух процессов может начаться только после того, как завершен другой процесс.

Лекция 4

Построение оценок производительности и эффективности параллельных компьютеров

4.1 Основные понятия и предположения

Несмотря на большое обилие различных архитектур, существует лишь *два способа параллельной обработки* данных: собственно *параллелизм* и *конвейерность*. В компьютере обычно реализуются все основные типы команд: *скалярные*, *векторные* и *конвейерные*. Команда, у которой все аргументы скалярные величины, называется скалярной командой. Если хотя бы один аргумент вектор, команда называется векторной. Соответственно в составе компьютера могут быть скалярные, векторные и конвейерные устройства. Введем необходимые определения и предположения, касающиеся оценки производительности вычислительных систем.

Предполагается, что система состоит из набора *функциональных устройств* (ФУ). Результат предыдущего срабатывания ФУ может сохраняться в нем только до момента очередного срабатывания. ФУ не может одновременно выполнять операцию и сохранять результат, т.е. не имеет *собственной памяти*. ФУ называется *простым*, если никакая последующая операция не может начаться раньше, чем предыдущая. *Конвейерное* ФУ состоит из цепочки простых ФУ, которые называют *элементарными*. Очередная операция считается выполненной после прохождения всех элементарных ФУ (ступеней конвейера).

Пусть время выполнения одной операции τ. Тогда за время T может быть выполнено приблизительно T/τ операций (здесь и во многих случаях далее для простоты мы не учитываем, что следует брать только целую часть результата деления). Время τ реализации одной операции называют *стоимостью операции*, а сумму стоимостей всех операций T *– стоимостью работы*. Минимально возможное время выполнения алгоритма определяется длиной критического пути.

Загруженностью устройства – p называют отношение стоимости реально выполненной работы к максимально возможной стоимости. Показатель эффективности одного процессора – количество операций, запускаемых за один такт процессора – IPC (instructions per sycle). Общая вычислительная мощность многопроцессорной системы оценивается *пиковой производительностью*, определяемой как максимальное количество операций, которое может быть выполнено системой за единицу времени при отсутствии потерь времени на связи между ФУ. Единица измерения производительности – Flops (одна вещественная операция в секунду).

Пиковая производительность многопроцессорной системы определяется как количество функциональных устройств, предназначенных для выполнения операций с плавающей точкой (равное числу IPC), умноженное на частоту работы процессора и на число процессоров. Например, для компьютера с двумя устройствами с плавающей точкой и частотой 500 МГц пиковая производительность равна 1000 Mflops (1 Gflops). Эффективность использования других функциональных устройств (целочисленная арифметика, обращение к памяти и др.) выявляется путем сравнения реально достижимой на тестах производительности с пиковой.

Реальная производительность – это количество операций, реально выполняемых в среднем в единицу времени. Реальная производительность обычно существенно меньше пиковой. Превышение пиковой производительности над реальной характеризует, насколько данная архитектура приспособлена к решению конкретной задачи. Отношение реальной производительности к пиковой называется *эффективностью* реализации задачи на данном конкретном компьютере.

Эффективность реализации программы повышается в том случае, когда возрастает относительная загруженность АЛУ. Для этого необходимо устранять узкие места, которые обычно связаны с временем обращения к памяти и временем пересылки данных. Если на большинстве задач эффективность работы компьютера более 0,5, ситуацию можно считать хорошей [2].

4.2 Построение соотношений для оценки производительности

Если s устройств системы имеют пиковые производительности $\pi_1,\ldots,\pi_s$ и работают с загруженностями $p_1, \ldots, p_s$, то реальная производительность системы выражается формулой [2]

$$r = p\pi, \tag{4.1}$$

где $\pi = \pi_1 + \ldots + \pi_s$, а p – *загруженность системы*, определяемая как

$$p = \sum_{i=1}^{s} \alpha_i p_i, \quad \alpha_i = \frac{\pi_i}{\sum_{j=1}^{s} \pi_j}. \tag{4.2}$$

Из (4.1) видно, что для достижения наибольшей реальной производительности системы при фиксированном числе устройств необходимо обеспечить наиболее полную ее загруженность. Дальнейшее повышение производительности достигается увеличением числа устройств.

Ускорение реализации алгоритма на вычислительной системе из s устройств определяется как [2]

$$R_s = \frac{r}{\pi_s}, \tag{4.3}$$

где π_s – пиковая производительность самого быстродействующего устройства системы. Это означает, что наибольшее ускорение – s системы из s устройств может достигаться только в случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены. *Реальное ускорение* для однородных вычислительных систем, имеющих одинаковую производительность устройств, часто определяют также как отношение времени решения задачи на одном процессоре – T_1 к времени T_s решения той же задачи на системе из s таких же процессоров:

$$R = \frac{T_1}{T_s}. \tag{4.4}$$

Это соотношение можно получить также из (4.3) с учетом (4.1), т.к. при одинаковой производительности устройств $p = T_1 / T_s \cdot s$, а $\pi = \pi_s \cdot s$.

Отношение *реального ускорения* к числу используемых процессоров s:

$$E_s = \frac{R}{s} = \frac{T_1}{T_s \cdot s} \qquad (4.5)$$

называют *эффективностью* системы. Второе равенство в (4.5) показывает, что при одинаковой производительности устройств эффективность системы совпадает со значением загруженности системы. Далее всюду, где имеет место этот случай, под загруженностью системы подразумевается эффективность и применяется обозначение, принятое в (4.5). Наилучшие показатели ускорения и эффективности – соответственно $R=s$, $E_s = 1$.

Для анализа производительности вычислительных систем, в которых имеют место направленные связи между устройствами, воспользуемся моделью в виде ориентированного *графа*, в котором вершины обозначают устройства, а дуги – связи между ними [2]. Предположим, дуга графа системы идет из i-го устройства в j-е. Поскольку результат i-го устройства является аргументом j-го, количество операций, выполняемых j-м устройством, не может более, чем на 1, отличаться от количества операций, реализованных i-м устройством:

$$N_i - 1 \le N_j \le N_i + 1. \qquad (4.6)$$

Допустим, связный граф содержит q дуг. Если k-е устройство за время T выполнило N_k операций, а l-е – N_l операций, то из (4.6) вытекает, что

$$N_l - q \le N_k \le N_l + q$$

для любых k, l, $1 \le k, l \le s$.

Перенумеруем устройства так, что $\pi_1 \le \pi_2 \le \ldots \le \pi_s$. Тогда в соответствии с последним неравенством можно записать

$$(N_1 - q)s + q \le \sum_{i=1}^{s} N_i \le (N_1 + q)s - q. \qquad (4.7)$$

Разделив все части неравенств (4.7) на T с учетом того, что

$$\frac{1}{T}\sum_{i=1}^{s} N_i = r, \quad \frac{N_1}{T} = \pi_1,$$

указанные неравенства можно переписать в виде

$$\pi_1 s - \frac{q(s-1)}{T} \le r \le \pi_1 s + \frac{q(s-1)}{T}. \qquad (4.8)$$

Слагаемые $q(s-1)/T$ в неравенствах (4.8) при увеличении T стремятся к нулю. Это означает, что для системы из s устройств с пиковыми производительностями $\pi_1,\ldots,\pi_s$, описываемой связным графом, максимальная производительность $r_{\max}$ определяется как

$$r_{\max} = s \min_{1 \le i \le s} \pi_i . \qquad (4.9)$$

4.3 Законы Амдала

Из (4.7), (4.8) вытекают важные следствия [2]:

1. Загруженность системы не превосходит

$$p_{\max} = \frac{s \min_{1 \le i \le s} \pi_i}{\sum_{i=1}^{s} \pi_i} . \qquad (4.10)$$

2. Ускорение системы не превосходит

$$R_{\max} = \frac{s \min_{1 \le i \le s} \pi_i}{\max_{1 \le i \le s} \pi_i} . \qquad (4.11)$$

3. ***1-й закон Амдала.*** Производительность вычислительной системы, состоящей из связанных между собой устройств, определяется самым непроизводительным устройством.

4. Асимптотическая производительность системы максимальна, если все устройства имеют одинаковые пиковые производительности.

Центральное значение для оценки производительности многопроцессорных вычислительных систем имеет [2].

***2-й закон Амдала*:**

Пусть система состоит из s одинаковых устройств, а n операций из общего числа операций алгоритма N могут выполняться только последовательно, тогда максимально возможное ускорение равно

$$R = \frac{s}{\beta \cdot s + (1-\beta)}, \qquad (4.12)$$

где $\beta = n/N$.

Покажем это. Если пиковые производительности всех устройств одинаковы и равны π, в соответствии с (4.1) – (4.3) ускорение определяется как

$$R = \sum_{i=1}^{s} p_i \,. \qquad (4.13)$$

Загруженность устройства, на котором выполняется последовательная часть программы, равна единице. Загруженности остальных устройств

$$p_i = \frac{(1-\beta)N/s}{\beta N + (\beta - 1)N/s}, \quad i = \overline{2,s}\,.$$

Следовательно, в соответствии с (4.13)

$$R = 1 + \sum_{i=2}^{s} \frac{(1-\beta)N/s}{\beta N + (1-\beta)N/s} = \frac{s}{\beta s + (1-\beta)}.$$

Формула Амдала используется для прогноза возможного ускорения. Например, в случае, когда половина операций не поддаются распараллеливанию, максимально достижимое ускорение в случае использования 2 процессоров в соответствии с (4.12) составит около 1,33, для 10 процессоров – менее 1,82, а для 100 процессоров – около 1,98. В данном примере наиболее «узким» местом является сам алгоритм решения задачи, а основные усилия должны быть направлены на поиск другой формулировки задачи, допускающей более высокую степень параллелизма.

4.4 Закон Густавсона – Барсиса

Оценку максимально достижимого ускорения параллельного алгоритма можно построить также исходя из имеющейся доли последовательных расчетов, задаваемой в виде [3]:

$$g = \frac{\tau_n}{\tau_n + \tau_{N-n}/s}, \qquad (4.14)$$

где τ_n, и τ_{N-n} – время, необходимое для выполнения последовательной и параллельной частей соответственно.

С учетом введенных обозначений время решения задачи на одном и s процессорах соответственно

$$T_1 = \tau_n + \tau_{N-n}, \quad T_{s1} = \tau_n + \tau_{N-n}/s\,. \tag{4.15}$$

С другой стороны, из соотношения (4.14) для величины g можно записать:

$$\tau_n = g\left(\tau_n + \tau_{N-n}/s\right), \qquad \tau_{N-n} = (1-g)s\left(\tau_n + \tau_{N-n}/s\right). \tag{4.16}$$

С учетом (4.4), (4.15) и (4.16) получаем оценку для ускорения

$$R = \frac{T_1}{T_s} = \frac{\tau_n + \tau_{N-n}}{\tau_n + \tau_{N-n}/s} = \frac{\left(g + (1-g)s\right)\left(\tau_n + \tau_{N-n}/s\right)}{\tau_n + \tau_{N-n}/s} = g + (1-g)s\,. \tag{4.17}$$

Оценку (4.17) называют законом Густавсона – Барсиса. Нетрудно заметить, что эту оценку можно также переписать в виде:

$$R = \frac{T_1}{T_s} = s + (1-s)g\,. \tag{4.18}$$

4.5 Производительность конвейерных систем

Если ФУ конвейерного типа, то операция разбивается на последовательность микроопераций. Каждую микрооперацию выделяют в отдельную часть устройства и располагают их в порядке выполнения так, чтобы входные аргументы прошли через все *ступени конвейера*. Рассмотрим возникающие при этом особенности оценки производительности устройства [2].

Предположим, что конвейерное устройство состоит из l ступеней, срабатывающих за один такт. Тогда, например, для сложения двух векторов из n элементов потребуется $l + n - 1$ тактов. Если при этом используются также векторные команды, то потребуется (возможно, несколько) дополнительных тактов σ для их инициализации. Эта величина учитывает также возможные пропуски тактов выдачи результатов на выходе конвейера, вследствие необходимости выполнения вспомогательных операций, связанных с организацией конвейера.

С использованием введенных обозначений запишем соотношение для оценки производительности конвейера:

$$E = \frac{n}{t} = \frac{n}{[(\sigma + l + n - 1)\tau]} = \frac{1}{\left[\tau + (\sigma + l - 1)\frac{\tau}{n}\right]}, \qquad (4.19)$$

где τ – время такта работы компьютера. На рис. 4.1 приведен график зависимости производительности от n, построенный по соотношению (4.19).

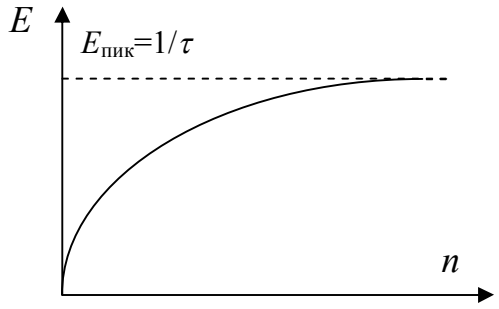


Рис. 4.1 Зависимость производительности конвейерного устройства от длины входного набора данных

Обычно вычислительные системы строятся с использованием одновременно всех типов устройств: скалярных, векторных и конвейерных. В частности, первый *векторно-конвейерный компьютер* Cray-1 (пиковая производительность 160 Mflops) имел 12 конвейерных функциональных устройств, причем все функциональные устройства могли работать одновременно и независимо друг от друга.

4.6 Масштабируемость параллельных вычислений

Параллельный алгоритм называют масштабируемым (scalable), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении эффективности использования процессоров. Для характеристики свойств масштабируемости оценивают накладные расходы (время T_0) на организациию взаимодействия процессоров, синхронизацию параллельных вычислений и т.п.:

$$T_0 = sT_s - T_1, \qquad (4.20)$$

где T_s, T_1 – те же, что и в (4.4).

Используя введенные обозначения, соотношения для времени параллельного решения задачи и соответствующего ускорения можно представить в виде

$$T_s = \frac{T_1 + T_0}{s}, \qquad (4.21)$$

$$R_s = \frac{T_1}{T_s} = \frac{sT_1}{T_1 + T_0}. \qquad (4.22)$$

Соответственно эффективность использования s процессоров

$$p_s = \frac{R_s}{s} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0/T_1}. \qquad (4.23)$$

Из (4.23) следует, что если время решения последовательной задачи фиксировано (T_1=const), то при росте числа процессоров эффективность может убывать лишь за счет роста накладных расходов T_0.

Если число процессоров фиксировано, эффективность их использования, как правило, растет при повышении времени (сложности) решаемой задачи T_1. Связано это с тем, что при росте сложности задачи накладные расходы T_0 обычно растут медленнее, чем объем вычислений T_1. Для характеристики свойства сохранения эффективности при увеличении числа процессоров и повышении сложности решаемых задач строят так называемую функцию изоэффективности. Рассмотрим схему ее построения.

Пусть задан желаемый уровень эффективности выполняемых вычислений:

p_s=const.

Из выражения для эффективности (4.23) можно записать

$$\frac{T_0}{T_1} = \frac{1 - p_s}{p_s}.$$

Или

$$T_1 = KT_0, \qquad \text{где} \qquad K = \frac{p_s}{1 - p_s}.$$

Из последнего равенства видно, что эффективность характеризуется коэффициентом K. Следовательно, если построить функцию вида

$$N = F(K, s),$$

то для заданного фиксированного уровня эффективности K каждому числу процессоров s можно поставить в соответствие требуемый уровень сложности –

N и наоборот. При рассмотрении конкретных вычислительных алгоритмов построение функции изоэффективности позволяет выявить пути совершенствования параллельных алгоритмов.

Для построения этих функций удобно использовать закон Густавсона – Барсиса. Эффективность использования s процессоров в соответствии с этим законом выражается в виде

$$E_s = \frac{R}{s} = 1 + \frac{(1-s)}{s} g .$$

При заданном фиксированном p_s=const с использованием этого равенства можно построить аналитическое соотношение для функции изоэффективности в следующем виде:

$$g = F(E_s, s).$$

Такая форма может оказаться более удобной в случае, когда известна доля времени на проведение последовательных расчетов в выполняемых параллельных вычислениях.

4.7 Верхняя граница времени выполнения параллельного алгоритма

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

$$T_s < T_\infty + T_1/s. \qquad (4.24)$$

Действительно, пусть H_∞ есть расписание для достижения минимально возможного времени выполнения T_∞. Для каждой итерации τ, $0<\tau<T_\infty$ выполнения расписания H_∞ обозначим через n_τ количество операций, выполняемых в ходе итерации τ. Расписание выполнения алгоритма с использованием s 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_∞. Эти операции могут быть выполнены не более чем за $]n_\tau/s[$ итераций при использовании s процессоров. Как результат, время выполнения алгоритма T_s может быть оценено следующим образом:

$$T_s = \sum_{T=T_1}^{T_\infty} \left\lceil \frac{n_\tau}{s} \right\rceil < \sum_{T=T_1}^{T_\infty} \left\lceil \frac{n_\tau}{s} \right\rceil + 1 = \frac{T_1}{s} + T_\infty , \qquad (4.25)$$

где ⌉*⌈ – означает операцию округления до целого числа в сторону увеличения.

Приведенная схема рассуждений, по существу, дает практический способ построения расписания параллельного алгоритма. Первоначально может быть построено расписание без учета ограниченности числа используемых процессоров (расписание для *паракомпьютера*). Затем, в соответствии с описанной выше схемой, может быть построено расписание для конкретного количества процессоров.

4.8 Факторы, влияющие на производительность, и способы ее повышения

Для того чтобы правильно интерпретировать достигнутые показатели ускорения и эффективности при решении конкретной задачи на параллельном компьютере, надо ясно представлять все факторы, которые влияют на производительность. Известно, что на компьютере с огромной пиковой производительностью можно не получить ускорения или даже получить замедление счета по сравнению с обычным персональным компьютером. Перечислим факторы, которые влияют на производительность.

1. Архитектура процессоров. Например, если решается задача, в которой отсутствуют массивы данных, элементы которых могут обрабатываться одновременно, а каждая следующая операция может выполняться лишь после завершения предыдущей, тогда применение мощного векторного суперкомпьютера ничего не даст.

2. Память и системная шина, соединяющая микропроцессоры с памятью. Пропускная способность системной шины оказывает большое влияние на показатели ускорения и эффективности, особенно если в задаче много обменов данными между процессорами.

3. Кэш-память. Большое значение имеет ее объем, частота работы, организация отображения основной памяти в кэш-память. Эффективность кэш-памяти

зависит от типа задачи, в частности, от рабочего множества адресов и типа обращений, которые связаны с *локальностью вычислений* и *локальностью использования данных*. Наиболее характерным примером конструкции, обладающей свойством локальности, является цикл. В циклах на каждой итерации выполняются одни и те же команды над данными, которые обычно получены на предшествующей операции. Существенное ускорение выполнения циклов достигается путем его размещения его данных в кэш-памяти. Если объема кэш-памяти не хватает, задействуется следующий уровень иерархии памяти, и т.д. Именно кэш-память чаще всего оказывает наиболее существенное влияние на характеристики программ вообще, и распараллеливаемой задачи в частности.

4. Коммутационные сети. Они определяют накладные расходы – время задержки передачи сообщения. Оно зависит от латентности (начальной задержки при посылке сообщений) и длины передаваемого сообщения. На практике о величине латентности судят по времени передачи пакета нулевой длины.

5. Программное обеспечение. Операционная система, драйвера сетевых устройств, программы, обеспечивающие сетевой интерфейс нижнего уровня, библиотека передачи сообщений (MPI), компиляторы оказывают огромное влияние на производительность параллельного компьютера. В настоящем курсе лекций эти вопросы не затрагиваются. Достаточно подробное рассмотрение этих вопросов можно найти в учебных пособиях, посвященных параллельному программированию [3, 7].

Повышение производительности обычно достигается за счет увеличения параллельно работающих процессоров. При этом основная проблема – организация связи между процессорами. Конечно, самый простой способ коммутации процессоров – использование общей шины. Однако в таких системах даже небольшое увеличение числа процессоров, подключаемых к общей шине, делает ее узким местом.

Применяются различные способы преодоления этой проблемы, основанные на использовании различных схем *коммутации.* Если число процессоров и модулей памяти, для связи между которыми используются коммутаторы, вели-

ко, в схеме также возможны большие задержки. Уменьшение задержек достигается путем подбора наиболее подходящей топологии сети, обеспечивающей уменьшение средней длины пути между двумя узлами системы. Среднюю длину пути можно уменьшить, применяя вместо простой линейки схему в виде кольца или гиперкуба.

Как уже указывалось, *Интернет* можно рассматривать как самый большой компьютер с распределенной памятью. В рамках этой архитектуры сокращение расходов на взаимодействие (обмен данными) параллельно работающих процессоров является наиболее острой проблемой. Подходящими для реализации в распределенной сети компьютеров являются задачи, которые могут быть представлены в виде фрагментов, не требующих частых обменов данными.

Отдельное, стоящее несколько особняком, направление повышения производительности компьютеров – применение *специализированных процессоров*. В этом случае высокая производительность достигается за счет использования особенностей конкретных алгоритмов. При этом компьютер теряет в гибкости и универсальности. Широко используемыми, например, являются спецпроцессоры для аппаратной поддержки реализации быстрого преобразования Фурье. Этот путь оправдан, когда суперкомпьютер создается для решения специального класса часто решаемых задач, например, для подготовки ежедневного прогноза погоды.

Лекция 5

Построение параллельных алгоритмов: инженерный подход

5.1 Постановка задачи

Известно, что перенос последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи. Для того чтобы построить эффективный параллельный алгоритм, строго говоря, следует провести анализ графа алгоритма и решить задачу отображения так, как это сформулировано в разделе 3.2. Решение такой задачи оптимизации на графах требует значительных усилий и высокой квалификации.

На практике разработку параллельного алгоритма обычно осуществляет специалист, работающий в некоторой предметной области, не всегда владеющий методами дискретной оптимизации. С другой стороны, строгое решение этой задачи требуется далеко не всегда. Обычно ограничения, связанные с типовым набором доступных архитектур, все равно вынуждают исследователя находить некоторое приемлемое для него решение, руководствуясь не вполне строгими, но проверенными на практике приемами и правилами.

В частности, если в конкретной задаче элементы некоторого массива исходных данных могут обрабатываться независимо друг от друга, то эти правила обычно очевидны и позволяют строить весьма эффективные параллельные алгоритмы. В этом случае задача переработки может свестись к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы в этом случае зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы) [2,3]. По мере увеличения числа (а значит уменьшения размеров) фрагментов данных, объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Иногда используется следующий простой способ построения эффективной параллельной программы, совмещенный с этапом ее отладки. Размеры фрагментов массива исходных данных уменьшают (соответственно увеличивают число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров. Этот подход обсуждался в работе [7].

Совокупность методов и приемов распараллеливания, не требующих строгого решения задачи отображения графа алгоритма на граф вычислительной системы, будем называть *инженерным* подходом. В настоящем разделе в рамках этого подхода рассматриваются некоторые правила и приемы построения параллельных алгоритмов, выработанные на основе опыта и здравого смысла. Успешность применения этих методов в значительной степени будет зависеть от соответствия структуры построенного параллельного алгоритма типу его внутреннего параллелизма. Поэтому начнем с рассмотрения классификации алгоритмов по этому признаку.

5.2 Классификация алгоритмов по типу параллелизма

Способность алгоритма к распараллеливанию потенциально связана с одним из двух (или одновременно с обоими) внутренних свойств, которые характеризуются как *параллелизм задач* (message passing) и *параллелизм данных*

(data parallel). Если алгоритм основан на параллелизме задач, вычислительная задача разбивается на несколько, относительно самостоятельных подзадач, каждая из которых загружается в "свой" процессор. Каждая подзадача реализуется независимо, но использует общие данные и/или обменивается результатами своей работы с другими подзадачами. Для реализации такого алгоритма на многопроцессорной системе необходимо выявлять независимые подзадачи, которые могут выполняться параллельно. Часто это оказывается далеко не очевидной и весьма трудной задачей. Методика решения этой задачи будет рассмотрена в следующем разделе.

При наличии в алгоритме свойства параллелизма данных, одна операция может выполняться сразу над всеми элементами массива данных. В этом случае различные фрагменты массива могут обрабатываться независимо на разных процессорах. Для алгоритмов этого типа распределение данных между процессорами обычно осуществляется до выполнения задачи на ЭВМ. Построение алгоритма, обладающего свойством параллелизма данных, и подбор подходящей архитектуры компьютера для него могут выполняться с использованием достаточно простых методик, не требующих применения сложного математического аппарата.

Для того чтобы в полной мере использовать структурные свойства алгоритма, необходимо прежде всего выявить, к какому типу он относится. Ниже приводится общая классификация алгоритмов, с точки зрения типа параллелизма, заимствованная из работы [10].

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

2. Алгоритмы с распределением данных (Data Partitioning). Это разновидность параллелизма данных, при котором пространство данных может быть

разделено на непересекающиеся области, с каждой из которых связаны независимые процессы, оперирующие каждый со своими данными. Требуется лишь редкий обмен между этими процессами.

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

4. Алгоритмы с синхронизацией итераций (Synchronous Iteration). Многие из стандартных численных итерационных параллельных алгоритмов требуют синхронизации в конце каждой итерации, заключающейся в том, что разрешение на начало следующей итерации дается после того, как все процессоры завершили предыдущую итерацию.

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

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

Нетрудно заметить, что некоторые алгоритмы из приведенного списка обладают явно выраженными свойствами параллелизма задач или параллелизма по данным. Вместе с тем ряд алгоритмов в той или иной мере обладают *обоими* указанными свойствами. Это следует учитывать при выборе способа и схемы декомпозиции задачи на подзадачи. Далее рассматривается основанная на разумных предположениях схема этапов построения параллельного алгоритма, основанного на декомпозиции данных.

5.3. Общая схема этапов разработки параллельных алгоритмов

В учебном пособии [3] описана технология подготовки параллельных приложений в виде следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.
2. Определение для сформированного набора подзадач информационных взаимодействий.
3. Масштабирование подзадач, определение количества процессоров.
4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

После выполнения указанных этапов и оценки качества параллельного алгоритма (ускорения, эффективности, масштабируемости) может оказаться необходимым повторение некоторых (или всех) этапов [3]. Если в результате ряда попыток желаемые показатели качества не достигаются, следует проанализировать и, возможно, изменить математическую постановку задачи с целью построения новой вычислительной схемы.

Следует заметить, что указанная последовательность этапов носит условный характер. Часто, приступая к разработке параллельного алгоритма, пользователь ориентируется на конкретную вычислительную систему, в частности, может быть известно возможное число доступных процессоров. Ясно, что на этапе декомпозиции по данным следует использовать эту информацию для выбора числа областей, определяющих число подзадач.

Если точное число процессоров неизвестно, но заданы границы доступного решающего поля, можно начать с масштабирования базового набора задач, а затем выполнить декомпозицию и выявление связей по информации. Другими словами, в приведенной общей схеме необходимым является лишь содержание этапов, в то время как сами этапы могут выполняться в любой последовательности, притом любой из них может оказаться как начальным, так и завершающим. На рис. 5.1 показана возможная схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений.

Если базовые подзадачи определены, установление информационных зависимостей между ними обычно не вызывает больших затруднений. При проведении анализа информационных зависимостей между подзадачами следует различать:

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

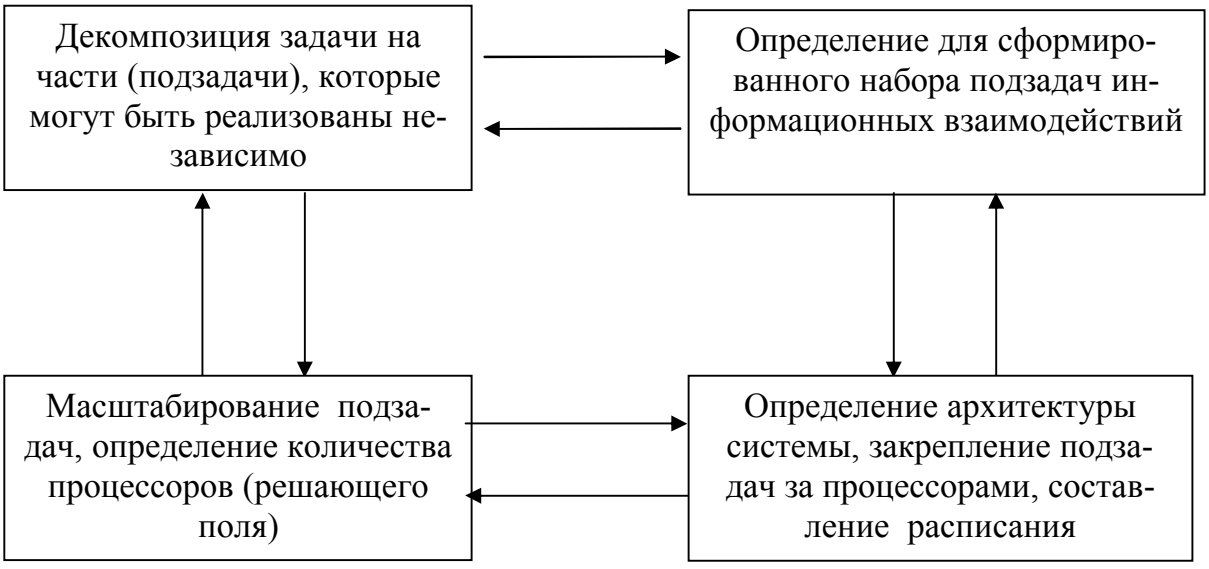


Рис. 5.1 Общая схема взаимосвязи этапов разработки параллельных алгоритмов

Если количество подзадач (областей данных) отличается от числа процессоров, то необходимо выполнить *масштабирование* параллельного алгоритма. Для сокращения количества подзадач укрупняют области исходных данных, притом в первую очередь объединяют области, для которых соответствующие

подзадачи обладают высокой степенью информационной взаимозависимости. Если число подзадач меньше числа доступных процессоров, выполняют декомпозицию. Масштабирование облегчается, если правила агрегации и декомпозиции параметрически зависят от числа процессоров.

Распределение подзадач между процессорами очевидно, если количество областей данных совпадает с числом имеющихся процессоров, а топология сети передачи данных – полный граф (все процессоры связаны между собой). Если это не так, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных. Требование минимизации информационных обменов между процессорами может вступить в противоречие с условием равномерной загрузки. Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений изменяется в ходе решения задачи. При этом необходимо перераспределение базовых подзадач между процессорами (динамическая балансировка) в ходе выполнения программы.

Центральной проблемой, как уже неоднократно указывалось выше, является выделение базовых подзадач на этапе декомпозиции. Эта проблема имеет много аспектов, в следующем разделе кратко рассматриваются лишь некоторые важнейшие.

Описанная выше схема этапов может использоваться также и для построения параллельного алгоритма, который характеризуется параллелизмом задач. При этом содержание этапов может существенно отличаться. В частности, центральной проблемой в этом случае является выявление взаимно независимых операторов, которые могут выполняться параллельно и независимо. Эти вопросы будут обсуждаться в следующей лекции.

5.4 Декомпозиция в задачах с параллелизмом по данным

Способ разделения вычислений на независимые части зависит от того, насколько полно решаемая задача обладает свойством декомпозируемости по данным, определяемого местом алгоритма в классификации, приведенной в

разделе 5.1. Если задача допускает реализацию в классе алгоритмов *с распределением данных (Data Partitioning)*, распараллеливание на подзадачи существенно облегчается. В данном случае одна операция или совокупность операций выполняются над всеми элементами массива данных, а задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. При этом обычно предъявляются требования обеспечить:

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

Рассмотрим выполнение этих требований при различных условиях.

Простейший и наиболее благоприятный с точки зрения организации параллельных вычислений случай, когда вся область исходных данных задачи может быть разделена на непересекающиеся области любых размеров, а вычисления в каждой области могут вестись независимо. Ясно, что в этом случае задача декомпозиции чрезвычайно проста: необходимо всю область разбить на подобласти, число которых равно числу доступных процессоров, а размеры подобластей подобрать так, чтобы обеспечить их равномерную загруженность, с учетом производительности каждого.

С точки зрения организации вычислений обычно более удобной является декомпозиция на области, с границами в виде прямых линий и плоскостей. На рис. 5.2 приведены примеры наиболее широко используемых регулярных структур базовых подзадач, при декомпозиции по данным.

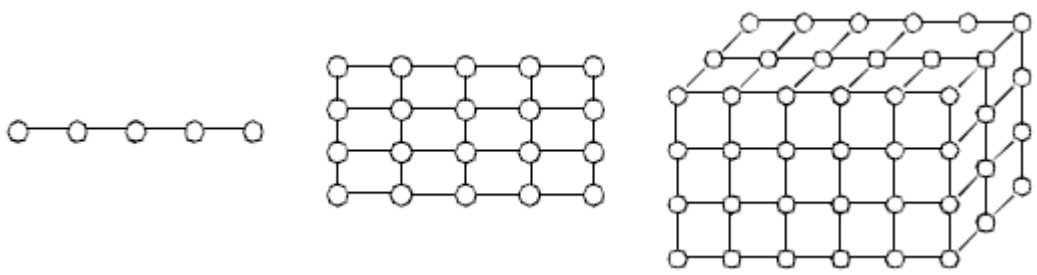


Рис. 5.2 Регулярные одно-, двух- и трехмерные структуры базовых подзадач после декомпозиции данных

Для большинства практических задач при декомпозиции по данным вычисления в каждой области не могут быть полностью независимыми. В частности, после каждой итерации (проведения вычислений во всех точках области) возникает потребность обмена результатами вычислений на границах соседних областей. Это, например, характерно для большинства сеточных методов, в которых для вычисления значения функции в некотором узле используются ее значения в нескольких соседних узлах. В этом случае выполнение указанных выше требований: сбалансированность загрузки процессоров и минимизация информационных обменов, зависит не только от размеров подобластей, но также и от их формы.

Например, в случае двумерной задачи наиболее часто используется один из двух типов декомпозиции: область может быть разделена на отдельные строки (или последовательные группы строк) – так называемая ленточная схема разделения данных, либо на прямоугольные наборы элементов – блочная схема разделения данных (см. рис. 5.3). Возникает естественный вопрос: какая из этих схем декомпозиции «лучше»?

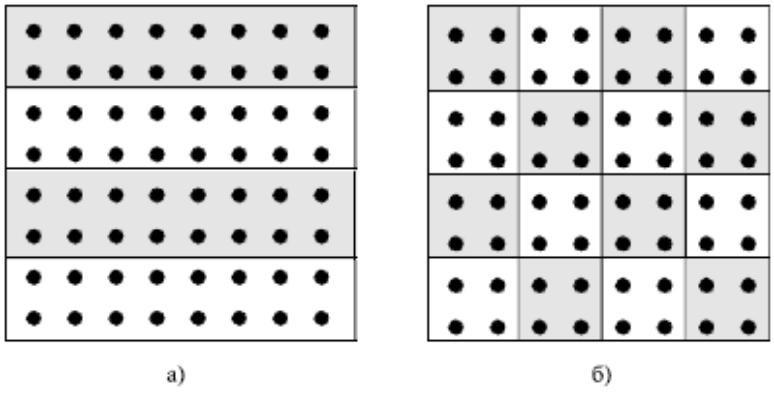


Рис. 5.3 Разделение данных на области:
а – ленточная схема, б – блочная схема

Выбор одной из указанных схем декомпозиции диктуется требованием минимизации пересылок данных между процессорами. Рассмотрим эту задачу для двумерного случая. Будем полагать, что области заданы в виде прямоугольников или квадратов, ширина полос данных в окрестности границ, которыми должны обмениваться процессоры, не зависит от направления границ фрагментов, а объем передаваемых данных определяется длиной сопряженных границ фрагментов. Приведем простой пример декомпозиции двумерной области,

имеющей размеры $H \times L$, $H \geq L$. При декомпозиции области данных на четыре подобласти (процессора) общий объем передаваемых данных при ленточном разделении данных (вдоль стороны L) пропорционален $3L$, а при блочном (на равные прямоугольники) – $H + L$.

Объем максимального межпроцессорного обмена данными между парами процессоров, обрабатывающих соседние области, составит соответственно L и $H/2$. Нетрудно заметить одинаковый общий и максимальный межпроцессорный обмены имеют место при $H=2L$. Если $H<2L$, выгоднее блочная декомпозиция, при $H>2L$ – ленточная. Ясно, что при другом числе процессоров (подобластей) результаты могут оказаться иными.

Если оказалось, что выгоднее блочная декомпозиция, то следующий важный вопрос – выбор размеров блоков. С точки зрения минимизации отношения длины граничных областей к их площади (пропорционального отношению объема межпроцессорного обмена к объему вычислений в данной подобласти) представляется, что форму подобластей следует взять в виде квадратов или прямоугольников близким к квадратам. Однако при этом возникает еще одна проблема.

При разбиении исходной области обработки данных на квадраты одинаковых размеров для фрагментов, расположенных на границах декомпозируемой области, длина границ, сопряженных с соседними фрагментами, а, следовательно и объем передаваемых данных, будет меньше. Указанное различие во времени передачи данных может оказывать существенное влияние на эффективность использования процессоров, если скорость передачи данных низкая. Неэффективность использования процессоров более заметна, когда число областей, на которые разбивается изображение, невелико.

Повышение эффективности использования процессоров может быть достигнуто увеличением размеров областей, находящихся на границах и в углах изображения. В следующем разделе этот вопрос будет детально рассмотрен для случая блочной декомпозиции.

5.5 Блочная декомпозиция с учетом локализации подобластей

Известно, что весьма широкий класс задач реализуется в классе алгоритмов *с распределением данных (Data Partitioning)*, в которых пространство данных может быть разделено на непересекающиеся области, а вычисления могут осуществляться независимо и требуется лишь редкий обмен между этими процессами. В частности, при моделировании на основе метода конечных элементов, секционной свертке изображений и др. после каждой итерации осуществляется обмен данными, полученными на границах соседних областей. Ясно, что в случае, когда размеры областей, на которые разбивается вся область значений, одинаковы, объемы пересылаемых данных будут различаться в зависимости от места расположения области.

Решим задачу такого разбиения исходной области на квадратные блоки с учетом локализации подобластей, при котором время работы всех процессоров с учетом пересылок максимально сбалансировано. Для простоты рассмотрим случай, когда исходная область квадратная: $X \times X$, где X – число отсчетов одной стороны области. Будем полагать, что для заданных: вычислительного алгоритма и вычислительной системы, известны константы: τ_p – время расчета при обработке одного отсчета (точки) области и τ_n – среднее время, затрачиваемое на передачу информации, необходимой для одной точки области.

Обычно, с точки зрения удобства организации вычислений, всю область данных разбивают на прямоугольные фрагменты. Пусть x - сторона фрагмента, численно равная числу точек. Потребуем, чтобы величина x удовлетворяла неравенству

$$\Delta_{доп} \le \left(4 \cdot x \cdot \tau_n\right) \Big/ \left(x^2 \tau_p\right) = \le \left(4 \cdot \delta\right) / x, \qquad (5.1)$$

где $\delta = \tau_n / \tau_p$ – отношение отрезков времени, необходимых для пересылки данных к времени обработки в расчете на одну точку области, а $\Delta_{доп}$ - допустимая величина отношения времени пересылок к времени обработки внутренней области, задаваемая из условия эффективной загрузки процессоров.

Ясно, что неравенство (5.1) выполняется также для областей, расположенных на границах исходной области, т.к. они имеют меньшую длину сопряженных границ. Области, находящиеся в углах изображения размером $x_0 \times x_0$, будем называть угловыми, области $x_0 \times x$ ($x \times x_0$) – граничными, а области $x \times x$ – внутренними. Задача заключается в том, чтобы найти x_0 и x такие, чтобы время обработки всех областей с учетом затрат на пересылку было одинаковым (рис. 5.4).

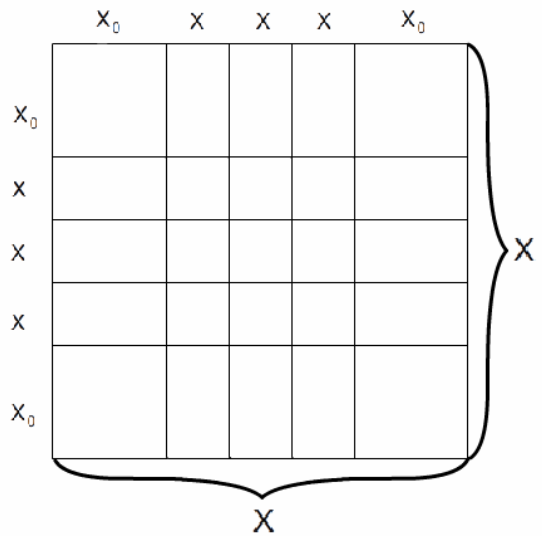


Рис. 5.4 Разбиение квадратного изображения на фрагменты

Для простоты полагаем, что ширина полос данных на границах областей, которыми они должны обмениваться, равна одному отсчету, поэтому объем передаваемых данных пропорционален длине границ. Тогда суммарное время обработки с учетом пересылок:

а) для внутренней области:

$$T_{вн} = x^2 \cdot \tau_р + 4 \cdot x \cdot \tau_п, \qquad (5.2)$$

б) для граничной:

$$T_{гр} = x \cdot x_0 \cdot \tau_р + 2 \cdot x_0 \cdot \tau_п + x \cdot \tau_п, \qquad (5.3)$$

в) для угловой:

$$T_{угл} = x_0^2 \cdot \tau_р + 2 \cdot x_0 \cdot \tau_п. \qquad (5.4)$$

Положим

$$x_0 = k \cdot x, \qquad (5.5)$$

где $1 < k < 1{,}5$ – коэффициент увеличения угловой (и соответственно граничной) области, который необходимо выбрать из условия балансировки процессоров.

При балансировке процессоров, обрабатывающих внутренние и граничные области, вычислительные затраты на обработку угловых областей существенно возрастают. Поэтому потребуем, чтобы выполнялось равенство

$$T_{угл} = T_{вн}.$$

или

$$k^2 \cdot x + 2 \cdot k \cdot \delta = x + 4 \cdot \delta . \qquad (5.6)$$

Отбрасывая из решений (5.7) отрицательные значения *k,* получаем

$$k = \frac{\delta}{x} + \sqrt{\frac{\delta}{x} + 1 + \frac{4 \cdot \delta}{x}} . \qquad (5.7)$$

С учетом неравенства (5.1) в соответствии с (5.7) можно записать условие для допустимых значений *k*:

$$k \le -\frac{\Delta_{доп}}{4} + \sqrt{\left(\frac{\Delta_{доп}}{4}\right)^2 + 1 + 4 \cdot \Delta_{доп}} . \qquad (5.8)$$

Остается подобрать удовлетворяющее условию (5.8) наибольшее значение *k*, при котором целое число *n* (число полос, на которые разбивается область решений) удовлетворяет равенству

$$(n-2)x + 2kx = X . \qquad (5.9)$$

Заметим, что обычно отношение *δ/x* невелико, при этом для значений *k*, удовлетворяющих (5.8), время обработки граничных областей не превышает времени обработки угловых и внутренних областей.

Соотношения (5.8), (5.9) могут использоваться для выбора начального разбиения исходной области на фрагменты. В действительности эффективность загрузки процессоров будет зависеть от многих других факторов, которые не учитывались в нашей упрощенной модели (например, латентность при передаче данных, а также тот факт, что исходная область может быть не квадратной, а *X* не обязано делиться без остатка на величину x, и др.).

Для более полного учета влияния всех факторов, которые не принимались во внимание в указанной упрощенной постановке, может использоваться технология итерационного планирования распределения ресурсов, описанная в работе [9]. В данном случае ее применение не вызовет дополнительных усложнений по сравнению с описанным в указанной работе вариантом, поскольку задача выбора k однопараметрическая.

5.6 Общие рекомендации по разработке параллельных программ

Ранее мы уже подчеркивали, что при переносе последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи, а именно: допускает ли задача распараллеливание по данным (*параллелизм данных*) или имеет место лишь *параллелизм задач* [7]. Как уже отмечалось выше, переработка последовательной программы существенно облегчается, если задача допускает распараллеливание по данным. В этом случае задача переработки может свестись к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы будет зависеть от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы) [7]. По мере увеличения числа (а значит уменьшения размеров) фрагментов данных объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Можно рекомендовать следующий простой способ построения эффективной программы, основанной на свойстве параллелизма данных. Размеры фрагментов массива исходных данных следует уменьшать (соответственно увеличивать число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров.

Если задача не допускает распараллеливания по данным, т.е. возможен лишь *параллелизм задач*, трудности существенно возрастают. Подход к программированию, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Для каждой подзадачи пишется своя собственная программа. Чем больше подзадач, тем большее число процессоров можно использовать и тем большего ускорения можно ожидать (если удастся обеспечить равномерную загрузку процессоров и минимизировать обмен данными между ними).

Для построения эффективного кода в данном случае программист должен провести анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсопотребляющих частей. Для этого могут использоваться различные, описанные в разделе 4, формальные модели (пространственно-временные диаграммы, модели в виде ориентированных графов, сетей Петри и др.).

Лекция 6

Выявление параллелизма алгоритмов на основе анализа графов

6.1 Постановка задачи распараллеливания

Рассматриваемые в настоящем и нескольких последующих разделах методы и алгоритмы актуальны в случае, когда распараллеливаемый алгоритм не декомпозируется по данным. Если алгоритм характеризуется лишь параллелизмом задач, выявление взаимно независимых операторов, которые могут выполняться параллельно и независимо, оказывается не столь очевидным, как это имеет место для задач, декомпозируемых по данным.

Параллелизм задач может проявляться различным образом. В частности, если алгоритм реализуется путем выполнения разных операций над одним и тем же набором данных (обработка последовательности запросов к информационным базам данных, вычисления с одновременным применением разных алгоритмов расчета и т.п.), говорят о существовании функционального параллелизма. Функциональная декомпозиция используется, например, для организации конвейерной обработки данных.

Общий подход к построению параллельных алгоритмов заключается в их представлении в виде одного или нескольких взаимодействующих процессов. Процессы – это логически завершенные, преимущественно значительные по объему работы, на которые можно и целесообразно разбить выполняемый алгоритм. Процессы выступают в роли элементарных работ, а структура общего алгоритма должна устанавливать правила взаимодействия процессов. При формировании процессов основная проблема – выделение элементарных работ, выполнение которых в вычислительной системе подлежит распараллеливанию.

С точки зрения простоты организации параллельных вычислений, конечно, удобнее делить задачу на «крупные» подзадачи, однако это может оказаться

препятствием для использования большого количества процессоров, а следовательно, для достижения высокого ускорения. Выбор нужного уровня декомпозиции вычислений (степени дробления операций) является трудно формализуемым этапом и требует опыта решения прикладных задач рассматриваемого типа. Если задача выделения элементарных (неделимых) так называемых *базовых* подзадач решена, далее возможна формализация распараллеливания алгоритма, допускающая автоматизацию процесса.

Исходным при этом является ориентированный граф алгоритма. Определяющими при распараллеливании по графу алгоритма являются два фактора:

- информационная и логическая взаимосвязь операторов (использование одними операторами выходной информации других операторов);
- объем работ в каждом операторе.

Связанные с объемом работ временные характеристики операторов t_i, $i=1,m$ называют их весом. Если вычислительная система однородная, достаточно оценить время t_i выполнения каждого оператора на одном процессоре. В этом случае говорят, что t_i, - скалярный вес оператора i. Если вычислительная система неоднородная, то процессор каждого типа $j=1,k$ выполняет оператор i за разное время. При этом вес можно представить вектором $t_i = \left[t_{i,1}, t_{i,2}, ..., t_{i,k}\right]^T$, $t_{i,j}, \quad i = 1,m, \quad j = 1,k$ – целое. Если j-й процессор не выполняет оператор i, полагают $t_{i,j} = \infty$.

При составлении информационно-логической граф-схемы алгоритма необходимо руководствоваться действительной зависимостью между операторами, обусловленной связями между ними по информации. Далее приводится пример построения взвешенного направленного графа для задачи вычисления массива значений, являющихся дискретными значениями переходного процесса в системе.

6.2 Построение графа алгоритма вычисления переходного процесса

Построение взвешенного направленного графа удобно иллюстрировать на конкретном примере. Чтобы пример имел содержательный смысл, рассмотрим задачу вычисления массива данных, являющихся дискретными значениями переходного процесса в системе, описываемой дифференциальным уравнением второго порядка с правой частью в виде полинома второго порядка. Основные доводы в пользу этого примера следующие. Граф алгоритма обладает вычислительной простотой, вместе с тем содержит разветвления. Это позволит нам на простом примере наглядно показать все основные правила преобразования графов.

Решение дифференциального уравнения

$$\left(a_0 p^2 + a_1 p \ + a_2\right) \cdot x(t) = b_0 t^2 + b_1 t \ + b_2, \qquad (6.1)$$

где $p=d/dt$ – алгебраизированный оператор дифференцирования, $x(t)$ - искомый процесс, а $a_0, a_1, a_2, b_0, b_1, b_2$ – заданные коэффициенты, представляют собой сумму его частного интеграла $x_ч(t)$ и общего интеграла $x_о(t)$ соответствующего однородного уравнения. Частное решение уравнения (6.1) в предположении, что система имеет запас устойчивости (т.е. вещественная часть корней характеристического уравнения не равна нулю), имеет вид

$$x_ч(t) = A_0 t^2 + A_1 t \ + A_2, \qquad (6.2)$$

где

$$\left.\begin{aligned} A_0 &= b_0 / a_2, \\ A_1 &= \left(b_1 a_2 - 2 b_0 a_1\right) / a_2^2, \\ A_2 &= \left(b_1 a_2^2 - 2 a_0 b_0 a_2 - a_1 b_1 a_2 + 2 b_0 a_1^2\right) / a_2^3. \end{aligned}\right\} \qquad (6.3)$$

Выражения для вычисления значений составляющей процесса $x_о(t)$ при различных типах корней характеристического уравнения сведены в таблицу 6.1. Здесь α, α_1, α_2, β – абсолютные значения соответствующих величин.

Алгоритм решения задачи можно представить в виде, показанном на рис. 6.1. Здесь вид некоторых формул изменен по сравнению с таблицей из методических соображений, что, впрочем, может оказаться целесообразным и по существу.

Таблица 6.1 Возможные решения дифференциального уравнения

Вещественные некратные корни α_1, α_2	Вещественный кратный корень α	Пара комплексных корней $\alpha \pm j\beta, \alpha, \beta$
$x_o(t) = C_1 e^{-\alpha_1 t} + C_2 e^{-\alpha_2 t}$, где $C_1 = \frac{\alpha_2 x_0 + x_0'}{\alpha_2 - \alpha_1}$, $C_2 = \frac{\alpha_1 x_0 + x_0'}{\alpha_1 - \alpha_2}$.	$x_o(t) = (C_1 + C_2 t) e^{-\alpha t}$, где $C_1 = x_0$, $C_2 = x_0' + \alpha x_0$.	$x_o(t) = (C_1 \cos \beta t + C_2 \sin \beta t) e^{-\alpha t}$, где $C_1 = x_0$, $C_2 = (x_0' + \alpha x_0)/\beta$.

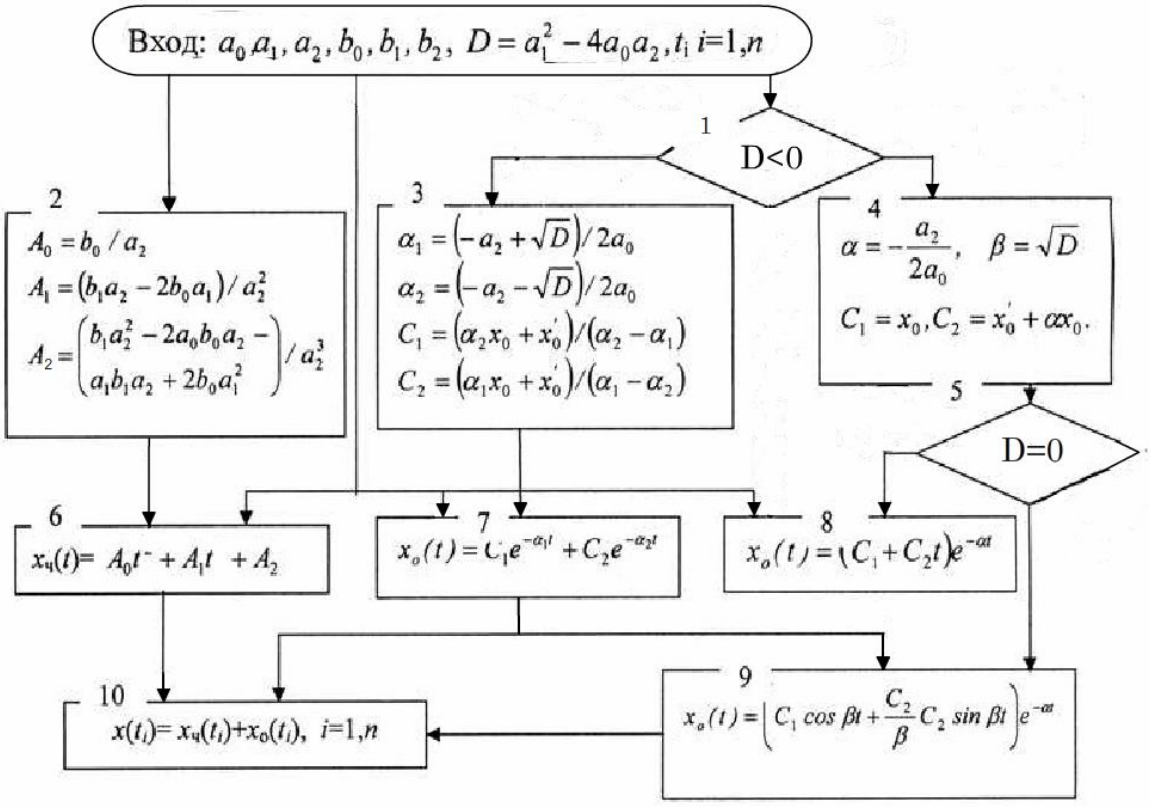


Рис. 6.1 Блок-схема алгоритма

На схеме отражены два типа операторов: логические (ромбом) и арифметические (прямоугольником). Логические определяют связи по управлению. Они задают состав и порядок выполнения операторов. Прочие операторы определяют только порядок выполнения операторов, определяемый информационными связями между ними.

Возможность распараллеливания алгоритмов зависит от степени детализации операторов. Например, на приведенной схеме любой из блоков, включающий совокупность арифметических выражений, может быть представлен в виде

параллельных арифметических операторов. Далее каждый блок, помеченный цифрой, мы будем называть оператором, независимо от того какую совокупность соотношений или задач он представляет.

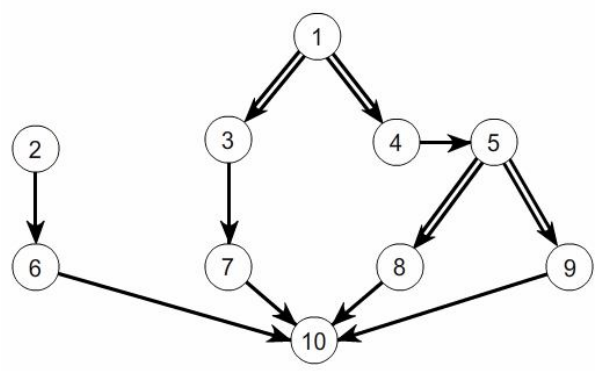


Рис. 6.2 Граф-схема алгоритма

Соответствующая блок-схеме алгоритма (рис. 6.1) граф-схема приведена на рис. 6.2. Здесь множество $\mathbf{X} = \{i\} = \{i = \overline{1,m}\}$ вершин графа соответствует множеству операторов алгоритма. Для определенности будем полагать, что оператор 1 закреплен за процессором 1, оператор 2 – за процессором 2 и т.д. Множество дуг состоит из двух типов, определяющих связи по управлению (двойные стрелки) и по информации (обычные стрелки).

Для составления расписания выполнения алгоритма должно быть также задано множество $\mathbf{P} = \{t_i\}, \quad i = \overline{1,m}$ весов вершин, определяющих время выполнения каждого оператора. На этапе решения задачи выявления операторов, которые могут выполняться независимо и параллельно, веса вершин могут не использоваться. Поэтому, исходя из задачи настоящего раздела, на рис. 6.2 для сокращения записей они не приводятся.

Переход от обычной блок-схемы алгоритма к модели в виде граф-схемы дает более ясное представление о структуре алгоритма, его свойствах и возможности направленных преобразований. Заметим, что каждая связь по управлению одновременно является связью по информации, т.к. она определяет строгий порядок следования операторов, задаваемый логической переменной. В простых случаях выявить операторы, которые могут выполняться независимо и параллельно, можно непосредственно по граф-схеме алгоритма. Если граф-схема имеет большое число ветвей, такой анализ становится затруднительным. В этом случае анализ граф-схем можно проводить с использованием формальных правил преобразований соответствующих матриц.

6.3 Построение и преобразование матрицы следования

Для удобства исследования и преобразования графов в рассмотрение вводят матрицу следования *S*. Вершине (оператору) *i* графа ставят в соответствие *i*-ю строку и *i*-й столбец матрицы. Элемент (*i*,*j*) этой матрицы будем отмечать знаком •, если между операторами с этими номерами существует связь вида $j \Rightarrow i$ (по управлению или информации). Для различения типа операторов там, где это необходимо, связи по управлению будем отмечать знаком **C**, а связи по информации – **I**.

Можно показать [1], что по графу без контуров может быть построена треугольная матрица следования с нулевыми элементами на главной диагонали, что упрощает некоторые задачи их анализа. На рис. 6.3 приведена матрица следования, соответствующая граф-схеме, показанной на рис. 6.2.

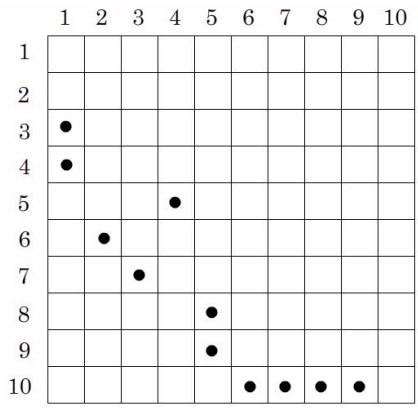


Рис. 6.3 Матрица следования

Для дальнейшего исследования матрицы *S* необходимо отразить в ней (неявные) связи между операторами, которые реально существуют и определяются задающими связями, но непосредственно между ними в графе отсутствуют. Их введение необходимо для выявления ветвей связанных операторов, которые не могут выполняться параллельно. Кроме того, введение этих связей позволяет выявить контуры в графе. Рассмотрим способы установления этих связей.

Если существуют задающие связи $\beta \to \gamma$ и $\gamma \to \delta$, но нет задающей связи $\beta \to \delta$, то дополнение вычислительной схемы такими связями не меняет результата решения задачи, а лишь подтверждает недопустимость определенного порядка следования операций. Связи, которые введены направленно внутри всех пар операторов, принадлежащих одному пути в графе *G* и не связанные задающими связями, образуют множество транзитивных связей [1]. Транзитивные

связи могут быть введены путем формальных преобразований матрицы S по следующему алгоритму.

Строки матрицы просматриваются последовательно сверху вниз. В каждой (i-й) строке просмотр элементов производится в порядке увеличения номеров столбцов. Если в очередном (j-м) столбце имеется знак •, указывающий на существование связи по информации или управлению, одноименные элементы строк с номерами i и j складываются по правилу дизъюнкции (или): •∨•=•, •∨0=•, 0∨•=•, 0∨0=0, т.е. в просматриваемую строку добавляются все знаки •, которые имеются в j-й строке. На рис. 6.4 приведена треугольная матрица следования, полученная из матрицы, показанной на рис. 6.3, путем ее дополнения транзитивными связями.

	1	2	3	4	5	6	7	8	9	10
1										
2										
3	•									
4	•									
5	•			•						
6		•								
7	•		•							
8	•			•	•					
9	•			•	•					
10	•	•	•	•	•	•	•	•	•	

Рис. 6.4. Матрица следования S, дополненная транзитивными связями

При построении алгоритмов распараллеливания часто приходится иметь дело с матрицами следования общего (не треугольного) вида. Если исходная матрица не треугольная, просмотр строк матрицы S производится неоднократно до установления факта ее неизменности. В результате указанных преобразований вводятся транзитивные связи, с помощью которых устанавливается факт наличия контура в информационно-логическом графе. О наличии контуров свидетельствуют ненулевые диагональные элементы.

Например, если предположить, что в рассматриваемом примере кроме указанных на рис. 6.2 и 6.3 связей существует также связь 10⇒4, матрица следования будет иметь вид, показанный на рис. 6.5, а. В результате двух шагов дополнения матрицы транзитивными связями, показанных на рис. 6.5, б и в матрица окончательно принимает неизменный вид, который показан на рис. 6.5, в. Эта матрица имеет ненулевые диагональные элементы (4,5,8,9,10), что указывает на существование контура. Нетрудно заметить, что именно операторы с указанными номерами входят в этот контур.

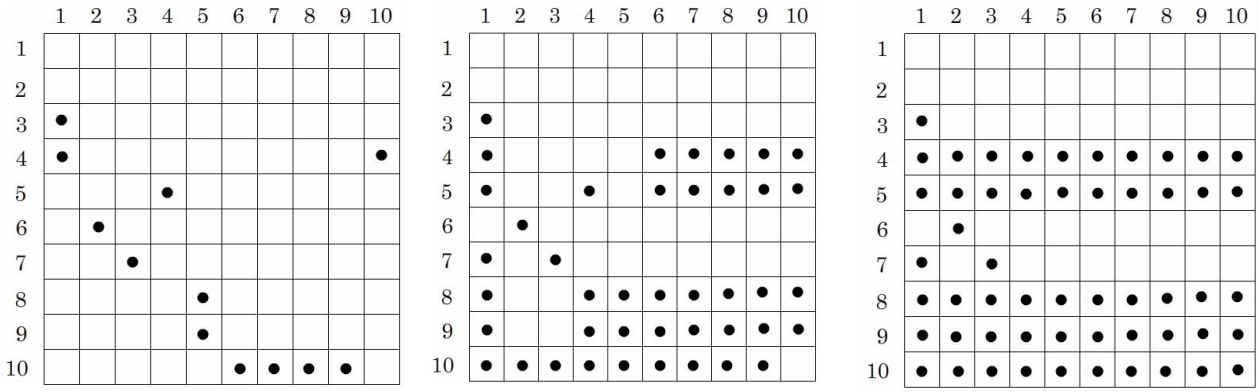


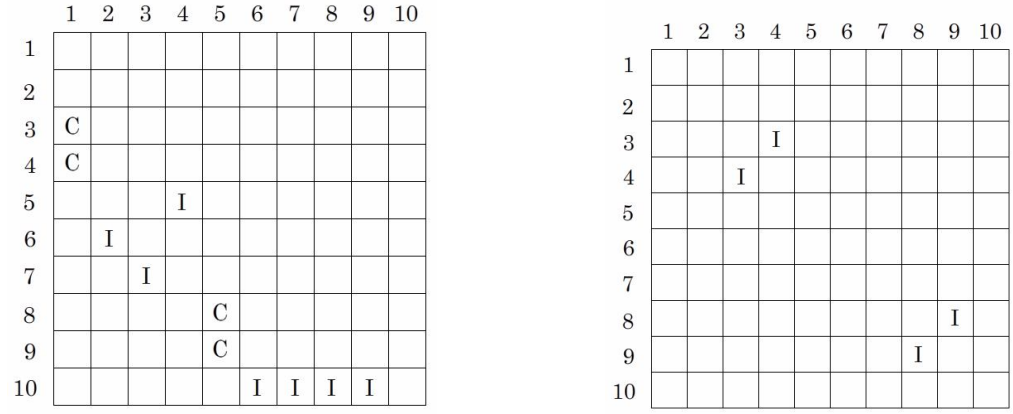
Рис. 6.5 Выявление контуров в графе: а – исходная матрица; б, в – матрицы – дополненные транзитивными связями (1-й шаг и 2-й шаг соответственно)

Вернемся к исходному примеру. Для того чтобы в явном виде показать, какие операторы не могут выполняться параллельно, необходимо использовать только информационные связи. Дело в том, что эти связи прямо указывают на то, какие операторы не могут выполняться одновременно. Например, в схеме на рис. 6.2 операторы 3, 4 не могут выполняться не только одновременно, но даже при одной реализации алгоритма. Такие операторы называются логически несовместимыми. Для выявления таких операторов используются матрицы L - логической несовместимости. Рассмотрим методику их формирования.

6.4. Выявление логически несовместимых операторов

Вначале по исходной треугольной матрице S, в которой связи по управлению отмечены знаком **C**, а связи по информации – **I,** формируются задающие связи логической несовместимости операторов по следующим правилам. Последовательно просматривают столбцы j=1,m матрицы S. Если очередной элемент (i_μ,j)=**C** и ранее в этом столбце также были зафиксированы элементы (i_k,j)= **C**, k=1,μ –1, то все элементы в i_μ-й строке с номером, совпадающим с номером строки, в которой ранее встречался знак **C**, заполняются единицами до i_μ-1 -го столбца включительно. Для каждой заполненной описанным способом единицы затем в матрицу вносится симметричная относительно главной диаго-

нали единица. Матрица следования граф-схемы алгоритма (рис. 6.2) с указанием отдельно связей по управлению и информации и соответствующая ей матрица логической несовместимости операторов L приведены на рис. 6.6.



	1	2	3	4	5	6	7	8	9	10
1										
2										
3	C									
4	C									
5				I						
6		I								
7			I							
8					C					
9					C					
10						I	I	I	I	

а)

	1	2	3	4	5	6	7	8	9	10
1										
2										
3				I						
4			I							
5										
6										
7										
8									I	
9								I		
10										

б)

Рис. 6.6. Исходная матрица следования (а) и соответствующая ей матрица логической несовместимости операторов (б)

Далее на основе заданных связей логической несовместимости определяется несовместимость для всех операторов схемы. Для этого вводятся транзитивные связи логической несовместимости по следующим правилам.

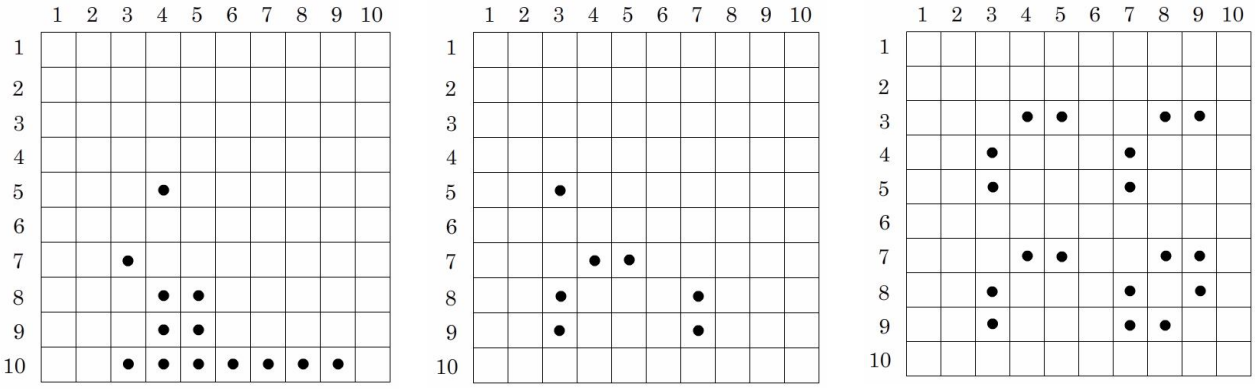
Последовательно просматривают строки треугольной матрицы следования S, дополненной транзитивными связями (нулевые строки пропускают). В очередной ненулевой i-й строке матрицы S анализируют множество ненулевых элементов. Из этого множества исключают номера операторов, образующих входы матрицы S, т.е. обнуляют элементы в столбцах, номера которых совпадают с номерами нулевых строк.

С использованием оставшегося множества номеров операторов (ненулевых элементов матрицы S) последовательно, начиная с первой строки, формируют строки i^*. Для этого объединяют по операции конъюнкции (И) строки матрицы L с номерами, соответствующими номерам столбцов матрицы S, в которых остались ненулевые элементы. Если ненулевой элемент в строке один, то строку i^* полагают равной строке матрицы L с номером, равным номеру столбца ненулевого элемента в анализируемой строке.

Далее обновляют матрицу L . Новую i-ю строку в матрице L формируют на основе ее старого значения и вновь сформированной строки i^* с помощью операции дизъюнкции $i=i\vee i^*$.

После обновления очередной строки матрицы L i-й столбец матрицы L полагают равным вновь сформированной i-й строке. Далее с использованием обновленной матрицы формируют очередную строку i^* по правилу, описанному выше. Процесс формирования продолжается до исчерпания строк.

На рис. 6.7 для используемого здесь сквозного примера показано построение матрицы L логической несовместимости операторов с транзитивными связями в соответствии с описанными правилами. Матрица логической несовместимости операторов, дополненная транзитивными связями логической несовместимости, позволяет судить о том, могут или не могут конкретно взятые операторы выполняться при одной реализации алгоритма.



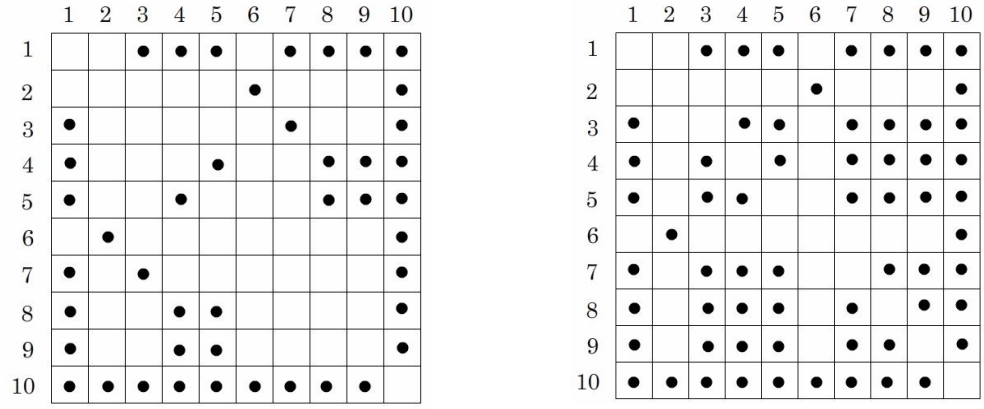
а) б) в)

Рис. 6.7 Дополнение матрицы логической несовместимости транзитивными связями: а – матрица S с исключенными входами; б – матрица, составленная из строк i^*; в – обновленная матрица логической несовместимости

При распределении работ между процессорами важно установить множество тех операторов, внутри которого имеет смысл решать задачу распараллеливания выполнения операторов. Для этого необходимо объединить информацию о логической несовместимости операторов и их информационно-логической связи по следующим правилам.

Вначале по матрице *S*, дополненной транзитивными связями, строят матрицу следования *S′* путем симметричного отображения элементов матрицы *S* относительно главной диагонали (т.е. путем сложения матрицы с ее транспонированной). Далее на сформированную матрицу *S*' накладывают матрицу *L*, дополненную транзитивными связями. Каждый элемент новой матрицы *M* формируется из одноименных элементов матриц *S′* и *L* по правилу дизъюнкции. Нулевые элементы полученной матрицы *M* указывают множество тех операторов, каждый из которых при выполнении некоторых условий может быть выполнен одновременно с данным, т.е. он информационно или по управлению не зависит от этого оператора и не является с ним логически несовместимым. На рис. 6.8 приведены построенные по указанному правилу матрицы *S′* и *M*.

Симметричную матрицу *M* называют матрицей независимости. Она отражает информационно-логические связи между операторами без учета их ориентации, а также связи логической несовместимости операторов с учетом всех транзитивных связей. Например, из матрицы *M* следует, что оператор 2 может выполняться независимо, а следовательно, если необходимо, то и параллельно с операторами 5 и 8, однако нельзя говорить об их взаимной независимости, так как существует связь 5⇒8. Если не рассматриваются связи по управлению, то матрица независимости *M* совпадает с матрицей следования *S*, дополненной транзитивными связями.



а) б)

Рис. 6.8 Матрица следования S' (а) и соответствующая матрица независимости *M*= *S*'∨*L* (б)

Говорят, что α и β взаимно независимые операторы (ВНО), если в матрице $M(\alpha,\beta)=(\beta,\alpha)=0$. Операторы $\{\alpha_i\}$, i=1,…,s образуют полное множество ВНО, если для любого оператора $j \notin \{\alpha_i\}$, существует пара элементов матрицы независимости $M(\alpha_i,j)=(j,\alpha_i)=1$, $i \in \{1,\ldots,s\}$. При решении задач распараллеливания необходимо перебирать все возможные полные множества ВНО и находить максимально полное множество ВНО, содержащее максимальное число операторов. Для его нахождения используется следующее свойство. Если два оператора μ и ν взаимно независимы, то при сложении по правилам дизъюнкции μ-й и ν-й строк матрицы независимости M образуется строка, в которой элементы в μ-м и ν-м столбцах обязательно нулевые.

Процедуру выявления максимально полного множества ВНО рассмотрим на примере матрицы, представленной на рис. 6.8, б. При сложении десятой строки матрицы M (по правилу дизъюнкции) с остальными строками получаются строки, содержащие все единичные элементы. Следовательно, оператор 10 образует полное множество ВНО. При сложении строк, соответствующих операторам 1 и 2, получается строка с нулевыми элементами в 1 и 2 столбцах. Следовательно, {1,2} - полное множество ВНО с числом элементом, превышающим ранее найденное. Множества с двумя элементами образуют также операторы {1,6}, {2,3}, {2,4}, {3,6}, {4,6}, {5,6}, {6,7}, {6,8}, {6,9}. Установленные полные множества ВНО содержат не более двух элементов. При сложении трех и более строк матрицы M в любых сочетаниях не удается получить строку, в которой элементы с номерами, совпадающими с номерами складываемых строк, оказались нулевыми. Следовательно, множество ВНО с двумя элементами является максимально полным множеством ВНО.

В заключение обратим внимание на тот факт, что начиная с этапа, когда была построена граф-схема алгоритма, все операции с соответствующими матрицами, включая поиск максимально полного множества ВНО, строго формализованы и могут быть реализованы в виде алгоритма. Следовательно, нет никаких препятствий для построения полностью автоматических (на указанных

этапах) процедур формирования параллельного алгоритма задачи. Однако для этого, как мы видели, необходимо структурировать алгоритм, т.е. задать операции и указать последовательность их реализации. Это абсолютно творческий этап, требующий высокой математической культуры и опыта.

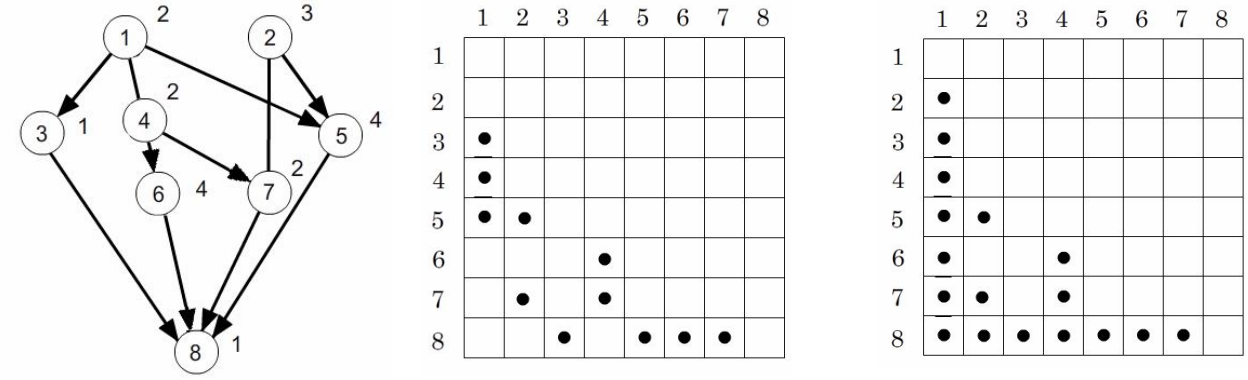
В ходе структурного анализа алгоритма, возможно, придется переформулировать какие-то части задачи, чтобы они эффективно решались с применением параллельных алгоритмов. При проведении такого анализа уже на начальных этапах требуется знание архитектуры многопроцессорной системы и данных о производительности составляющих ее процессоров. Это чрезвычайно трудоемкий процесс. Тем не менее эти усилия обычно не напрасны, т.к. параллельные алгоритмы, построенные с учетом конкретной архитектуры параллельной ЭВМ, как правило, дают наибольшее ускорение.

Лекция 7

Временные характеристики алгоритмов

7.1 Определение и характеристики информационного графа

Достаточно широким классом являются алгоритмы, при одной реализации которых выполняются все операторы. Такие алгоритмы не содержат логических операторов и представляются информационными графами, в которых отсутствуют связи по управлению. Для обозначения связей по информации будем использовать символ (•), а соответствующие элементы называть *единичными*. На рис. 7.1 в качестве примера приведены граф-схема алгоритма и соответствующие ему матрица следования и матрица следования, дополненная транзитивными связями.



а б в

Рис. 7.1 Информационная граф-схема алгоритма со скалярными весами вершин: а – информационный граф; б – матрица следования; в – матрица следования *S* с транзитивными связями

Алгоритмы указанного типа, во-первых, имеют место, когда речь идет о распараллеливании множества работ значительного объема, когда операторами являются некоторые логически завершенные достаточно крупные взаимодействующие подзадачи общего алгоритма. Во-вторых, даже если это не так и решается задача распараллеливания алгоритма с логическими операторами, все равно необходимо рассмотреть все возможные варианты, в которых все операторы обязаны выполняться при одной реализации. Каждый отдельно взятый вариант при этом представляется информационным графом.

Важнейшей характеристикой информационного графа является длина критического пути. Длиной пути в информационно-логическом графе со скалярными весами вершин называют сумму весов вершин, составляющих этот путь. Путь максимальной длины $T_{кр}$ называют *критическим*. Если пути в графе имеют одинаковую длину, любой из них может считаться критическим.

Если отсчет времени ведется с момента начала выполнения операторов, то момент (τ_{1i}, $i=\overline{1,m}$) окончания выполнения любого (*i*-го) оператора не может быть меньше максимальной из длин всех путей, заканчивающихся вершиной, соответствующей этому оператору. Величина τ_{1i} является *ранним сроком* выполнения данного оператора. Если выполнение алгоритма ограничено временем *T*, то для каждого оператора можно найти также и *поздний срок* окончания его выполнения $\tau_{2i}(T)$. Он определяется как разность между величиной *T* и максимальной из длин путей, в первую из вершин которых входит дуга, исходящая из вершины, соответствующей данному оператору. При $T=T_{кр}$ ранние и поздние сроки окончания выполнения операторов, находящихся на критическом пути, совпадают.

7.2 Определение ранних сроков выполнения операторов

Ранние и поздние сроки окончания выполнения операторов удобно определять с использованием матрицы следования. Общая процедура анализа графа со скалярными весами вершин t_j строится в виде следующей последовательности шагов.

Положив $\tau_{11}=\tau_{12}=...=\tau_{1m}=0$, последовательно обрабатывают строки матрицы следования *S*. Если очередная (*j*-я) строка не содержит единичных элементов, полагают $\tau_{1j}=t_j$. Если *j*-я строка содержит единичные элементы, из множества $\{\tau_{11},...,\tau_{1m}\}$ выбирают подмножество элементов $\{\tau_{1j}\}$, $\nu=\overline{1,k}_j$, соответствующих номерам единичных элементов *j*-й строки. В случае, когда все они отличны от нуля, полагают

$$\tau_{1j} = \max_{\nu=1,\dots,k_j} \tau_{1j_\nu} + t_j . \qquad (7.1)$$

Если же среди элементов множества $\{\tau_{1j}\}$, $\nu = \overline{1,k}_j$ есть хотя бы один нулевой (т.е. данное значение раннего срока окончания выполнения оператора еще не определено), переходят к следующей необработанной строке.

Например, по матрице *S*, представленной на рис. 7.1, можно определить:

$\tau_{11} = t_1 = 2$; $\tau_{12} = t_2 = 3$;

$\tau_{13} = \tau_{11} + t_3 = 3$; $\tau_{14} = \tau_{11} + t_4 = 4$;

$\tau_{15} = max\{\tau_{11}\ \tau_{12}\} + t_5 = 7$; $\tau_{16} = \tau_{14} + t_6 = 8$;

$\tau_{17} = max\{\tau_{12}, \tau_{14}\} + t_7 = 6$; $\tau_{18} = max\{\tau_{13}, \tau_{15}, \tau_{16}, \tau_{17}\} + t_8 = 9$.

Для учета фактора различной производительности процессоров может быть построена информационная граф-схема алгоритма с векторными весами вершин. Анализ таких граф-схем существенно сложнее, вследствие значительного увеличения числа возможных вариантов распределения работ. Для определения наилучшего плана распределения работ, строго говоря, необходимо осуществлять перебор всех возможных вариантов, в каждом из которых каждый оператор закреплен за определенным процессором. Подробное изложение методов анализа различных граф-схем можно найти в монографии [1].

7.3 Определение поздних сроков выполнения операторов

Поздние сроки окончания выполнения операторов при заданном значении T можно найти по следующему алгоритму.

На первом шаге полагаем $\tau_{21}(T) = \dots = \tau_{2m}(T) = 0$. Находим первый справа столбец из необработанных еще столбцов матрицы *S*. Предположим *j* – номер необработанного столбца. Если j -й столбец не содержит единичных элементов, полагаем

$\tau_{2j}(T) = T$.

Если же j -й столбец содержит единичные элементы, выбирают элементы множества

$$\{\tau_{21}(T),...,\tau_{2m}(T)\},$$

соответствующие номерам единичных элементов j -го столбца. Если все выбранные таким образом элементы

$$\{\tau_{2j_\nu}(T)\}\subset\{\tau_{21}(T),...,\tau_{2m}(T)\},\qquad \nu=\overline{1,k_j}$$

отличны от нуля, полагают

$$\tau_{2j}(T)=\min_{\nu=1,...k_j}\{\tau_{2j_\nu}(T)-t_{j_\nu}\}. \qquad (7.2)$$

Если же среди элементов множества $\{\tau_{2j}(T)\}$ есть хотя бы один нулевой (т.е. значение позднего срока окончания выполнения оператора еще не определено), осуществляется переход к следующему необработанному столбцу. Если матрица следования треугольная, поздние сроки окончания выполнения операторов определяются за один просмотр столбцов (справа налево).

Для указанного на рис. 7.1 примера имеем:

$$\tau_{28}(10)=10;\qquad \tau_{27}(10)=\tau_{28}(10)-t_8=9;$$

$$\tau_{26}(10)=\tau_{28}(10)-t_8=9;\qquad \tau_{25}(10)=\tau_{28}(10)-t_8=9;$$

$$\tau_{24}(10)=min\{\tau_{26}(10)-t_6,\tau_{27}(10)-t_7\}=5;$$

$$\tau_{23}(10)=\tau_{28}(10)-t_8=9;$$

$$\tau_{22}(10)=min\{\tau_{25}(10)-t_5,\tau_{27}(10)-t_7\}=5;$$

$$\tau_{21}(10)=min\{\tau_{23}(10)-t_3,\ \tau_{24}(10)-t_4,\ \tau_{25}(10)-t_5\}=3.$$

Для удобства представления наряду с представлением алгоритмов в виде информационного графа используют временные диаграммы выполнения операторов при заданных значениях времени начала (окончания) их выполнения. Операторы обозначаются прямоугольниками с длиной, равной времени их выполнения. Стрелки, связывающие прямоугольники, соответствуют дугам информационного графа. На рис. 7.2 представлены диаграммы выполнения работ для ранних и поздних сроков окончания выполнения операторов.

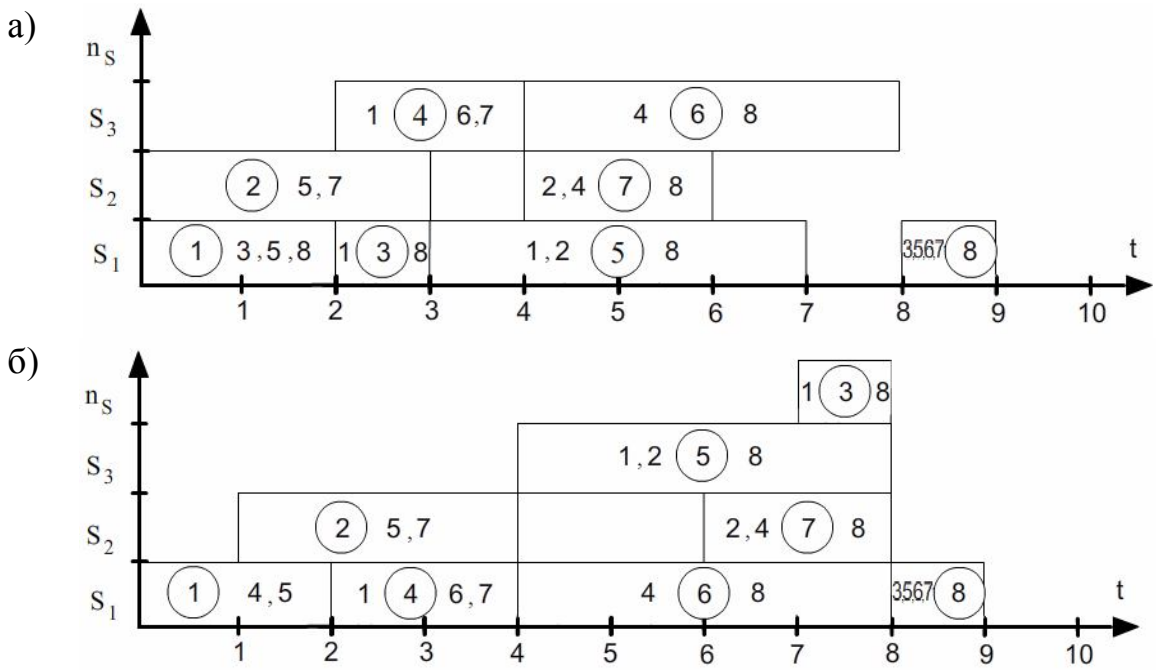


Рис. 7.2 Диаграмма выполнения работ при ранних (а) и поздних (б) сроках окончания выполнения операторов

7.4 Определение минимального числа процессоров, необходимых для выполнения алгоритма

Получим оценки для минимального числа процессоров, необходимых для выполнения алгоритма за время T. Пусть τ_j – произвольное значение момента окончания выполнения j-го оператора, $j = \overline{1,m}$, A – множество вершин верхнего яруса, т.е. вершин, в которые не входит ни одна дуга, а B – множество вершин нижнего яруса, т.е. вершин, из которых не исходит ни одна дуга. Тогда заданная информационным графом упорядоченность операторов и область определения значений τ_j описываются системой неравенств:

$$\begin{aligned}
&\tau_j \geq t_j, \text{ если } j \in A;\\
&\tau_j \geq \tau_j + t_j, \text{ если есть связь } i \to j,\ i \in X \setminus B,\quad j \in X \setminus A; \qquad (7.3)\\
&T \geq \tau_j, \text{ если } j \in B.
\end{aligned}$$

Соотношения (7.3) задают многоугольник в m-мерном пространстве $\{\tau_1,...,\tau_m\}$. Для произвольной точки этого пространства можно задать так называемую функцию плотности загрузки:

$$F(\tau_1,...,\tau_m,t)=\sum_{i=1}^{m} f(\tau_j,t), \qquad (7.4)$$

где $f(\tau_j,t)=\begin{cases}1 & при \quad t\in\lfloor\tau_j-t_i, \quad \tau_j\rfloor,\\ 0 & в\ противном\ случае.\end{cases}$

Значение функции F в каждый момент времени t совпадает с числом одновременно (параллельно) выполняемых в этот момент операторов. Например, по диаграмме, показанной на рис. 7.2, можно построить функцию плотности загрузки $F(2,3,3,4,7,8,6,9,t)$, которая при t=1 равна 2, при t=2 равна 3, при t=7 равна 2 и т.д.

Предположим теперь, что граф, дополненный транзитивными связями содержит l полных множеств ВНО (каждая пара таких множеств может иметь непустое пересечение). Обозначим $r_i, \quad i=\overline{1,l}$, число операторов, образующих i-е полное множество, и найдем

$$R=max\{r_1,...,r_l\}.$$

Тогда

$$R=\max_{(\tau_1,...,\tau_m)} F(\tau_1,...,\tau_m,t),$$

т.к. возможно такое распределение выполняемых операторов, когда на каком-то отрезке времени выполняются все операторы, входящие в полное множество ВНО с числом операторов R.

Из этого следует, что минимальное число процессоров одинаковой специализации и производительности, способных выполнить данный алгоритм за время $T\geq T_{кр}$, не превышает

$$R=max\{r_1,...,r_l\},$$

где $r_i, \quad i=\overline{1,l}$ - число операторов, входящих в i-е полное множество ВНО, составленное по соответствующему этому алгоритму информационному графу.

7.5 Построение оценок минимального числа процессоров, необходимых для выполнения алгоритма за заданное время

Для построения приближенных оценок минимального времени выполнения алгоритма при заданном числе процессоров *n*, либо числа процессоров для выполнения алгоритма за время *T* может использоваться так называемая функция загрузки вычислительной системы на отрезке $[\theta_1,\theta_2]\subset[0,T]$ для заданного набора $(\tau_1,...,\tau_m)$:

$$\Phi(\tau_1,...,\tau_m,\theta_1,\theta_2)=\int_{\theta_1}^{\theta_2}F(\tau_1,...,\tau_m,t)dt\,. \qquad (7.5)$$

Эта функция определяет суммарный объем работ (необходимое время) на указанном фиксированном отрезке. Например, для отрезка времени $[0,4]$ по рис. 7.2, а) можно определить: $\Phi(2,3,3,4,6,7,8,9,0,4)=8$.

На множестве допустимых, принадлежащих многоугольнику (7.3) планов алгоритма, характеризующихся различными наборами значений $(\tau_1,...,\tau_m)$, можно определить минимальную загрузку отрезка $[\theta_1,\theta_2]\subset[0,T]$:

$$\varphi(T,\theta_1,\theta_2)=\min_{\forall\{\tau_1,...,\tau_m\}}\Phi(\tau_1,...,\tau_m,\theta_1,\theta_2). \qquad (7.6)$$

Для того чтобы T было наименьшим временем выполнения алгоритма на *n* процессорах, либо *n* процессоров было достаточно для выполнения алгоритма за время *T*, необходимо, чтобы для данного отрезка времени выполнялось неравенство

$$\varphi(T,\theta_1,\theta_2)\le n\cdot(\theta_2-\theta_1). \qquad (7.7)$$

Необходимость этой оценки вытекает из того, что если для некоторого набора сроков окончания выполнения операторов $(\tau_1,...,\tau_m)$ существует минимальное значение $T=max\{\tau_1,...,\tau_m\}$, для обеспечения которого достаточно *n* процессоров, то в силу (7.5), (7.6) для любого отрезка времени

$$\varphi(T,\theta_1,\theta_2)\le\Phi(\tau_1,...,\tau_m,\theta_1,\theta_2)=\int_{\theta_1}^{\theta_2}F(\tau_1,...,\tau_m,t)dt\le n\cdot(\theta_2-\theta_1).$$

С использованием (7.7) можно построить оценку минимального числа n процессоров, необходимого для выполнения заданного алгоритма за время T. Для этого перебираются все возможные отрезки $[\theta_1,\theta_2]\subset[0,T]$, например, в следующем порядке:

$[0,1]$;

$[0,2]$; $[1,2]$;

$[0,3]$; $[1,3]$;$[2,3]$;

. . . . . .

$[0,T]$; $[1,T]$; . . . , $[T-1,T]$.

Всего таких отрезков будет $T(T-1)/2-1$. Для каждого отрезка определяется значение

$$n' = \frac{\varphi(T,\theta_1,\theta_2)}{\theta_2-\theta_1}. \qquad (7.8)$$

После перебора всех отрезков среди полученных по соотношению (7.8) значений находим максимальное – n', которое равно максимальному из значений, удовлетворяющих условию (7.7).

7.6 Построение оценок минимального времени выполнения алгоритма на заданном числе процессоров

Методика построения оценки для минимального времени T выполнения заданного алгоритма на n процессорах опирается на следующее утверждение. Пусть алгоритм, описываемый информационным графом со скалярными весами вершин, реализуется на n процессорах, а T_1 – некоторая оценка снизу времени выполнения алгоритма. Пусть также на отрезке времени $[\theta_1,\theta_2]\subset[0,T]$ выполняется соотношение

$$\varphi(T_1,\theta_1,\theta_2)-n\cdot(\theta_2-\theta_1)=d>0.$$

Тогда наименьшее время T реализации алгоритма удовлетворяет соотношению

$$T\geq T_1+d/n.$$

Алгоритм построения оценки для минимального времени строится следующим образом. Первоначально полагают

$$T_1 = \max\left\{\left]\frac{1}{n}\sum_{i=1}^{m} t_i\right[, T_{кр}\right\},$$

где $]x[$ означает операцию взятия ближайшего целого числа не меньшего x.

Затем организуется перебор всех отрезков, например, в той же последовательности, как описано в предыдущем разделе. Заметим, что при таком порядке перебора отрезков значение T_1 можно увеличивать в процессе выполнения алгоритма. Для каждого отрезка времени $[\theta_1, \theta_2]$ определяется значение

$$d = \varphi(T, \theta_1, \theta_2) - n \cdot (\theta_2 - \theta_1).$$

Если $d > 0$, новое значение оценки минимального времени определяется по соотношению

$$T_2 = T_1 + \left]\frac{d}{n}\right[.$$

Для найденного нового значения T_2 определяются поздние сроки выполнения операторов:

$$\tau_{2j}(T_2) = \tau_{2j}(T_2) + \left]\frac{d}{n}\right[, \; j = \overline{1,m}.$$

После перебора всех отрезков $[\theta_1, \theta_2]$ найденное окончательное значение и будет нижней оценкой минимального времени выполнения данного алгоритма на n процессорах.

7.7 Определение временных характеристик с учетом обмена информацией

Рассмотрим теперь случай, когда нельзя пренебречь временем обмена информацией при распределении множества информационно взаимосвязанных задач. Обозначим $R_{\alpha\to\beta}$ – объем информации, передаваемой в направлении дуги, соединяющей две вершины α и β. Граф, в котором каждая, соединяющая вершины α и β дуга, снабжена весом $R_{\alpha\to\beta}$, будем обозначать $G = (X, P, R, \Gamma)$.

Предположим l – параметр, характеризующий пропускную способность межпроцессорного обмена. Работы по обмену обладают спецификой, заключающейся в возможности их прерывания, т.е. реализации их в несколько приемов на некотором отрезке времени. В связи с этой спецификой каждая работа по обмену информацией наряду с другими вариантами реализации может быть выполнена условно введенным процессором за время $R_{\alpha\to\beta}/l$ в принятых единицах времени. На рис. 7.3, б приведен пример графа $G^* = \left(X^*, P^*, \Gamma^*\right)$, полученного путем дополнения, показанного на рисунке 7.3, а исходного графа $G = \left(X, P, R, \Gamma\right)$ условно введенными операторами.

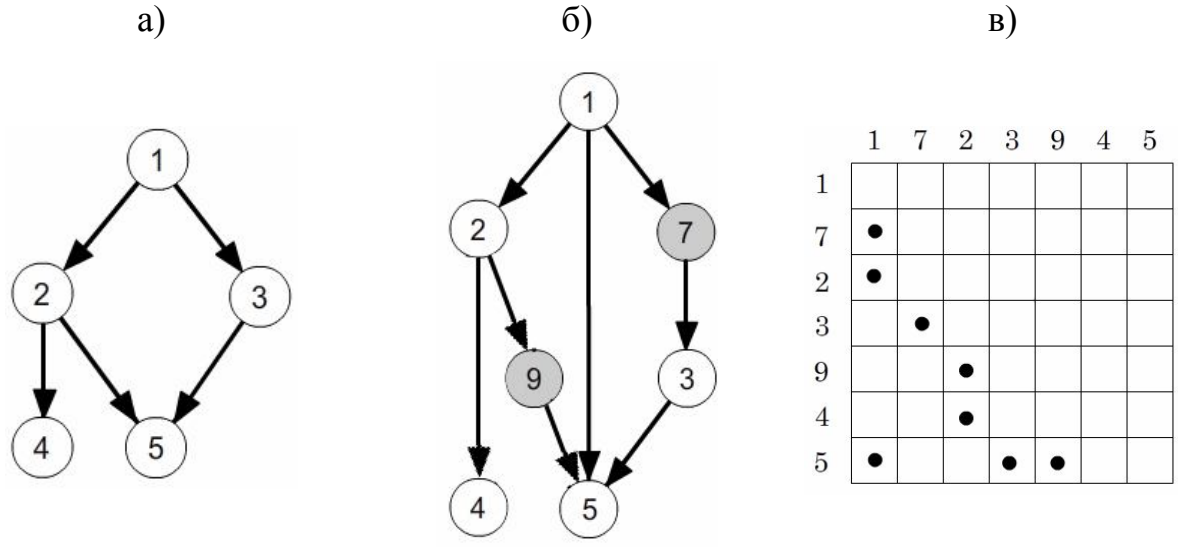


Рис. 7.3. Граф *G* (а), дополненный операторами обмена (7,9) (б), и соответствующая матрица следования *S* (в)

Один из вопросов, на который необходимо ответить при замене операции передачи данных условным оператором, возможно ли выполнение операторов α и β, принадлежащих одному пути в графе, на одном процессоре. Очевидно, что если в число операторов, входящих в этот путь, не входит ни один оператор обмена, операторы α и β могут выполняться на одном процессоре. Если этот путь содержит один оператор обмена между операторами α и β, они должны

выполняться на разных процессорах. Если же этот путь содержит два оператора обмена, то операторы, находящиеся между ними, выполняются на другом процессоре, а оператор, следующий за вторым оператором обмена, в принципе, может выполняться на том же процессоре.

Если существует несколько различных путей, соединяющих операторы α и β, то допустимость варианта выполнения этих операторов на одном процессоре имеет место только в том случае, если это допустимо для всех вариантов путей. Например, для графа, показанного на рис. 7.3, б существует несколько путей, соединяющих операторы 1 и 5 (1→2→9→5, 1→5, 1→7→3→5). Хотя второй путь (1→5) допускает выполнение операторов 1 и 5 на одном процессоре, однако этот вариант следует исключить, поскольку первый и третий пути не допускают этого.

Если известно минимальное время выполнения каждой работы по обмену (множество значений вида $R_{\alpha\to\beta}/l$), то, как и прежде, можно для каждого допустимого варианта обмена найти множество ранних $\{\tau_{1i}\}$, а при заданном значении T множество поздних $\{\tau_{2i}(T)\}$ сроков окончания всех работ, в том числе работ по обмену информацией.

Лекция 8

Распараллеливание алгоритмов по информационному графу

8.1 Постановки задач распараллеливания

Как уже отмечалось выше, основная проблема конструирования эффективных параллельных алгоритмов, по существу, всегда сводится к решению, с той или иной степенью строгости, проблемы отображения графа алгоритма на архитектуру вычислительной системы. В рамках подхода, основанного на использовании взвешенных направленных графов, обычно рассматриваются следующие две взаимно обратные задачи.

Задача 1. Для данного алгоритма, которому соответствует информационный граф G со скалярными весами вершин, и для времени T, отведенного для его выполнения, найти наименьшее необходимое число n процессоров, входящих в состав однородной вычислительной системы (ВС), и план выполнения операторов на них.

Задача 2. Для данного алгоритма, которому соответствует информационный граф G со скалярными весами вершин, найти минимальное время T и план реализации этого алгоритма на данной однородной ВС, в состав которой входят n процессоров.

При решении этих задач в настоящей лекции не будем учитывать потери времени на обмен информацией между процессорами. Такая ситуация может соответствовать отсутствию обмена данными между процессорами или наличию в ВС либо общей оперативной памяти, либо высокоскоростных каналов обмена данными, так что временем межпроцессорного обмена можно пренебречь.

8.2 Общая схема решения задачи определения минимально необходимого числа процессоров

Решение задачи 1 заключается в определении минимального числа *n*, для которого можно построить преобразованный граф $G' = (X, P, \Gamma')$, объединив вершины, соответствующие операторам каждого полного множества ВНО, содержащего $r > n$ операторов $r - n$ ориентированными дугами в *n* путей, не содержащих общих вершин. При этом длина критического пути в графе G' не должна превышать значение T. Суть алгоритма нахождения графа G' состоит в том, что выбрав начальное расписание, в котором все операторы занимают на оси времени крайнее левое положение, продвигаются шаг за шагом и находят точки на оси времени, в которых функция плотности загрузки превышает первоначально заданное значение *n*. Вводя дополнительные связи, допускаемые ограничением на *T*, пытаются уменьшить (сгладить) функции плотности. Если это не удается, вводят другую комбинацию связей. При переборе всех комбинаций связей число процессоров увеличивают на единицу и начинают процесс сначала. Рассмотрим алгоритм построения графа G' более подробно.

Вначале для заданного информационного графа $G = (X, P, \Gamma)$ со скалярными весами вершин и данного времени *T* находят значения ранних $\{\tau_{1i}\}$ и поздних $\{\tau_{2i}(T)\}$, $i = \overline{1, m}$ сроков окончания выполнения операторов. При этом для корректности задачи должно выполняться соотношение

$$\max_i \{\tau_{1i}\} = T_{кр} \le T .$$

Значения $\{\tau^{(v)}{}_{1i}\}$ фиксируют исходное расписание выполнения операторов за время, не превышающее *T*.

Для сформированного расписания в соответствии с алгоритмом, описанным в разделе 7.4, находят оценку *n* минимального числа процессоров, которое необходимо для выполнения заданного алгоритма, соответствующего графу $G = (X, P, \Gamma)$ за время, не превышающее *T*.

Далее полагают $\nu = 1, \quad G^{(\nu)} = G, \quad \tau^{(\nu)}{}_{1i} = \tau_{1i}, \ \tau^{(\nu)}{}_{2i}(T) = \tau_{2i}(T), \quad i = \overline{1,m}$ и ищут наименьшее значение $t_\nu = \tau^{(\nu)}{}_{1i} - t_j, \quad j \in \{1,...,m\}$, такое, что

$$F\left(\tau_{11}^{(\nu)}, \ldots, \tau_{1m}^{(\nu)}, t_\nu\right) = r_\nu > n. \qquad (8.1)$$

Если такое значение не находится, то *n* решение задачи.

Если существует t_ν, удовлетворяющее (8.1), выделяют множество операторов $A_\nu = \{\alpha_{j_\mu}\}, \quad \mu = 1, \ldots, r_\nu$, для которых

$$\tau_{1j_\mu}^{(\nu)} - t_{j_\mu} \le t_\nu < \tau_{1j_\mu}^{(\nu)}.$$

Заметим, что множество A_ν является подмножеством некоторого полного множества ВНО.

Затем вводят очередную комбинацию $r_\nu - n$ связей, так чтобы длина критического пути в изменившемся при этом графе $G^{(\nu)}$ не превысила *T*. Если такая комбинация найдена, значение счетчика числа итераций увеличивают на единицу: $\nu = \nu + 1$, запоминают новые значения $\{\tau^{(\nu)}{}_{1i}\}$ $\{\tau^{(\nu)}{}_{2i}(T)\}$ и вновь ищут наименьшее значение t_ν, удовлетворяющее (8.1) (если же искомой комбинации связей не существует, либо все они уже испытаны, напротив, значение счетчика числа итераций уменьшают на единицу: $\nu = \nu - 1$, вводят очередную комбинацию $r_\nu - n$ и также ищут наименьшее значение t_ν).

Если при уменьшении индекса числа итераций оказывается, что $\nu = 0$, т.е. на первом же шаге не подтверждается предположение, что *n* решение задачи, полагают $n = n + 1$, вновь задают значения $\nu = 1, \quad G^{(\nu)} = G, \quad \tau^{(\nu)}{}_{1i} = \tau_{1i}$, $\tau^{(\nu)}{}_{2i}(T) = \tau_{2i}(T). \ i = \overline{1,m}$ и повторяют поиск наименьшего значения t_ν, удовлетворяющего (8.1).

Подчеркнем, что построение графа $G' = (X, P, \Gamma')$ по описанной схеме является достаточным для решения задачи 1. Действительно, предположим внутри каждого полного множества ВНО с числом операторов $r > n$ в соответствии с

описанной схемой введено $r - n$ связей и n – наименьшее из целых чисел, для которых это возможно. Тогда в сформированном графе G' каждое полное множество ВНО содержит не более n операторов. Но это, в соответствии с выводами, содержащимися в разделе 7.3, как раз и означает, что минимальное число процессоров, способных выполнить данный алгоритм за время T, не может быть больше n. С другой стороны, допущение о том, что число процессоров меньше n, противоречит результату работы алгоритма, в соответствии с которым n – наименьшее из целых чисел, для которого возможно введение $r - n$ связей.

После нахождения графа-решения G' распределение множества операторов между n процессорами не представляет трудностей, т.к. известны найденные по этому графу ранние сроки выполнения операторов. В силу алгоритма при этих сроках одновременно могут выполняться не более n операторов. Поэтому остается только закрепить эти операторы за процессорами. Для этого необходимо просматривать значения ранних сроков начала выполнения операторов в порядке их неубывания и закреплять за процессорами, которые в эти моменты свободны. Это может быть легко реализовано даже в процессе решения вычислительной задачи.

8.3 Определения минимально необходимого числа процессоров методом направленного перебора

Описанная выше задача 1 – определения минимально необходимого числа процессоров, может быть решена методом направленного перебора (ветвей и границ). В данном случае ветвление определяется различными способами введения дополнительных связей в каждое полное множество ВНО, число операторов в котором превышает n, или его подмножество. Границы определяются допустимыми изменениями длины критического пути в графе после введения дополнительных связей.

Для оценки изменения длины критического пути может использоваться оценка, определяемая неравенством (7.7). При этом полезно использовать два следующих утверждения.

1. Если связь $\alpha \to \beta$ привела к формированию графа, длина критического пути в котором превышает T, т. е. $\tau_{1\alpha} > \tau_{2\beta}(T) - t_{\beta}$, то следующее введение связей не может исправить положение и вновь привести к уменьшению длины критического пути. Это означает, что сокращение длины критического пути может быть достигнуто лишь путем замены «неудачных» связей.

2. Если оператор α, входящий в некоторое полное множество ВНО, которое содержит $r > n$ операторов, удалось назначить для выполнения, так что ранние и поздние сроки окончания выполнения оставшихся операторов этого множества не изменились, то это назначение не ограничивает числа вариантов распределения этих операторов.

Последнее утверждение говорит о том, что можно перебирать не полные множества ВНО, а их подмножества, если удается заранее назначить некоторые операторы так, что на отрезке времени их выполнения плотность загрузки не превышает заданное число n процессоров.

Важнейшим этапом алгоритма поиска решения методом направленного перебора является ветвление при построении графа G', заключающееся в различных способах введения дополнительных связей. Как способ сокращения перебора на шаге 6 алгоритма, описанного в разделе 8.2, рекомендуется введение дополнительных связей не только по критерию допустимого увеличения длины критического пути, но и по значениям функции минимальной загрузки отрезка.

Для этого, вводя очередную комбинацию связей на ν-м шаге, которая не приводит к увеличению длины критического пути сверх заданного значения T, необходимо вновь пересчитать значение функции $\varphi(T, \theta_1, \theta_2)$ с учетом новых связей на всех отрезках $[\theta_1, \theta_2] \subset [0, T]$. Если при этом выполняется соотношение (7.7), данная комбинация связей может считаться удачной.

Задача выбора допустимой комбинации связей на шаге 6 алгоритма построения графа G' является отдельной важной проблемой, которая в основном определяет затраты времени на осуществление направленного перебора. Рассмотрим ее более детально.

8.4 Выбор допустимой комбинации связей

Отдельного рассмотрения требует выполняемая в ходе построения графа G' процедура выбора допустимой комбинации $r_\nu - n$ связей внутри множества ВНО, содержащего r_ν операторов. Для этого из соответствующей множеству ВНО $A_\nu = \{j_\mu\}$, $\mu = 1, \ldots, r_\nu$ $r_\nu \times r_\nu$ матрицы L_ν, все элементы которой равны нулю, формируется матрица следования, соответствующая введенным дополнительным связям. При этом следуют следующим правилам:

1. Диагональные элементы матрицы L_ν равны нулю (отсутствие контуров).
2. В каждой строке и каждом столбце содержится не более одной единицы.
3. Если $T = T_{кр}$, первые l строк закрепляются за операторами, принадлежащими критическому пути, и полагаются нулевыми.

Вводимые связи упорядочиваются так, что единица, соответствующая p-й связи $p = 1, \ldots, r_\nu - n$, оказывается записанной в строке, выше которой есть только $p-1$ ненулевых строк матрицы L_ν. Таким образом, p-я связь при проверке всех комбинаций пробегает строки от p до $r_\nu - [(r_\nu - n) - p] = n + p$, а при $T = T_{кр}$ от $l + p$ до $n + p$.

Алгоритм введения дополнительных связей можно описать в виде следующей последовательности шагов.

1. Каждому оператору $j_\mu \in A_\nu$ ставятся в соответствие значения $\tau^{(\nu-1)}{}_{1_\mu}$ и $\tau^{(\nu-1)}{}_{2\,j_\mu}(T)$, $\mu = 1, \ldots, r_\nu - n$.
2. Вводится некоторая комбинация связей, удовлетворяющая приведенным выше условиям (при этом единичные элементы располагаются в матрице

L_ν параллельно главной диагонали, начиная с позиции (1,2), или с позиции (l+1,1) при $T = T_{кр}$).

3. Последовательно просматриваются строки матрицы L_ν.

4. Пусть очередная μ-я анализируемая строка матрицы не содержит единичных элементов. Тогда полагают $\tau^*_{1j_\mu} = \tau^{(\nu-1)}_{1j_\mu}$ и переходят к выполнению шага 7.

5. Пусть очередная μ-я анализируемая строка матрицы содержит единичный элемент в λ-м столбце. Тогда, если уже найдено значение $\tau^*_{1j_\lambda}$, полагают $\tau^*_{1j_\mu} = \tau^*_{1j_\lambda} + t_{j_\mu}$. В противном случае выполняется шаг 3.

6. Если $\tau^*_{1j_\lambda} > \tau^{(\nu-1)}_{2j_\lambda}(T)$, данная комбинация связей недопустима и переходят к формированию следующей комбинации (шаг 8), в противном случае выполняется следующий шаг.

7. Если просмотр строк матрицы L_ν завершен, но не все значения $\tau^*_{1j_\mu}(T)$, $\mu = 1,\ldots,r_\nu$ найдены, повторно просматривают строки матрицы L_ν и переходят к выполнению шага 3. Если все указанные значения найдены, это означает, что искомая комбинация связей для шага 6 алгоритма, приведенного в разделе 8.2, также найдена. Эта комбинация вводится в информационный граф, полученный на $(\nu-1)$-м шаге указанного алгоритма, и определяются новые значения ранних и поздних сроков окончания выполнения операторов. Далее для этой комбинации вычисляется функция φ на всех отрезках $[\theta_1,\theta_2] \subset [t_\nu, T]$, и если для испытываемого значения n соотношение (7.7) не выполняется, данная комбинация может быть признана неудачной. Тогда поиск необходимой комбинации связей внутри данного множества ВНО должен быть продолжен со следующего шага.

8. Пусть в соответствии с упорядочением вводимых связей единичные элементы матрицы L_ν, соответствующие связям $p+1,\ldots,r_\nu - n$, занимают крайнее правое положение в строках матрицы с номерами $n+p+1,\ldots,r_\nu$ со-

ответственно. Эти элементы находятся под главной диагональю. Если $p = 0$, то комбинация единичных элементов – последняя из возможных, т.е. необходимой комбинации связей не существует.

Если $p > 0$, полагают равными нулю эти элементы и смещают единицу, соответствующую p-й связи по строке, в которой она расположена, на одну позицию вправо или в крайнее левое положение на следующую строку, которая в этом случае обязана быть нулевой. Это смещение производится так, чтобы по отношению к связям $1, \ldots, p-1$ выполнялось условие 2, т.е. в результате проверки условия 2 это смещение может быть произведено неоднократно.

9. Если $p \neq r_v - n$, вновь последовательно вводят единичные элементы матрицы L_v, соответствующие связям $p+1, \ldots, r_v - n$, так, что каждый из вводимых единичных элементов должен занимать крайнее левое положение в очередной строке, следующей за той, в которой введена p-я связь. В то же время вновь вводимая связь должна удовлетворять в совокупности с уже введенными связями условию 2.
10. Если новую комбинацию единичных элементов матрицы, удовлетворяющую условию 2, удалось ввести, устанавливаем, не содержит ли матрица L_v контуров. Если контуры есть, вновь выполняем шаг 8.
11. Если новая комбинация единичных элементов матрицы L_v удовлетворяет условиям 1 и 2, переходим к анализу комбинации связей на шаге 3.

8.5 Пример определения минимального числа процессоров

Найдем минимальное число процессоров, необходимое для выполнения алгоритма, представленного графом, показанным на рис. 8.1, за время T=7. В соответствии с алгоритмом, описанным в разделе 7.4, исследуем 28 значений функции минимальной загрузки отрезков $[\theta_1, \theta_2] \subset [0,7]$ и найдем нижнюю оценку минимального числа процессоров – n=2. При этом

$\tau_{11}=1, \tau_{12}=\tau_{13}=\tau_{14}=2, \tau_{15}=\tau_{16}=4, \tau_{17}=5,$ $\tau_{21}(7)=\tau_{22}(7)=4,$

$\tau_{23}(7)=\tau_{24}(7)=\tau_{26}(7)=6,$ $\tau_{25}(7)=\tau_{27}(7)=7.$

Легко проверить, что $F(\tau_{11},\tau_{12},\tau_{13},\tau_{14},\tau_{15},\tau_{16},\tau_{17},0)=4>2$, т.е. $t_1=0$. В формировании этого значения участвуют операторы 1,2,3,4. Составим квадратную матрицу L_1 (первоначально с нулевыми элементами) и будем вводить в нее два единичных элемента в соответствии с алгоритмом, описанным в разделе 8,5.

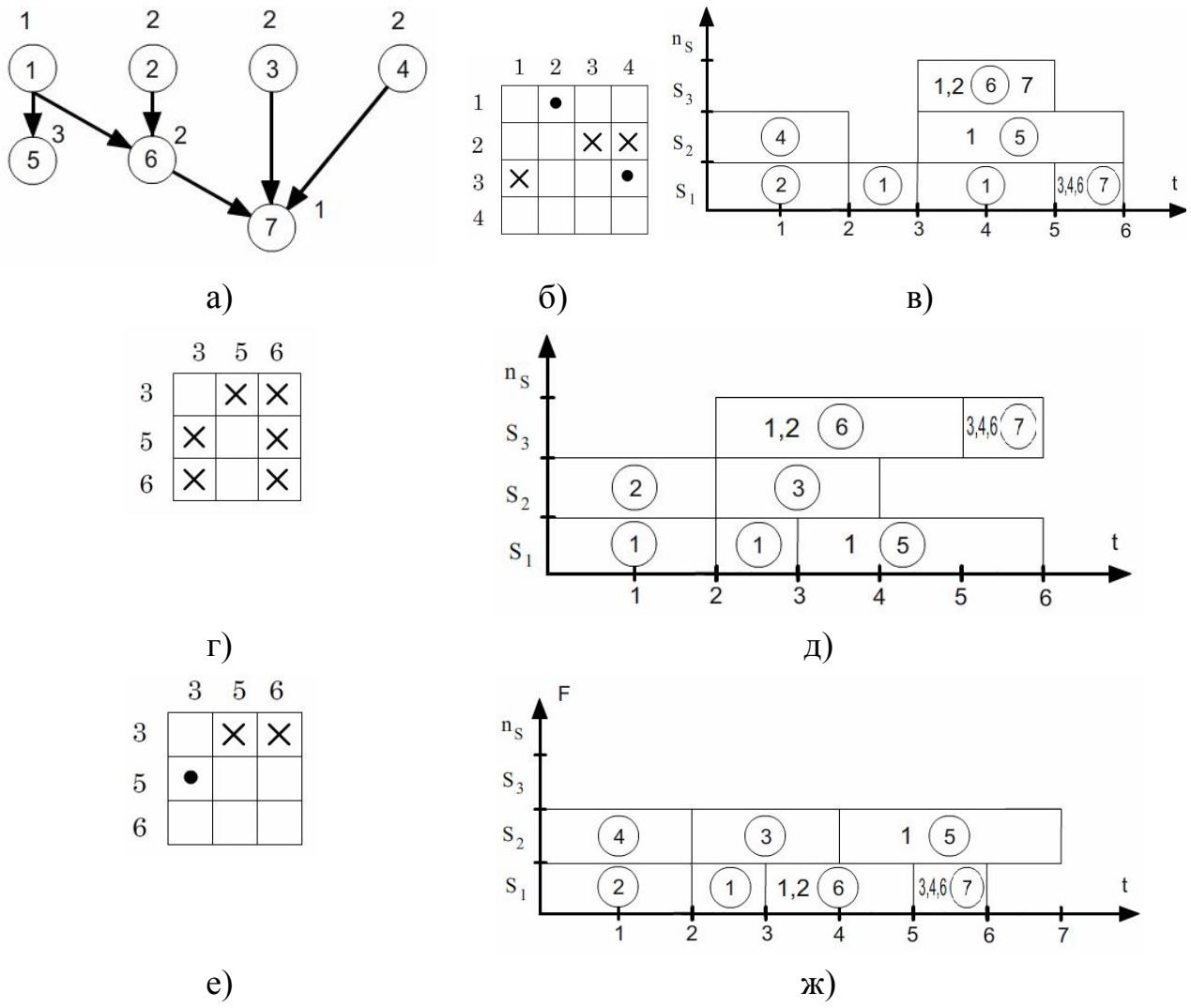


Рис. 8.1 Решение задачи определения минимального числа процессоров

Первая возможная комбинация двух таких элементов соответствует связям 3→2→1. При введении дуг, соответствующих этим связям, длина критического пути превышает 7. Новая комбинация 4→2→1 также приводит к недопустимо-

му увеличению длины критического пути. Пробуем еще одну комбинацию: 2→1→3. При введении дуг, соответствующих этим связям, длина критического пути не превышает допустимого значения. С учетом этих связей находим ранние и поздние сроки окончания выполнения операторов. В данном случае $\tau_{11}^{(1)}=3, \tau_{12}^{(1)}=\tau_{14}^{(1)}=2, \tau_{13}^{(1)}=\tau_{16}^{(1)}=5, \tau_{15}^{(1)}=\tau_{17}^{(1)}=6$. Соответствующая диаграмма реализации алгоритма приведена на рис. 8.1, в.

Найдем $t_2=3$, так как $F(3,2,5,2,6,5,6,3)=3>2$. Это значение функции плотности загрузки определяется выполнением операторов 3, 5, 6. Составляем матрицу L_2 (рис. 8.1, г) и стараемся ввести в ней единственный единичный элемент. Перебор всех возможных способов введения такого элемента приводит к недопустимой длине критического пути.

Возвращаемся на шаг назад к анализу матрицы L_1 и пробуем ввести другую допустимую комбинацию связей. Такой комбинацией является 2→1, 4→3. На рис. 8.1, д представлена диаграмма выполнения операторов при уточненных ранних сроках окончания их выполнения $\tau_{11}^{(1)}=3, \tau_{12}^{(1)}=\tau_{14}^{(1)}=2$, $\tau_{13}^{(1)}=4, \tau_{15}^{(1)}=\tau_{17}^{(1)}=6, \tau_{16}^{(1)}=5$.

Вновь находим значение $t_2=3$, при котором $F(3,2,4,2,6,5,6,3)=3>2$. Это значение так же как и ранее определяется выполнением операторов 3, 5, 6. Вновь формируем матрицу L_2 и находим первую из допустимых связей – связь 3→5. На рис. 8.1, ж представлена диаграмма выполнения алгоритма, удовлетворяющая условиям задачи.

8.6 Определение плана реализации алгоритма за минимальное время

В основе алгоритма решения задачи 2 лежит та же идея, что и при решении задачи 1. Для того чтобы T было минимальным временем выполнения алгоритма, представленного информационным графом $G=(X,P,\Gamma)$ на n процессорах, достаточно построить граф $G'=(X,P,\Gamma')$, для которого длина критического пу-

ти минимальна среди всех графов, полученных из заданного путем объединения вершин, соответствующих операторам каждого полного множества ВНО, содержащего $r > n$ операторов, $r - n$ ориентированными дугами в n путей, не содержащих общих вершин.

Действительно, пусть T – минимальное время, для которого удалось построить такой граф G' и $T = T'_{кр}$. Каждое полное множество ВНО в графе G' содержит не более n операторов. Следовательно, в соответствии с выводами раздела 7.3 алгоритм, представленный графом G', может быть выполнен за время $T'_{кр} = T$. Далее схема решения задачи 2 рассматривается детально.

В соответствии с методикой, изложенной в разделе 7.5, находят оценку $T = T_0$ минимального времени выполнения заданного алгоритма на n процессорах. Полученная оценка вполне может оказаться искомым минимальным временем выполнения алгоритма. Чтобы проверить это, целесообразно с помощью алгоритма, описанного в разделе 8.2, установить, способны ли n процессоров выполнить данный алгоритм за время T_0. Если нет, продолжаем решение задачи 2.

Для этого полагаем $\nu = 1,\ \tau_{1j}^{(\nu)} = \tau_{1j},\ j = \overline{1,m},\ \mu = 1,\ T_\mu = \infty$, где ∞ – заведомо большое число, и находим наименьшее значение $t_\nu = \tau^{(\nu)}{}_{1j} - t_j,\ j \in \{1,...,m\}$ такое, что

$$F\left(\tau_{11}^{(\nu)}, \ldots, \tau_{1m}^{(\nu)}, t_\nu\right) = r_\nu > n. \qquad (8.2)$$

Если такого значения t_ν нет на первой же итерации (при $\nu = 1$), то $T_{кр}$ – решение задачи 2. Если такого значения нет и при $\nu > 1$, то увеличивают значение μ на единицу и полагают $T_\mu = max\left\{\tau_{11}^{(\nu)}, \ldots, \tau_{1m}^{(\nu)}\right\}$, т.е. длине критического пути в графе G', полученном из заданного путем объединения вершин каждого полного множества ВНО, содержащего $r > n$ операторов, $r - n$ ориентированными дугами в n путей, не содержащих общих вершин. В случае, когда оказывается, что $T_\mu = T_0 + 1$, то поскольку T_0 – не решение задачи, искать значение

$T < T_{\mu}$ нецелесообразно, т.е. найденное значение T_{μ} – решение задачи 2. Если это не так, не меняя значения ν, вводим очередную, еще не испытанную комбинацию $r_{\nu} - n$ связей так, чтобы длина критического пути в полученном графе $G^{(\nu)}$ была меньше T_{μ}. Если такая комбинация связей существует, значение ν увеличивают на единицу, уточняют для него новые значения ранних сроков окончания выполнения операторов и ищут наименьшее значение $t_{\nu} = \tau^{(\nu)}_{1j} - t_j$, удовлетворяющее неравенству (8.2).

Если значение t_{ν}, обеспечивающее выполнение неравенства (8.2), найдено, выделяют множество операторов $A_{\nu} = \{\alpha_{j_{\rho}}\}, \rho = 1, \ldots, r_{\nu}$, для которых $\tau^{(\nu)}_{1j_{\rho}} - t_{j_{\rho}} \le t_{\nu} < \tau^{(\nu)}_{1j_{\rho}}$. Множество A_{ν} является подмножеством некоторого полного множества ВНО. Далее вводят еще не испытанную комбинацию $r_{\nu} - n$ связей так, чтобы длина критического пути в полученном при этом графе $G^{(\nu)}$ была меньше T_{μ}. Если такая комбинация связей существует, значение ν увеличивают на единицу, уточняют для него новые значения ранних сроков окончания выполнения операторов и вновь ищут наименьшее значение t_{ν}, удовлетворяющее неравенству (8.2).

Если такой комбинации связей не существует или все они уже перебраны и при этом $\nu > 1$, уменьшают ν на единицу, вводят очередную не испытанную комбинацию и повторяют указанные выше действия. Если уже испытаны все комбинации связей при $\nu = 1$, то последнее значение T_{μ} определяет искомое $T = T_{\mu}$.

8.7 Пример задачи определения минимального времени реализации алгоритма

Для *n*=2 решим задачу для графа, представленного на рис. 8.2, а. На рис. 8.2, б представлена диаграмма выполнения алгоритма при ранних сроках окончания выполнения операторов.

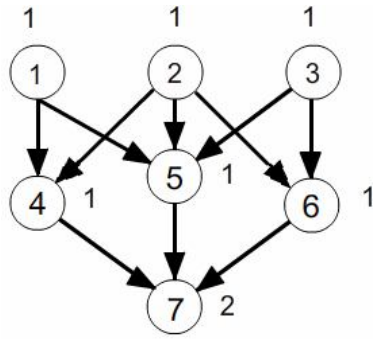
По алгоритму, описанному в разделе 7.5 найдем время T_1 :

$$T_1 = \max\left\{\left]\frac{1}{n}\sum_{i=1}^{7} t_i\right[, T_{кр}\right\} = \max\{4,4\} = 4.$$

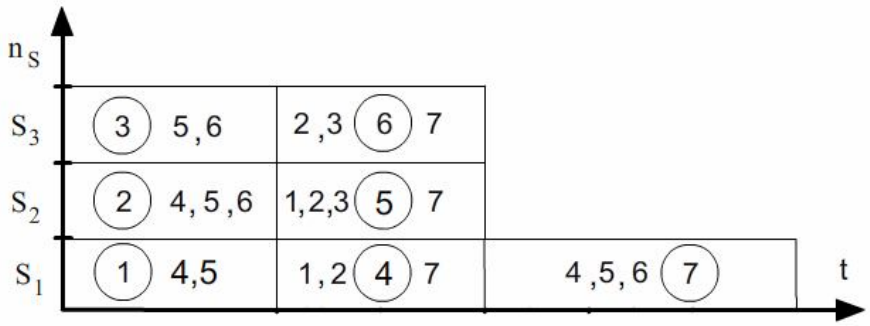
Найдем, что $\varphi^{(4)}(0,1)/1 = 3 > 2$. Уточним значение T_1 :

$$T_1 = 4 + \left]\frac{3-2}{2}\right[= 5.$$

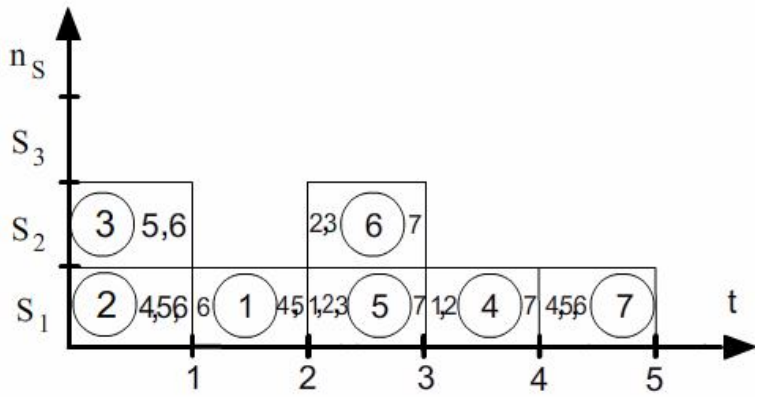
Для полученного T_1 проверим выполнение неравенства (7.7) на отрезках [0,2], [1,2], [0,3], [1,3], [2,3], [0,4], [1,4], [2,4], [3,4]. Это не приводит к необходимости дальнейшего уточнения значения T_1.



а)



б)



в)

Рис. 8.2 Решение задачи определения минимального времени реализации алгоритма

Применив алгоритм, описанный в разделе 8.2, найдем, что T=5 не является решением задачи 2. Начнем процесс последовательного сглаживания плот-

ности загрузки с момента времени t_1=0. Составляем матрицу L_1 для операторов 1, 2, 3. Возможные комбинации вводимой связи при v=1 в порядке их формирования 2→1, 3→1, 1→2, 3→2, 1→3, 2→3. При каждой из этих связей для момента времени t_2=2 приходится формировать матрицу L_2 для операторов 4, 5, 6, что приводит к перебору одного и того же множества связей: 5→4, 6→4, 4→5, 6→5, 4→6, 5→6. При всех комбинациях эти связи не меняют значения T_2=6, впервые полученного после введения первых связей 2→1 и 5→4 на соответственно первом и втором шагах сглаживания плотности загрузки.

Лекция 9

Простейшие параллельные алгоритмы

9.1 Вычисление суммы последовательности числовых значений

Простейшим и вместе с тем наиболее широко применяемым в разнообразных задачах является алгоритм вычисления суммы числовой последовательности:

$$S = \sum_{i=1}^{n} x_i \,, \tag{9.1}$$

где n – количество суммируемых значений.

Граф-схемы традиционных (линейного и каскадного) алгоритмов решения этой задачи в качестве примеров были приведены в разделе 3 на рис. 3.1 и 3.2 соответственно. Оценим необходимое для реализации этих схем число вычислительных операций.

Очевидно, что для реализации алгоритма последовательного суммирования необходимо $n-1$ операций. Общее количество операций суммирования в каскадной схеме такое же, как в последовательном алгоритме:

$$K_{посл} = \frac{n}{2} + \frac{n}{4} + ... + 1 = n - 1. \tag{9.2}$$

Если все операции на каждой итерации каскадной схемы выполняются параллельно, количество параллельных операций равно числу итераций k каскадной схемы:

$$K_{пар} = k = log_2 n \,. \tag{9.3}$$

Полагая время выполнения всех вычислительных операций одинаковым и равным τ, имеем $T_1 = K_{посл} \cdot \tau$, $T_s = K_{пар} \cdot \tau$ и с учетом (9.2), (9.3) получаем оценки ускорения и эффективности:

$$R = \frac{T_1}{T_s} = \frac{n-1}{log_2 n}, \tag{9.4}$$

$$E_s = \frac{n-1}{s \cdot \log_2 n} = \frac{2 \cdot (n-1)}{n \cdot \log_2 n}. \qquad (9.5)$$

Равенство (9.5) записано в предположении, что число процессоров, необходимых для реализации каскадной схемы, выбрано равным $s = n/2$. Нетрудно заметить, что эффективность каскадной схемы падает с ростом числа слагаемых:

$$\lim_{n \to \infty} p_s = 0.$$

Указанный недостаток преодолевается применением модифицированной каскадной схемы. Граф-схема соответствующего этой схеме алгоритма для случая $n = 2^k$, $k = 2^s$, s=2 приведена на рис. 9.1. Здесь цифрами 1-16 обозначены операции ввода, а цифрами 17-31 – операции суммирования. В этом варианте каскадной схемы вычисления проводятся в два этапа:

- на первом этапе все суммируемые значения подразделяются на $n / log_2 n$ групп по $log_2 n$ элементов в каждой группе, вычисления внутри группы выполняются последовательно, а вычисления для групп осуществляются параллельно на $s = n / \log_2 n$ процессорах;
- на втором этапе к полученным $n / \log_2 n$ суммам применяется каскадная схема.

На первом этапе требуется $log_2 n$ операций (при использовании $s = n / log_2 n$ процессоров). Для выполнения второго этапа необходимо

$$log_2 (n / log_2 n) \le log_2 n \qquad (9.6)$$

параллельных операций, выполняемых на

$$s_2 = (n / log_2 n) / 2$$

процессорах. С учетом неравенства (9.6) для описанной схемы при

$$s = n / log_2 n \qquad (9.7)$$

имеем

$$T_s \cong 2 \cdot log_2 n. \qquad (9.8)$$

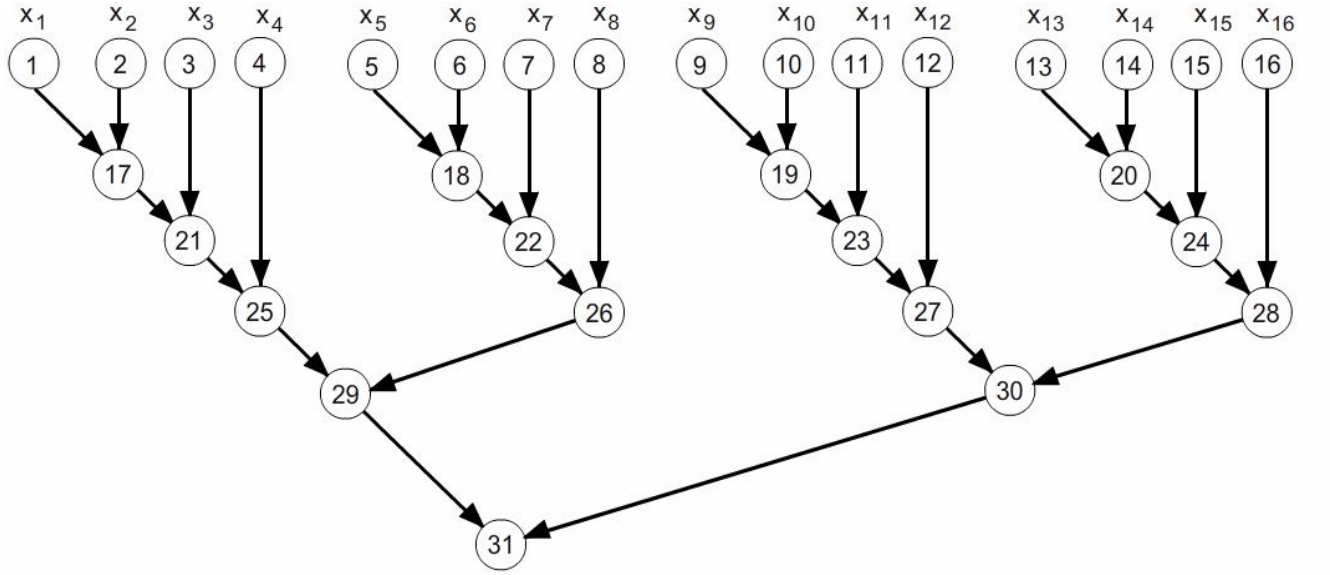


Рис. 9.1 Модифицированная каскадная схема суммирования

С учетом приведенных оценок (9.7), (9.8) показатели ускорения и эффективности модифицированной каскадной схемы определятся соотношениями

$$R = \frac{T_1}{T_s} = \frac{n-1}{2 \cdot log_2 n}, \tag{9.9}$$

$$E_s = \frac{R}{s} = \frac{n-1}{2 \cdot \log_2 n \cdot \left(n / \log_2 n\right) \cdot} = \frac{n-1}{2 \cdot n}. \tag{9.10}$$

Сравнивая оценки (9.9), (9.10) с показателями обычной каскадной схемы (9.4), (9.5), нетрудно заметить, что ускорение в данном случае уменьшилось в 2 раза, зато имеет место ненулевая оценка снизу для эффективности:

$$\lim_{n\to\infty} E_s = \lim_{n\to\infty} \frac{n-1}{2 \cdot n} = 0{,}5.$$

В отличие от обычной каскадной схемы, модифицированный каскадный алгоритм является *оптимальным по стоимости*, поскольку вычислительные затраты в данном случае определяются как

$$C_s = sT_s = \left(n / log_2 n\right)\left(2 log_2 n\right) = 2n,$$

т.е. пропорциональны времени выполнения последовательного алгоритма.

9.2 Задача вычисления всех частных сумм

Соотношение для вычисления всех частных сумм имеет вид

$$S_k = \sum_{i=1}^{n} x_i, \quad 1 \le k \le n, \tag{9.11}$$

где n – количество суммируемых значений. Для последовательного алгоритма вычисления всех частных сумм (9.11) потребуется

$$T_1 = n$$

операций.

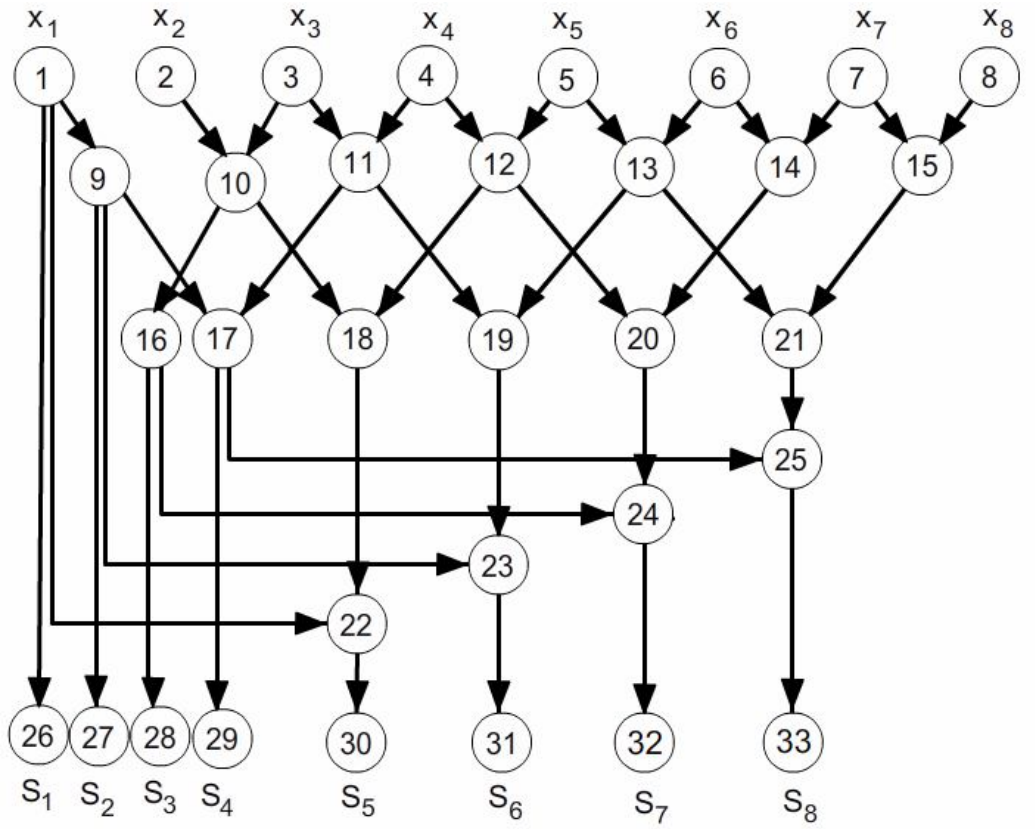
- Одна из возможных граф-схем параллельного алгоритма нахождения всех частных сумм, обеспечивающего получение всех результатов за $log_2 n$ операций, приведена на рис. 9.2. Здесь цифрами 1-8 обозначены операции ввода, цифрами 9-25 – операции суммирования, а 26-31 – операции вывода результатов S_i, $i = \overline{1,8}$. Необходимое количество процессоров для получения результатов за $log_2 n$ шагов определяется количеством суммируемых значений (s=n).

Показатели ускорения и эффективности для описанного параллельного алгоритма оцениваются следующим образом:

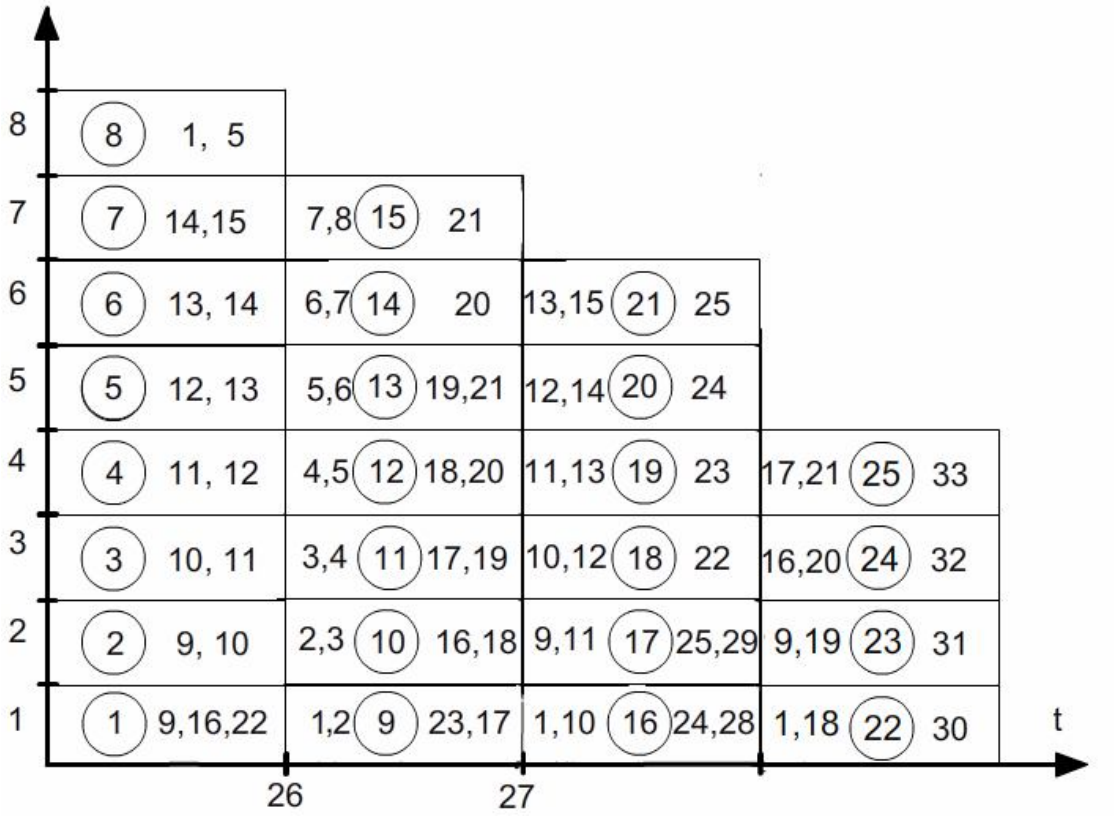
$$R = \frac{T_1}{T_s} = \frac{n}{log_2 n}, \tag{9.12}$$

$$E_s = \frac{R}{s} = \frac{n}{s \cdot \log_2 n} = \frac{n}{n \cdot \log_2 n} = \frac{1}{\log_2 n}. \tag{9.13}$$

Нетрудно заметить, что эффективность (9.13) алгоритма вычисления всех частных сумм быстро падает при увеличении числа суммируемых значений. Это говорит о невысокой степени внутреннего параллелизма в этой задаче.



а)



б)

Рис. 9.2 Граф-схема параллельного алгоритма (а) и временная диаграмма (б) вычисления всех частных сумм

9.3 Умножение матрицы на вектор, способы декомпозиции

Операция умножения матрицы на вектор – одна из самых часто встречающихся в самых различных задачах. Поэтому многие стандартные библиотеки программ содержат процедуры для выполнения матричных операций. Тем не менее изучение основных схем их распараллеливания полезно, т.к. во-первых, это дает необходимые сведения для обоснованного выбора наиболее подходящей для системы с данной топологией процедуры из имеющегося набора стандартных, во-вторых, матричные операции являются классическими примерами для демонстрации многих приемов и методов распараллеливания задач. Далее для общности будем полагать, что матрицы являются плотными, т.е. число нулевых элементов в них мало по сравнению с общим количеством элементов матриц.

При распараллеливании задачи умножения матрицы на вектор обычно используются два типа декомпозиции, показанные на рис. 5.3:

1. Ленточное разбиение – на полосы по горизонтали или вертикали.
2. Разбиение данных на прямоугольные фрагменты (блочное разделение).

При ленточном разбиении каждому процессору выделяется некоторое количество (обычно подряд идущих) строк или столбцов. При этом в случае горизонтального разбиения матрица A представляется в виде

$$A = (A_0, A_1, ..., A_{s-1})^T, \qquad A_i = (a_{i0}, a_{i1}, ..., a_{ik-1}),$$

$$ij = ik + j, 0 \le j < k, k = m / s,$$

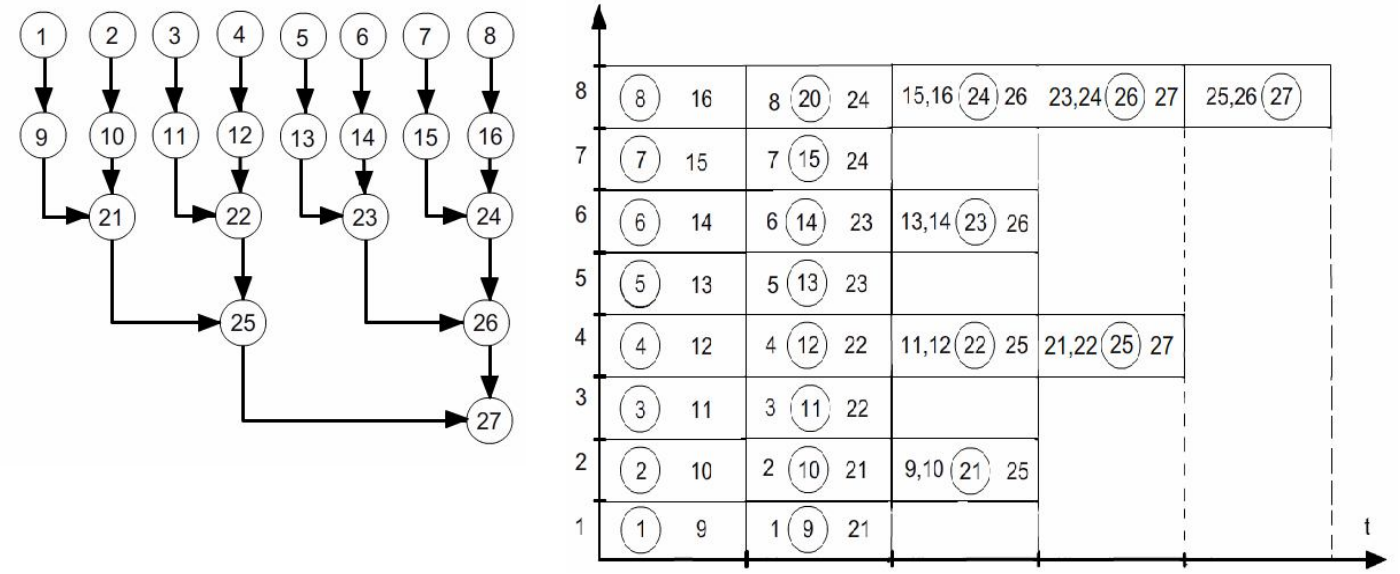
где $a_i = (a_{i,1}, a_{i,2}, ..., a_{i,n}), 0 \le i \le m$ – i-я строка матрицы A (предполагается, что количество строк m кратно числу процессоров: $m=k \cdot s$). Иногда с точки зрения балансировки процессоров для конкретной топологии более предпочтительным является циклическое чередование строк или столбцов. В этом случае

$$ij = i + js, 0 \le j < k, k = m / s.$$

На рис. 9.3 в качестве примера приведены граф-схема и временная диаграмма параллельного алгоритма умножения матрицы на вектор на 8 процессо-

рах при ленточном разбиении по строкам. Здесь цифрами 1-8 обозначены операции ввода данных, цифрами 9-16 – операции умножения выделенных каждому процессору полос на вектор. В результате выполнения этих операций на каждом процессоре будет получена 1/8 часть искомого вектора. Цифрами 21-27 обозначены операции «сборки». Сборка осуществляется за три шага. После первого шага (выполнения операций 21-24) на 4 процессорах окажется по 1/4 части искомого вектора. На следующем шаге (операции 25-26) на двух процессорах будет сформировано по 1/2 части искомого вектора. Наконец на завершающем 3-м шаге в результате выполнения операции 27 будет получен результирующий вектор.

При ленточном разбиении по столбцам схема формирования результирующего вектора существенно отличается. Сборке результирующего вектора в этом случае предшествует формирование промежуточных векторов. Соответствующий пример будет приведен в разделе 9.5.



а) б)

Рис. 9.3 Граф алгоритма (а) и временная диаграмма (б) при разделении матрицы по строкам

При блочной декомпозиции матрица делится на прямоугольные фрагменты обычно из подряд идущих элементов. Если количество процессоров $s = p \cdot q$, количество строк матрицы кратно p, а количество столбцов – q ($m = k \cdot p$,

$n = l \cdot q$), матрицу A можно представить в виде набора прямоугольных блоков следующим образом:

$$A = \begin{bmatrix} A(1,1) & A(1,2)... & A(1,q) \\ .... & ... & ... \\ A(p,1) & A(p,2)... & A(p,q) \end{bmatrix},$$

где $A(i,j)$ — блок матрицы, состоящий из элементов:

$$A(i,j) = \begin{bmatrix} a_{1,1}(i,j) & a_{1,2}(i,j)... & a_{1,l}(i,j) \\ .... & ... & ... \\ a_{K,1}(i,j) & a_{K,2}(i,j)... & a_{K,l}(i,j) \end{bmatrix}.$$

При блочном разбиении целесообразна топология системы в виде решетки из p строк и q столбцов. При этом вычислительный процесс следует организовать так, чтобы соседние в структуре решетки процессоры обрабатывали смежные блоки исходной матрицы. Для блочной схемы может быть применено также циклическое чередование строк и столбцов.

9.4 Умножение матрицы на вектор, разделение по строкам

В данном случае умножение $m \times n$-матрицы на $n \times 1$-вектор сводится к вычислению m скалярных произведений векторов длины n. Каждое такое произведение требует n операций умножения и $n-1$ операций сложения. Таким образом, вычислительная сложность последовательного алгоритма составит $T_1 = m(2n-1)$, а в случае квадратной $n \times n$-матрицы

$$T_1 = n(2n-1). \qquad (9.14)$$

Если умножение $n \times n$-матрицы выполняется параллельно (см. рис. 9.3), за каждым процессором закрепляется не более

$$m_s =]n/s[\qquad (9.15)$$

строк, где $]*[$ – здесь и далее означает операцию округления до целого в большую сторону. Количество процессоров, за которыми будет закреплено меньше чем m_s строк, определяется конкретными значениями n и s.

Для построения оценок ускорения и эффективности с учетом затрат на вычисления и коммуникации предполагаем, что все $(2n-1)$ операций умножения и сложения имеют одинаковую длительность τ, а вычислительная система однородна, т.е. все процессоры обладают одинаковой производительностью. Тогда временные затраты параллельного алгоритма, связанные непосредственно с вычислениями, с учетом (9.15) составят

$$T_s = \left\lceil \frac{n}{s} \right\rceil (2n-1)\cdot\tau . \qquad (9.16)$$

Если сеть передачи данных имеет структуру гиперкуба или полного графа, операция сбора данных может быть выполнена за $\lceil log_2 s \rceil$ итераций. На первой итерации взаимодействующие пары процессоров обмениваются сообщениями объемом $w(n/s)$, где w – размер одного элемента вектора в байтах. На второй итерации этот объем увеличивается вдвое и оказывается равным $2w\lceil n/s \rceil$ и т.д. Общая длительность сбора данных

$$T_s = \sum_{i=1}^{\lceil \log_2 s \rceil} \left(\alpha + 2^{i-1} w\right) \lceil n/s \rceil / \beta = \alpha \lceil \log_2 s \rceil + w \lceil n/s \rceil \left(\sum_{i=1}^{\lceil \log_2 s \rceil} 2^{i-1} \right) / \beta , \qquad (9.17)$$

где α– латентность сети передачи данных, β – пропускная способность сети. Можно показать, что

$$\sum_{i=1}^{\lceil \log_2 s \rceil} 2^{i-1} = \left(2^{\lceil \log_2 s \rceil} - 1\right) = s - 1 . \qquad (9.18)$$

С учетом (9.16), (9.17) и (9.18) общее время выполнения параллельного алгоритма

$$T_s = \left\lceil \frac{n}{s} \right\rceil (2n-1)\cdot\tau + \alpha \lceil \log_2 s \rceil + w \lceil n/s \rceil (s-1)/\beta . \qquad (9.19)$$

Если n/s и $log_2 s$ целые числа, в соответствии с (9.19) оценки для ускорения и эффективности принимают вид

$$R = \frac{T_1}{T_s} = \frac{n(2n-1)\cdot\tau}{(n/s)(2n-1)\cdot\tau + \alpha(log_2 s) + w(n/s)(s-1)/\beta} , \qquad (9.20)$$

$$E_s = \frac{R}{s} = \frac{n \cdot (2n-1) \cdot \tau}{n \cdot (2n-1) \cdot \tau + s\{\alpha(\log_2 s) + w(n/s)(s-1)/\beta\}}. \qquad (9.21)$$

Из соотношений (9.21) видно, что для сохранения требуемого уровня эффективности при увеличении числа используемых процессоров соответственно должна возрастать вычислительная сложность задачи (размерности матрицы и вектора). Заметим также, что если в (9.20), (9.21) пренебречь затратами на передачу данных, будем иметь

$$R = T_1 / T_s = s,$$

$$E_s = R/s = 1.$$

Высокий внутренний параллелизм задачи связан с тем, что в данном случае отсутствуют операции, которые не поддаются распараллеливанию.

9.5 Умножение матрицы на вектор, разделение по столбцам

При параллельном умножении матрицы на вектор с разделением данных по столбцам каждый столбец матрицы *A* умножается на один (соответствующий ему) элемент вектора *b*. Если за процессором закреплена полоса, элементы столбцов, из которых она составлена, умножаются на соответствующие элементы вектора *b*, которые также должны быть закреплены за этим процессором. После умножения полученные значения суммируются для каждой *m*-й строки матрицы *A* в отдельности:

$$c'_m(i) = \sum_{j=j_0}^{j_l - 1} a_{mj} b_j, \quad 0 \le m < n,$$

где (j_0 и $j_{l\text{-}1}$ – начальный и конечный индексы столбцов подзадачи i, $0 \le i < n$).

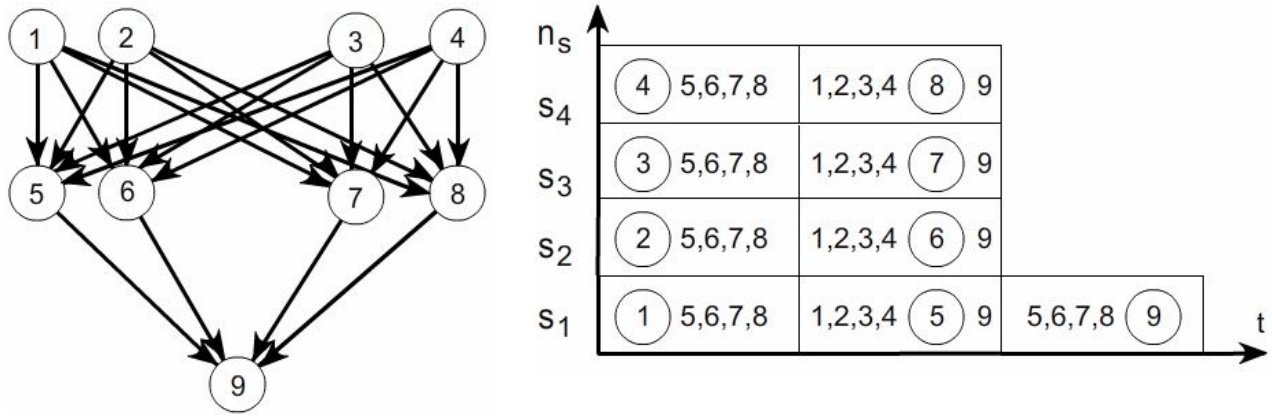
Вычислительные затраты для вычисления одного элемента промежуточного вектора на этом – первом этапе могут быть оценены как

$$2 \cdot]n/s[- 1. \qquad (9.22)$$

В результате реализации первого этапа на каждом процессоре получается столбец промежуточных значений.

Для получения элементов результирующего вектора (второй этап) подзадачи должны обменяться этими промежуточными данными между собой. Притом процессор-отправитель должен отослать на процессор-получатель только ту часть промежуточного вектора (]*m*/*s*[), за вычисление которой в результирующем векторе «отвечает» процессор-получатель. После обмена данными между подзадачами каждый процессор суммирует полученные значения для своего блока результирующего вектора. Завершающая операция – «сборка» результирующего вектора.

На рис. 9.4 в качестве примера приведены граф-схема и временная диаграмма умножения матрицы на вектор при разбиении матрицы по столбцам на 4 процессора. Здесь цифрами 1-4 обозначены операции умножения столбца или выделенной полосы (операции ввода данных для простоты не показаны) на соответствующий элемент (фрагмент) вектора, цифрами 5-8 – операции формирования (пересылка и суммирование) закрепленных за каждым процессором фрагментов искомого вектора, а операция 9 – завершающий этап «сборки» результирующего вектора.



а) б)

Рис. 9.4 Граф алгоритма (а) и временная диаграмма (б) вычислений при разделении матрицы по столбцам на 4 процессора

С учетом сказанного общее время выполнения вычислительных операций параллельного алгоритма на первом и втором этапах вместе составит:

$$T_s = \left]m \cdot \left(2 \cdot \right]n/s\left[-1\right) + \left(s-1\right)\right]m/s\left[\right[\cdot \tau \,. \qquad (9.23)$$

Заметим, что здесь в соответствии с (9.22)

$$m \cdot \left(2 \cdot \right]n/s\left[-1\right)$$

– оценка максимального числа операций (умножения и сложения) для вычисления всех элементов одного промежуточного $m\times 1$-вектора.

Количество суммируемых значений для каждого элемента результирующего вектора на втором этапе равно числу промежуточных векторов, т.е. совпадает с числом процессоров s. Максимальный размер блока результирующего вектора, закрепленный за одним процессором-получателем в общем случае равен $]m/s[$. Следовательно, оценка числа операций на этом (втором) этапе равна $s]m/s[$.

Оценим теперь ускорение и эффективность параллельного алгоритма с учетом затрат времени на пересылку данных. Временные затраты на передачу данных, необходимых процессору-получателю для вычисления одного блока результирующего вектора составят

$$\left(\alpha + w\right]m/s\left[/\beta\right), \qquad (9.24)$$

где α – латентность сети передачи данных, β – пропускная способность сети, w – размер элемента данных в байтах.

Общие временные затраты на коммуникации зависят от топологии сети. Если имеется возможность одновременно отправлять и принимать сообщения между любой парой процессоров, тогда временные затраты с учетом (9.24) составят

$$T_s^1 = \left(s-1\right)\left(\alpha + w\right]m/s\left[/\beta\right). \qquad (9.25)$$

Если топология вычислительной системы представлена в виде гиперкуба, передача всех данных может быть выполнена за $log_2\, s$ шагов, на каждом из которых максимально возможная (т.к. в общем случае размеры закрепляемых за процессором блоков различаются) длина передаваемых и получаемых сообщений $]m/s[$ элементов

$$T_s^2 =]\log_2 s\left[\left(\alpha + w\right]m/s\left[/\beta\right). \qquad (9.26)$$

К временным затратам, определяемым соотношениями (9.25), (9.26), следует еще добавить время, необходимое на окончательную «сборку» результирующего вектора:

$$(s-1)(\alpha + w]m/s[/\beta).$$

С учетом этих затрат общее время выполнения параллельного алгоритма для указанных способов передачи данных соответственно

$$T_s^1 =]m \cdot (2 \cdot]n/s[-1) + (s-1)]m/s[[\cdot \tau + 2 \cdot (s-1)(\alpha + 2 \cdot w]m/s[/\beta),$$

$$T_s^2 = [m \cdot (2 \cdot [n/s] - 1) + s[m/s]] \cdot \tau + (s - 1 + [log_2 s])(\alpha + 2 \cdot w[m/s]/\beta).$$

Если матрица квадратная размерности $n \times n$, а n кратно числу процессоров, т.е. n/s – целое число, ускорение и эффективность для первого типа топологии сети соответственно

$$R = \frac{n \cdot (2n-1) \cdot \tau}{\frac{n(2 \cdot n-1)}{s} \cdot \tau + 2 \cdot (s-1)(\alpha + 2 \cdot w(n/s)/\beta)}, \qquad (9.27)$$

$$E_s = \frac{n \cdot (2n-1) \cdot \tau}{n(2 \cdot n-1) \cdot \tau + 2 \cdot s(s-1)(\alpha + 2 \cdot w(n/s)/\beta)}. \qquad (9.28)$$

Для второго типа топологии сети имеем соответственно

$$R = \frac{n \cdot (2n-1) \cdot \tau}{\frac{n(2 \cdot n-1)}{s} \cdot \tau + (s-1+]\log_2 s[)(\alpha + 2 \cdot w(n/s)/\beta)}, \qquad (9.29)$$

$$E_s = \frac{n \cdot (2n-1) \cdot \tau}{n(2 \cdot n-1) \cdot \tau + s(s-1+]\log_2 s[)(\alpha + 2 \cdot w(n/s)/\beta)}. \qquad (9.30)$$

В данном случае также отсутствуют операции, которые могут выполняться только последовательно, поэтому без учета потерь на коммуникации (9.25), (9.26) максимально возможные теоретические значения ускорения и эффективности в соответствии с (9.29), (9.30) равны соответственно $R=s$, $E_s=1$.

9.6 Умножение матрицы на вектор при блочном разделении данных

Общее время умножения блоков матрицы A размером $(n/p)\times(n/q)$ на соответствующие блоки вектора b можно записать, используя полученные ранее формулы для ленточного разбиения по строкам и столбцам:

$$T_s = \lceil n/p \rceil \cdot \left(2 \cdot \lceil n/q \rceil - 1\right) \cdot \tau .$$

Здесь $\left(2 \cdot \lceil n/q \rceil - 1\right)$ - количество операций сложения и умножения в одной строке блока при разбиении на n/q столбцов, а n/p - количество строк в каждом блоке.

С учетом последующего суммирования промежуточных векторов вычислительные затраты увеличатся. Объединение промежуточных результатов может быть выполнено с использованием каскадной схемы за $log_2 q$ итераций. Приближенно эти затраты можно учесть, добавив слагаемое $\tau \ log_2 q$. При этом оценка общих потерь с учетом потерь на коммуникации составит

$$T_s = \lceil n/p \rceil \cdot \left(2 \cdot \lceil n/q \rceil - 1\right) \cdot \tau + \lceil \log_2 q \rceil \left(\tau + \alpha + w(n/p)/\beta\right).$$

Если количество строк матрицы кратно p, а количество столбцов – кратно q, т. е. $s=p\cdot q$, $m=k\cdot p$ и $n=k\cdot$q, выражения для ускорения и эффективности можно записать в виде

$$R = \frac{n(2n-1)\cdot\tau}{(n/p)\cdot\left(2\cdot(n/q)-1\right)\cdot\tau + \lceil \log_2 q \rceil \left(\tau + \alpha + w(n/q)/\beta\right)}, \qquad (9.31)$$

$$E_s = \frac{n(2n-1)\cdot\tau}{n\cdot(2n-q)\cdot\tau + s\lceil \log_2 q \rceil \left(\tau + \alpha + w(n/q)/\beta\right)}. \qquad (9.32)$$

Напомним, что здесь $s=p\cdot q$.

В заключение укажем на возможный способ выбора блочной структуры матрицы A. В данном случае увеличение числа блоков по горизонтали приво-

дит к возрастанию числа итераций в операции суммирования промежуточных векторов, а увеличение числа блоков по вертикали позволяет снизить объем передаваемых данных между процессорами.

Если пренебречь затратами на телекоммуникации, то в соответствии с соотношениями (9.31), (9.32) ускорение равное s и эффективность 1 достигаются при

$$log_2 q = \frac{n(q-1)}{s} = \frac{n}{p}\frac{(q-1)}{q}.$$

Это соотношение может использоваться для подбора соотношения между p·и q, обеспечивающего соблюдение указанного свойства.

Лекция 10

Перемножение матриц

10.1 Последовательный алгоритм умножения матриц

Умножение матрицы A размера $m \times n$ на матрицу B размера $n \times l$ приводит к получению матрицы **C** размера $m \times l$, каждый элемент которой

$$c_{ij} - \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, \quad 0 \le i < m, \quad 0 \le j < l,$$

т.е. каждый элемент матрицы C вычисляется как скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = \left(a_i, b_j^T\right),$$

$$a_i = \left(a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}\right),$$

$$b_j^T = \left(b_{i,0}, b_{i,1}, \ldots, b_{i,n-1}\right)^T$$

Легко подсчитать, что для реализации перемножения матриц потребуется $m \times n \times l$: операций умножения и $m \times (n\text{-}1) \times l$ операций сложения. Существуют приемы, позволяющие перемножить матрицы за меньшее число операций. Однако мы ограничимся рассмотрением алгоритма, реализующего указанный простейший случай. Подсчитаем общее число операций последовательного алгоритма для перемножения квадратных $n \times n$-матриц.

Для вычисления одного элемента потребуется $2n-1$ операций (n операций умножения и $n-1$ операций сложения); для вычисления одной строки – $n \cdot (2n-1)$ операций, а общее число операций сложения и умножения для получения результирующей матрицы C в последовательном алгоритме составит

$$N_{A*B} = n^2 \cdot (2n-1). \qquad (10.1)$$

Далее рассматриваются два возможных параллельных алгоритма перемножения матриц при ленточном разбиении и декомпозиции на блоки.

10.2. Параллельные алгоритмы при ленточном разбиении матрицы

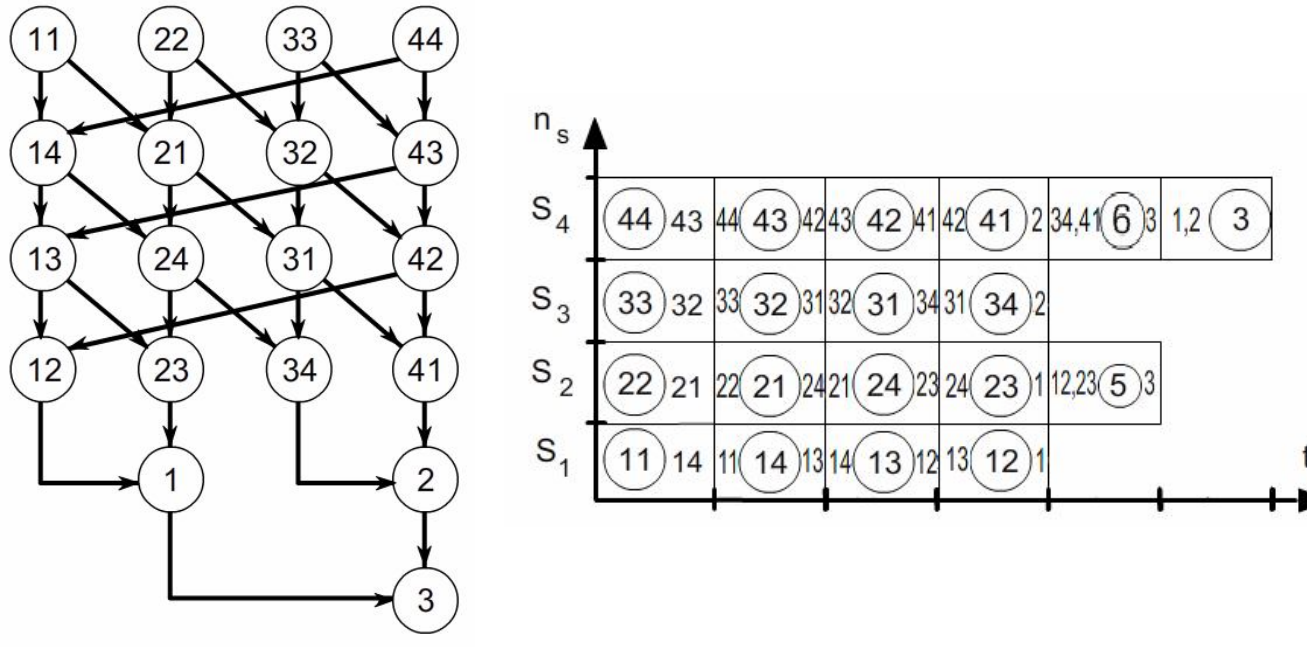
Рассмотрим два параллельных алгоритма умножения матриц с ленточным разбиением на строки или столбцы (полосы). Если каждая подзадача содержит по одной строке матрицы A и одному столбцу матрицы B, общее число подзадач (необходимых процессоров) равно n^2. При большой размерности матриц это может быть неприемлемо. Чаще декомпозицию осуществляют таким образом, что подзадача заключается в вычислении элементов одной строки матрицы C. При этом количество подзадач (потребных процессоров) равно n. Далее рассмотрим именно этот случай.

Для выполнения всех вычислений в подзадаче процессору должны быть доступны одна из строк матрицы A и все столбцы матрицы B. Если дублирование матрицы B во всех подзадачах неприемлемо в силу большой размерности матрицы, необходимо строить итерационный процесс с обменом данными между процессорами. Возможны две схемы организации вычислений с обменом данными.

Первый алгоритм. На каждой итерации каждая подзадача содержит по одной строке матрицы A и одному столбцу матрицы B. При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк и столбцов, что приводит к получению соответствующих элементов результирующей матрицы C. В конце каждой итерации столбцы матрицы B передаются между подзадачами. Процесс передачи организуется так, чтобы после завершения итераций в каждой подзадаче последовательно «побывали» все столбцы матрицы B.

Для указанной схемы передачи данных наиболее подходящей является топология связей подзадач в виде кольцевой структуры. При этом на каждой итерации подзадача i, $0 \le i < n$ передает свой столбец матрицы B подзадаче с номером $i+1$, а подзадача n-1 передает свои данные подзадаче с номером 0. На рис. 10.1 приведены граф-схема и временная диаграмма алгоритма перемножения матриц для случая, когда матрицы состоят из четырех строк и четырех столбцов (n=4).

Для наглядности здесь для операций вычисления произведений строк на столбцы принята нумерация в виде двух цифр. Первая цифра обозначает номер строки матрицы A, а вторая – номер столбца матрицы B в произведении. Цифрами 1, 2 и 3 обозначены операции «сборки» результатов. В частности, оператором 1 осуществляется сборка части искомой матрицы, включающей первую и вторую строку, а оператором 2 – объединение третьей и четвертой строк. Операция 3 – реализует завершающий этап «сборки» всей результирующей матрицы.



а) б)

Рис. 10.1 Граф-схема (а) и временная диаграмма (б)
первого алгоритма перемножения матриц (блоки по строкам)

Второй алгоритм. Отличие в том, что в подзадачах располагаются не столбцы, а строки матрицы *B*. А перемножение матриц сводится к умножению строк матрицы *B* на соответствующие элементы строк матрицы *A* и последующему сложению получающихся строк.

Например, в случае, когда за каждым процессором закреплена одна строка матрицы *A*, для получения первой строки матрицы ***C*** каждая, например, *i*-я строка матрицы *B* умножается на соответствующий элемент, в данном случае

i-й, первой строки матрицы *A*, а затем все полученные промежуточные векторы (*i*=1,2,..,*n*) складываются. Для получения второй строки матрицы *C* все строки матрицы *B* умножаются на соответствующие элементы второй строки матрицы *A* и т.д.

В этой схеме вычислений после каждого шага умножения очередного элемента матрицы *A* на соответствующую строку матрицы *B* в каждой подзадаче (на каждом процессоре) получается строка частичных результатов, которые поэлементно суммируются со своей строкой матрицы *C*. Для выполнения следующего шага умножения строк матрицы *B* на элементы матрицы *A* необходимо передать во все подзадачи соответствующие строки матрицы *B*. Передача строк матрицы *B* между подзадачами в данном случае может быть выполнена с использованием кольцевой структуры информационных связей, т.е. наиболее подходящей является топология «кольцо».

До начала вычислений в каждой подзадаче *i*, $0 \le i < n$ располагаются *i*-е строки матрицы *A* и матрицы *B*. В результате их перемножения подзадача определяет *i*-ю строку частичных результатов искомой матрицы *C*. Далее подзадачи осуществляют обмен строками, в ходе которого каждая подзадача передает свою строку матрицы *B* следующей подзадаче до завершения всех итераций параллельного алгоритма.

Для случая, когда матрицы состоят из четырех строк и четырех столбцов (*n*=4), граф-схема алгоритма и временная диаграмма могут быть представлены в точно таком же виде, как это показано на рис. 10.1. Для этого достаточно придать другое содержание обозначениям операций. В частности, теперь первая цифра в обозначении операции перемножения должна служить для обозначения столбца матрицы A, а вторая – строки матрицы B.

Цифрой 1 теперь обозначена операция объединения первого и второго столбца, цифрой 2 – операция объединения третьего и четвертого столбца, а цифрой 3 – операция «сборки» всех столбцов искомой матрицы.

Если размерность матрицы n оказывается больше, чем число процессоров s, подзадачи можно укрупнить, объединив несколько соседних строк и столбцов перемножаемых матриц. При этом исходная матрица A разбивается на ряд горизонтальных полос, а матрица B представляется в виде набора вертикальных (для первого алгоритма) или горизонтальных (для второго алгоритма) полос. Если n кратно s, размер полос $k=n/s$. Распределение подзадач между процессорами следует осуществлять таким образом, чтобы подзадачи, являющиеся соседними в кольцевой топологии, располагались на процессорах, между которыми имеются прямые линии передачи данных.

10.3 Анализ эффективности алгоритмов при ленточном разбиении матриц

Проведем теперь анализ эффективности описанных выше двух алгоритмов. Время выполнения последовательного алгоритма

$$T_1 = n^2 \cdot (2n-1) \cdot \tau, \qquad (10.2)$$

где τ – время выполнения одной элементарной скалярной операции. Выпишем соотношения для подсчета вычислительных затрат при параллельной реализации алгоритма.

Первый вариант параллельного алгоритма – это наиболее часто используемая, обычная схема вычислений, при которой, как уже указывалось выше, для вычисления одного элемента необходимо $2n-1$ операций (n операций умножения и $n-1$ операций сложения). А для вычисления всех элементов потребуется $n^2 \cdot (2n-1)$ операций.

Во втором варианте потребуется n^2 умножений и $n(n-1)$ сложений в каждой строке. Для вычисления всех n строк матрицы C соответственно потребуется $n\left(n^2 + n(n-1)\right) = n^2(2n-1)$, т.е. вычислительная сложность обоих вариантов одинакова. Различия в ускорении и эффективности вариантов могут иметь место лишь за счет разных затрат на коммуникации. Построим эти оценки.

Если n/s - целое число, время выполнения вычислений параллельного алгоритма, с учетом приведенных выше соотношений, составит

$$T_s = \left(n^2 / s\right) \cdot \left(2n - 1\right) \cdot \tau, \qquad (10.3)$$

где, как и ранее, τ – время выполнения одной скалярной операции.

Будем предполагать, что все операции передачи данных между процессорами, соединенными в «кольцо», в ходе одной итерации алгоритма могут быть выполнены параллельно. Объем передаваемых данных между процессорами определяется размером полос и составляет n/s строк или столбцов длины n. Общее количество параллельных операций передачи сообщений на единицу меньше числа итераций алгоритма (на последней итерации передача данных не является обязательной). Таким образом, оценка затрат на передачу определяется как

$$T_s = \left(s - 1\right) \cdot \left(\alpha + w \cdot n \cdot \left(n/s\right) / \beta\right), \qquad (10.4)$$

где α– латентность, β – пропускная способность сети передачи данных, а w – размер элемента матрицы в байтах.

С учетом полученных соотношений достижимое ускорение

$$R = \frac{n^2 \cdot \left(2n - 1\right) \cdot \tau}{\left(n^2 / s\right) \cdot \left(2n - 1\right) \cdot \tau + \left(s - 1\right) \cdot \left(\alpha + w \cdot n \cdot \left(n/s\right) / \beta\right)}, \qquad (10.5)$$

а эффективность

$$E_s = \frac{n^2 \cdot \left(2n - 1\right) \cdot \tau}{n^2 \cdot \left(2n - 1\right) \cdot \tau + s \cdot \left(s - 1\right) \cdot \left(\alpha + w \cdot n \cdot \left(n/s\right) / \beta\right)}. \qquad (10.6)$$

Из полученных соотношений видно, что для сохранения эффективности, увеличение числа процессоров оправдано только в случае соответственного роста вычислительной сложности задачи.

Тот факт, что в обоих алгоритмах отсутствуют операции, которые не могут быть распараллелены и могут выполняться только последовательно, проявляется в том, что вычисленное по указанным формулам максимально достижимое

ускорение и эффективность при отсутствии потерь на коммуникации составляет соответственно $R=s$, и $p_s=1$,

10.4 Умножение матриц при блочном разделении данных

При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Для простоты далее предполагается, что все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали одинаково и равно q (т.е. размер всех блоков равен $k \times k$, $k=n/q$, а n кратно q).

При указанных предположениях матрицу A можно представить в следующем виде:

$$A = \begin{bmatrix} A(1,1) & A(1,2) \ldots & A(1,q) \\ \ldots & \ldots & \\ A(q,1) & A(q,2) \ldots & A(q,q) \end{bmatrix}$$

где $A(i,j)$ — блок матрицы, состоящий из элементов:

$$A(i,j) = \begin{bmatrix} a_{1,1}(i,j) & a_{1,2}(i,j) \ldots & a_{1,k}(i,j) \\ \ldots & \ldots & \\ a_{k,1}(i,j) & a_{k,2}(i,j) \ldots & a_{k,k}(i,j) \end{bmatrix}$$

При таком же представлении матриц B и C блок C_{ij}, полученной в результате перемножения матриц A и B матрицы C, определяется как

$$C(i,j) = \sum_{r=1}^{q} A_{i,r} B_{r,j} \ .$$

При блочном разбиении данных в качестве подзадач естественно взять совокупности вычислительных операций, выполняемых над матричными блоками. Другими словами, определим подзадачу как процедуру вычисления всех элементов одного из блоков матрицы С. Для выполнения всех необходимых вычислений подзадаче должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B.

Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема требуемой памяти. Поэтому целесообразно вычисления организовать таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь необходимые для реализации текущего этапа данные. При этом для доступа к остальным данным понадобится передача данных между процессорами. Ниже рассматривается два возможных способа организации такой вычислительной схемы (алгоритм Фокса) [3].

В рассматриваемой (первой) вычислительной схеме в каждой подзадаче на каждой итерации расчетов располагается только по одному блоку исходных матриц A и B. Для нумерации подзадач используются индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока $C_{i,j}$ – тем самым набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

С учетом принятых обозначений в ходе вычислений каждой подзадаче (i,j) должны быть доступны следующие четыре матричных блока:

- блок C_{ij} матрицы C, вычисляемый подзадачей;
- блок A_{ij} матрицы A, размещаемый в подзадаче перед началом вычислений;
- блоки A'_{ij} , B'_{ij} матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

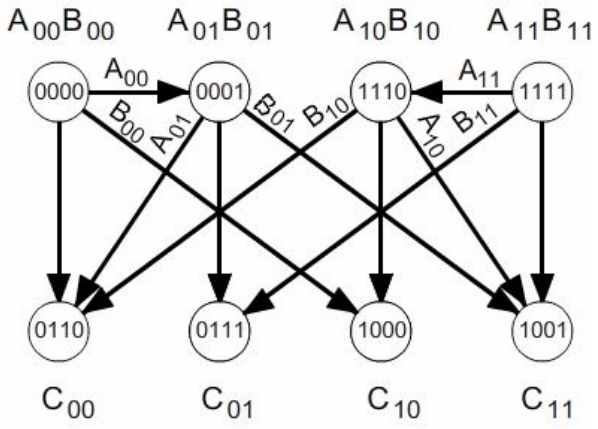
Выполнение параллельного алгоритма состоит из следующих шагов:

- этап инициализации, на котором каждой подзадаче (i,j) передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;
- этап вычислений, в рамках которого на каждой итерации l, $0 \le l < q$, осуществляются следующие операции:
- для каждой строки i, $0 \le i < q$, блок A_{ij} подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

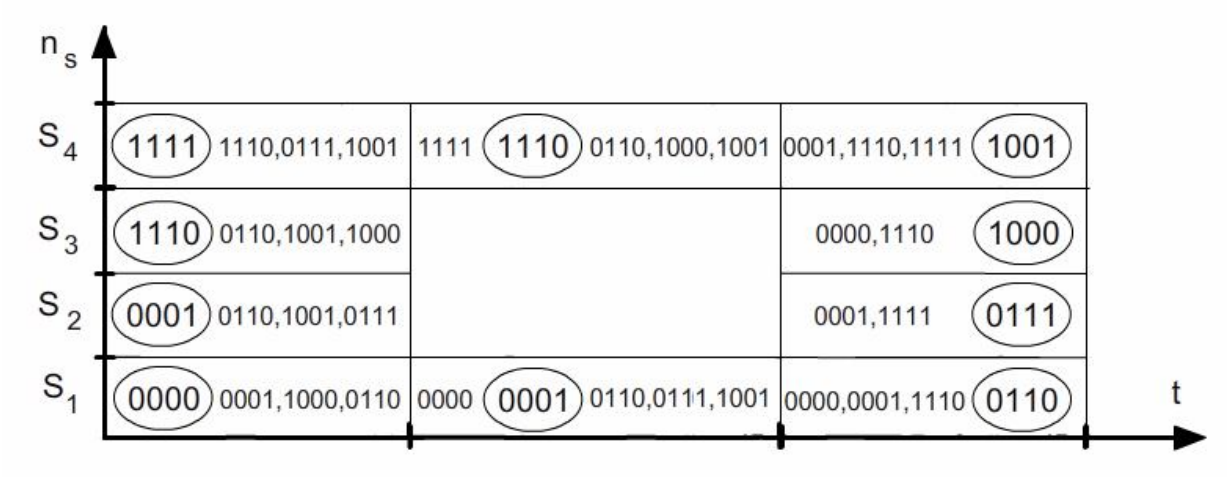
$j = (i + l) mod\, q$, где mod – операция получения остатка от целочисленного деления;

- полученные в результате пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij} ;
- блоки B'_{ij} каждой подзадачи (i,j) пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Для пояснения алгоритма на рис. 10.2 приведены граф-схема и временная диаграмма перемножения матриц при разбиении на блоки для решетки подзадач 2×2. Здесь для наглядности используются обозначения операций из 4 цифр. Первые две цифры являются индексами в обозначении участвующего в операции блока матрицы A, а вторые две цифры – индексами в обозначении участвующего в произведении блока матрицы B.



а)



б)

Рис. 10.2 Граф-схема (а) и временная диаграмма (б) алгоритма перемножения матриц для решетки подзадач 2×2

При разбиении матрицы на блоки следует, как обычно, стремиться к достижению высокой эффективности с учетом числа доступных процессоров. Например, в наиболее простом случае, когда число процессоров представимо в виде $s=\delta^2$, количество блоков в матрицах по вертикали и горизонтали целесообразно принять одинаковым и равным δ. При этом объем вычислений в каждой подзадаче будет одинаковым, и при одинаковой производительности процессоров будет достигаться полная балансировка вычислительной нагрузки между ними.

Заметим, что в описанном алгоритме в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач. Поэтому наиболее целесообразной в данном случае является топология сети вычислительной системы в виде решетки или полного графа.

Определим вычислительную сложность описанного алгоритма. Будем полагать, что все матрицы квадратные размера $n\times n$, количество блоков по горизонтали и вертикали одинаково и равно q. Таким образом, размер всех блоков – $k\times k$ ($k=n/q$) процессоры образуют квадратную решетку и их количество равно $p=q^2$. Для простоты полагаем также, что n/q и n/s – целые числа.

Для выполнения описанного алгоритма перемножения матриц требуется q итераций, в ходе которых каждый процессор перемножает свои текущие блоки

матриц A и B и прибавляет результаты умножения к текущему значению соответствующего блока матрицы C. При сделанных предположениях общее количество операций имеет порядок n^3/s. Поскольку в данном алгоритме отсутствуют вычисления, которые могут выполняться только последовательно, при отсутствии потерь на коммуникации могут достигаться показатели ускорения и эффективности s и 1 соответственно.

Подсчитаем теперь количество вычислительных операций параллельного алгоритма с учетом затрат на выполнение операций передачи данных между процессорами. Сложность выполнения скалярного умножения строки блока матрицы A на столбец блока матрицы B можно оценить как $2(n/q) - 1$. Количество строк и столбцов в блоках равно n/q. Следовательно, вычислительная сложность блочного умножения равна $(n^2/s)(2n/q - 1)$. Для сложения блоков требуется n^2/s операций. С учетом сказанного время выполнения всех операций алгоритма

$$T_s = q\left[\left(n^2/s\right)\cdot\left(2n/q - 1\right) + \left(n^2/s\right)\right]\cdot\tau\,, \qquad (10.7)$$

где τ – время выполнения одной скалярной операции.

Оценим теперь затраты на выполнение операций передачи данных между процессорами. На каждой итерации перед умножением блоков один из процессоров каждой строки решетки процессоров рассылает свой блок матрицы A остальным процессорам своей строки. При топологии сети в виде гиперкуба или полного графа выполнение этой операции может быть обеспечено за $log_2 q$ шагов, а объем передаваемых блоков равен n^2/p. При этом время выполнения операции передачи блоков матрицы A составит

$$T_s = log_2\, q\left(\alpha + w\left(n^2/s\right)/\beta\right), \qquad (10.8)$$

где α – латентность, β – пропускная способность сети передачи данных, а w – размер элемента матрицы в байтах.

Если топология строк решетки процессоров – кольцо, выражение для оценки времени передачи блоков матрицы A принимает вид

$$T^1_{s,comm} = (q/2)(\alpha + w(n^2/s)/\beta). \qquad (10.9)$$

После умножения матричных блоков процессоры передают свои блоки матрицы B предыдущим процессорам по столбцам решетки процессоров. При этом первые передают свои данные последним процессорам в столбцах решетки. Эти операции могут быть выполнены параллельно, поэтому время на передачу данных на этом этапе составит:

$$T^2_{s,comm} = \alpha + w(n^2/s)/\beta\,. \qquad (10.10)$$

Просуммировав соотношения (10.7), (10.8), (10.10), с учетом параметра q получаем общее время выполнения алгоритма:

$$\begin{aligned} T &= q\left[(n^2/s)\cdot(2n/q-1)+(n^2/s)\right]\cdot\tau + \\ &+ q\,log_2\,q\left(\alpha + w(n^2/s)/\beta\right) + (q-1)\cdot\left(\alpha + w(n^2/s)/\beta\right) = \\ &= q\left[(n^2/s)\cdot(2n/q-1)+(n^2/s)\right]\cdot\tau + (q\,log_2\,q + (q-1))(\alpha + w(n^2/s)/\beta). \end{aligned} \qquad (10.11)$$

Напомним, что параметр q определяет размер процессорной решетки и

$$q = \sqrt{s}\,.$$

Теперь с использованием (10.11) нетрудно записать соотношения для ускорения и эффективности, аналогичные (10.5), (10.6). Читателям предлагается выполнить это самостоятельно.

10.5 Модифицированный метод умножения матриц при блочном разделении данных

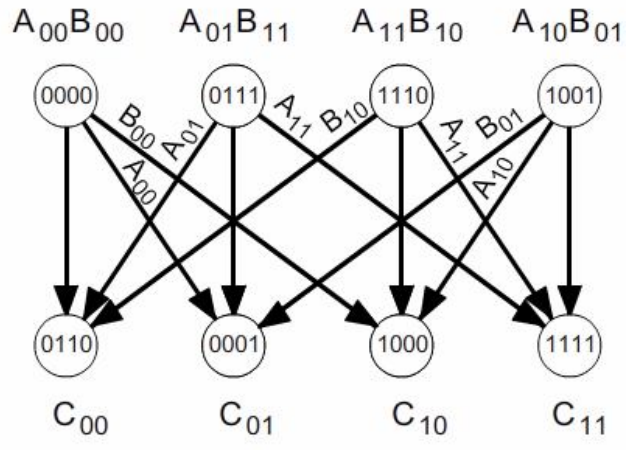
Для уменьшения затрат на коммуникации применяют модифицированный алгоритм умножения матриц при блочном разделении данных. Отличие от рассмотренного выше алгоритма заключается в изменении схемы начального распределения блоков перемножаемых матриц между подзадачами. Начальное расположение блоков подбирается так, чтобы блоки можно было перемножать без дополнительных передач данных, а перемещение блоков между подзадачами в ходе вычислений осуществляется с использованием более простых коммуникационных операций (алгоритм Кэннона) [3].

На этапе инициализации алгоритма выполняются следующие операции передачи данных:

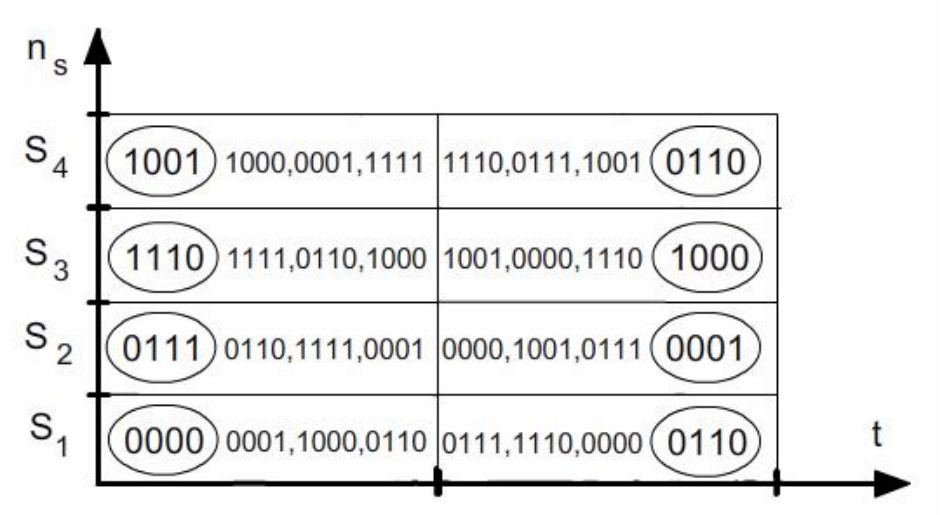
- в каждую подзадачу (i,j) передаются блоки A_{ij}, B_{ij};
- для каждой строки i решетки подзадач блоки матрицы A сдвигаются на (i– 1) позиций влево;
- для каждого столбца j решетки подзадач блоки матрицы B сдвигаются на (j– 1) позиций вверх.

Перераспределение матричных блоков в процессе вычислений осуществляется с использованием операций циклического сдвига.

На рис. 10.3 приведены граф-схема и временная диаграмма модифицированного алгоритма для решетки подзадач 2×2. Здесь, как и в предыдущем разделе, для наглядности схем используется нумерация операций из четырех цифр. Первые две цифры обозначают номер участвующего в произведении блока матрицы A , а вторые две – соответствующего блока матрицы B. Например, операция 1110 на первом шаге означает произведение $A_{11}B_{10}$, а операция 1000 на втором шаге означает, что к указанному произведению добавляется матрица, являющаяся произведением матриц $A_{10}B_{00}$.



а)



б)

Рис. 10.3 Граф-схема (а) и временная диаграмма (б) модифицированного алгоритма перемножения матриц для решетки подзадач 2×2

В результате такого начального распределения каждой подзадаче будут доступны блоки, которые могут быть перемножены без дополнительных операций передачи данных. При этом существенно упрощается получение всех последующих блоков для подзадач. После выполнения очередной операции блочного умножения каждый блок матрицы A должен быть передан предшествующей подзадаче влево по строкам решетки подзадач, а каждый блок матрицы *B* – предшествующей подзадаче вверх по столбцам решетки. Последовательность циклических сдвигов и умножений блоков исходных матриц *A* и *B* приведет к получению в подзадачах соответствующих блоков результирующей матрицы *C*.

В данном случае остаются справедливыми рекомендации по выбору топологии сети передачи данных между процессорами (в виде решетки или полного графа), которые сформулированы для предыдущего алгоритма. Что касается анализа эффективности описанной модификации алгоритма, заметим, что этот алгоритм отличается только характеристиками коммуникационных операций.

В частности, на этапе инициализации производится перераспределение блоков матриц *A* и *B* при помощи циклического сдвига матричных блоков по строкам и столбцам процессорной решетки. В случае сети со структурой пол-

ного графа все пересылки блоков могут быть выполнены одновременно, поэтому длительность операции равна времени передачи одного матричного блока. Для сети с топологией гиперкуба операция циклического сдвига требует выполнения $log_2 q$ итераций. Для сети с кольцевой структурой связей требуемое количество итераций равно q–1.

Для наиболее распространенного класса кластерных вычислительных систем с топологией связи в виде полного графа время начального перераспределения блоков определяется как

$$T^1_{s,comm} = 2 \cdot \left(\alpha + w\left(n^2/s\right)/\beta\right).$$

Величина n^2/s – размер пересылаемых блоков, а коэффициент 2 соответствует двум выполняемым операциям циклического сдвига.

После умножения матричных блоков процессоры передают свои блоки предыдущим процессорам по строкам (для блоков матрицы A) и столбцам (для блоков матрицы B) процессорной решетки. Эти операции также могут быть выполнены параллельно, так что

$$T^2_{s,comm} = 2 \cdot \left(\alpha + w\left(n^2/s\right)/\beta\right). \qquad (10.12)$$

Поскольку количество итераций алгоритма – q, с учетом оценки (10.7) общее время выполнения параллельных вычислений:

$$T^2_{s,comm} = q \left]\left(n^2/s\right) \cdot \left(2n/q - 1\right) + \left(n^2/s\right)\right[\cdot \tau + \left(2q + 2\right)\left(\alpha + w\left(n^2/s\right)/\beta\right).$$

Здесь по-прежнему $q = \sqrt{s}$ - размер решетки процессоров.

Лекция 11

Параллельное решение систем линейных уравнений

11.1 Алгоритм решения систем линейных уравнений методом Гаусса

Системы линейных уравнений возникают при решении многих прикладных задач. Матрицы коэффициентов систем линейных уравнений могут иметь различные структуру и свойства. Мы будем полагать, что решаемая система имеет плотную матрицу высокого порядка.

Линейная система n уравнений с n неизвестными x_0, x_1, …, x_{n-1} может быть представлена в виде

$$\mathbf{Ax} = \mathbf{b} \qquad (11.1)$$

где $\mathbf{A} =$

Здесь $n \times n$-матрица **A** и $n \times 1$-вектор **b** составлены из вещественных чисел, а задача заключается в нахождении неизвестного вектора **x**.

Метод Гаусса – широко используемый алгоритм решения систем линейных уравнений с невырожденной матрицей **A**. Метод заключается в приведении матрицы **A** системы (11.1) к верхнему треугольному виду

$\mathbf{U} =$

путем последовательных эквивалентных преобразований (прямой ход), с последующей последовательной подстановкой (обратный ход). К числу эквивалентных преобразований относятся:

- умножение любого из уравнений на ненулевую константу;

- перестановка уравнений;
- прибавление к уравнению любого другого уравнения системы.

Прямой ход метода Гаусса состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений. На итерации i, $0 \le i < n-1$ метода производится исключение неизвестной i для всех уравнений с номерами k, большими i (т.е. $i < k \le n-1$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу (a_{ki}/a_{ii}), с тем чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым.

Вычисления, выполняемые над элементами матрицы **A** и вектора **b**, определяются следующими соотношениями.

$$a'_{kj} = a_{kj} - \left(a_{kj} / a_{ii}\right) \cdot a_{ij}, \qquad (11.2)$$

$$b'_{k} = b_{k} - \left(a_{kj} / a_{ii}\right) \cdot b_{i},$$

$$i \le j \le n-1, \quad i < k \le n-1, \quad 0 \le i < n-1.$$

Выполнение прямого хода метода Гаусса покажем на примере системы:

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1, \\ 2x_0 + 7x_1 + 5x_2 &= 18, \\ x_0 + 4x_1 + 6x_2 &= 26. \end{aligned}$$

На первой итерации производится исключение неизвестной x_0 из второй и третьей строк. Для этого из этих строк нужно вычесть первую строку, умноженную соответственно на 2 и 1. После этих преобразований система уравнений принимает вид

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1, \\ x_1 + x_2 &= 16, \\ x_1 + 4x_2 &= 25. \end{aligned}$$

В результате остается выполнить последнюю итерацию и исключить неизвестную x_1 из третьего уравнения. Для этого необходимо вычесть вторую строку, при этом система принимает вид

$$x_0 + 3x_1 + 2x_2 = 1,$$
$$x_1 + x_2 = 16,$$
$$3x_2 = 9.$$

На рис. 11.1 представлена общая схема состояния данных на *i*-й итерации прямого хода алгоритма Гаусса. Все коэффициенты при неизвестных, расположенные ниже главной диагонали и левее столбца *i*, уже являются нулевыми. На *i*-й итерации прямого хода метода Гаусса осуществляется обнуление коэффициентов столбца *i*, расположенных ниже главной диагонали, путем вычитания строки *i*, умноженной на величину a_{kj} / a_{ii}, соответствующую *k*-й строке. После *n*-1–й итерации матрица приводится к верхнему треугольному виду.

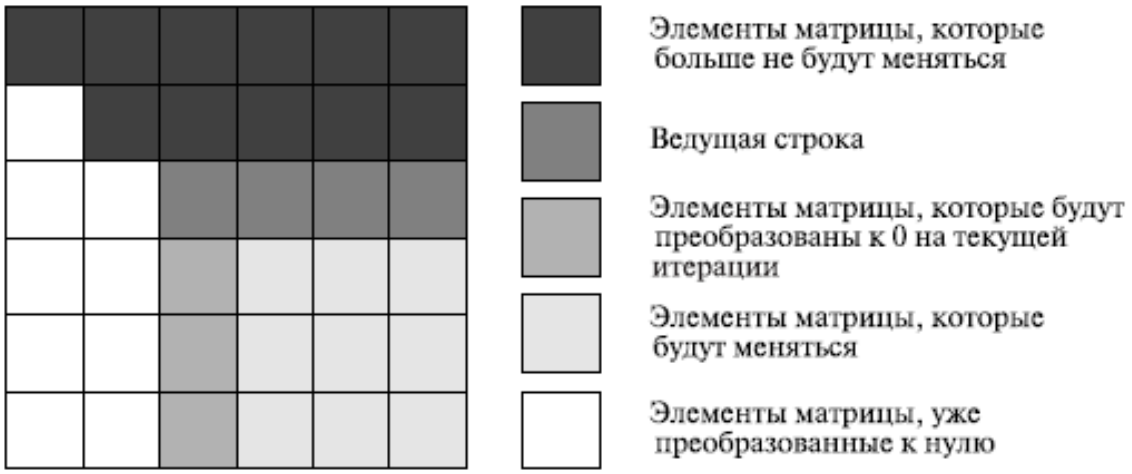


Рис. 11.1. Итерация прямого хода алгоритма Гаусса

Строка, которая используется для исключения неизвестных в прямом ходе метода Гаусса, называется ведущей, а диагональный элемент ведущей строки – ведущим элементом. Из (11.2) видно, что выполнение вычислений невозможно, если ведущий элемент равен нулю. Даже если ведущий элемент $a_{i,i}$ отличен от нуля, но имеет малое значение, деление на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Для того чтобы избежать этой проблемы, на каждой итерации прямого хода в столбце, соответствующем исключаемой неизвестной, определяют коэффициент с максимальным значением по абсолютной величине:

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

а в качестве ведущей принимают строку, в которой располагается этот коэффициент (метод главных элементов). Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок $O(n^3)$.

После приведения системы к верхнему треугольному виду из последнего уравнения преобразованной системы может быть вычислено значение переменной $x_{n-1} = x_2$, после этого из предпоследнего уравнения становится возможным определение переменной $x_{n-2} = x_1$ и т.д. В общем виде вычисления, выполняемые при обратном ходе метода Гаусса, представляются следующим образом:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$

$$x_i = \left(b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j \right) / a_{i,i}, \quad i = n-2, n-3, \ldots, 0.$$

Продолжим рассмотрение приведенного выше примера. Из последнего уравнения нетрудно видеть, что неизвестная x_2=9/3=3. Подставляя это значение во второе уравнение, определим значение неизвестной x_1=16–3=13. На последней итерации обратного хода метода Гаусса определяется значение неизвестной x_0 = 1–3·13–2·3 = –44. Заметим, что в обратном ходе исключение переменной, после того как она определена, может выполняться одновременно, т.е. параллельно, во всех уравнениях системы.

Определим вычислительную сложность метода Гаусса. В прямом ходе алгоритма для выбора ведущей строки на каждой итерации должно быть определено максимальное значение в столбце с исключаемой неизвестной. По мере исключения неизвестных количество строк и элементов в строках сокращается. Текущее число элементов строки (включая правую часть), с которыми произво-

дятся действия сложения и вычитания (первый элемент без вычислений полагается равным нулю), равно (n-i), где i, $0 \le i < n-2$ – номер итерации прямого хода. Поскольку кроме выполнения двух операций (умножения и вычитания) с каждым элементом строки предварительно должен быть вычислен масштабирующий коэффициент a_{ik}/a_{ii}, общие затраты на выполнение действий в одной строке составят $2(n-i)+1$ операций.

С учетом того, что на каждой итерации обрабатывается $n-i$ строк, общее число операций в прямом ходе метода Гаусса определяется выражением

$$T^1 = \sum_{i=0}^{n-2} \left] 2(n-i)^2 + n - i \right[. \qquad (11.3)$$

Для реализации обратного хода на каждой i-й итерации, $0 \le i < n-2$ (итерациям для удобства присваиваем номера от $n-2$ до 0) необходимо произвести $n-i-1$ умножений, столько же вычитаний, а также одно деление для определения очередной неизвестной. Следовательно, общая вычислительная сложность обратного хода составит

$$T^2 = \sum_{i=0}^{n-2} \{2(n-i-1)+1\} = \sum_{i=0}^{n-2} \{2(n-i)-1\}. \qquad (11.4)$$

Суммируя затраты на реализацию прямого и обратного хода, получаем

$$T = \sum_{i=0}^{n-2} \left\{ 2(n-i)^2 + (n-i) \right\} + 2\sum_{i=0}^{n-2} (n-i) - (n-1) = \sum_{j=2}^{n} \left(3j + 2j^2 \right) - (n-1). \quad (11.5)$$

В последнем равенстве пределы и порядок суммирования изменены с учетом замены $j = n - i$.

Используя элементарные формулы суммирования, в частности, известное равенство

$$1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6},$$

получаем следующее соотношение для оценки вычислительной сложности метода Гаусса при его реализации в виде последовательного алгоритма:

$$T = \frac{4n^3 + 9n^2 + 10n - 12}{6}.$$

При большом n можно приближенно положить

$$T \approx \frac{2}{3}n^3 + \frac{3}{2}n^2. \qquad (11.6)$$

11.2 Построение параллельного алгоритма решения методом Гаусса

Поскольку решение методом Гаусса сводится к последовательности однотипных вычислительных операций умножения и сложения над строками матрицы, в качестве подзадач можно принять вычисления, связанные с обработкой одной или нескольких строк матрицы **A** и соответствующего элемента вектора **b**. Каждая итерация, связанная с решением очередной подзадачи, начинается с выбора ведущей строки. Ищется строка с наибольшим по абсолютной величине значением среди элементов i-го столбца, соответствующего исключаемой переменной x_i.

Поскольку строки матрицы **A** закреплены за разными подзадачами, для поиска максимального значения в столбце подзадачи с номерами k, $k<i$ должны обменяться элементами при исключаемой переменной x_i. После сбора всех указанных коэффициентов может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Для продолжения вычислений ведущая подзадача должна разослать свою строку матрицы **A** и соответствующий элемент вектора **b** всем остальным подзадачам с номерами k, $k<i$. Получив ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

В обратном ходе метода Гаусса, как только какая-либо, например i–я подзадача, $0 \le i < n-1$, определила свою переменную x_i, это значение рассылается всем подзадачам с номерами k, $k<i$. В каждой подзадаче полученное значение неизвестной умножается на соответствующий коэффициент и выполняется корректировка соответствующего элемента вектора **b**.

Масштабирование и распределение подзадач по процессорам

В качестве подзадач могут быть взяты строки матрицы **A**, каждая из которых при этом закрепляется за одним процессором. Если число строк матрицы больше, чем число доступных процессоров ($s<n$), подзадачи можно укрупнить, объединив несколько строк матрицы. Если при этом используется последовательная схема разделения данных, при которой в одной подзадаче оказываются соседние строки матрицы, по мере исключения (в прямом ходе) или определения (в обратном ходе) неизвестных, все большая часть процессоров, для которой вычисления завершены, окажется простаивающей.

Для достижения хорошей балансировки процессоров целесообразно применить ленточную *циклическую* схему распределения данных между укрупненными подзадачами. Матрица **A** делится на полосы таким образом, чтобы в каждой полосе содержались как строки, обработка которых завершается раньше, так и обрабатываемые в последнюю очередь (см. рис. 11.2).

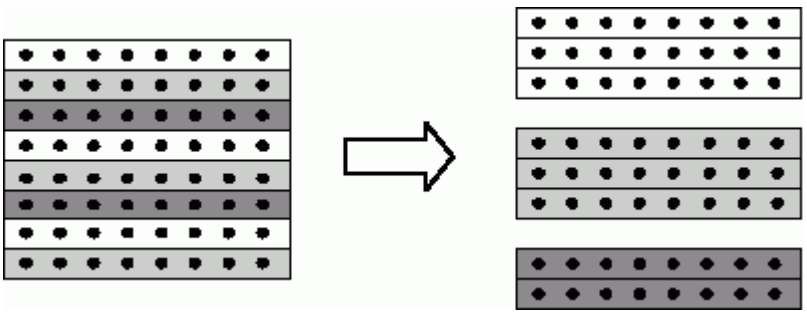


Рис. 11.2. Пример использования ленточной циклической схемы разделения строк матрицы между тремя процессорами

Распределение подзадач между процессорами должно также учитывать характер обмена данными между подзадачами. В рассматриваемом методе Гаусса взаимодействие подзадач заключается в передаче данных от одного процессора всем процессорам вычислительной системы. Поэтому в данном случае целесообразна топология сети передачи данных в виде гиперкуба или полного графа.

Анализ эффективности

Пусть n – порядок системы линейных уравнений, а s, $s<n$ – число используемых процессоров, т.е. матрица **A** имеет размер $n \times n$, а n/s – размер полосы на каждом процессоре (для простоты полагаем, что n/s – целое число). Определим сложность параллельного варианта метода Гаусса.

В прямом ходе алгоритма для выбора ведущей строки на каждой итерации и каждом процессоре должно быть определено максимальное значение в столбце с исключаемой неизвестной в пределах закрепленной за ним полосы. По мере исключения неизвестных количество строк в полосах сокращается. После сбора полученных максимальных значений, определения и рассылки ведущей строки на каждом процессоре выполняется вычитание ведущей строки из каждой строки оставшейся части строк своей полосы матрицы **A**.

Как мы установили выше, общие затраты на выполнение действий в одной строке на i-й итерации, $0 \le i < n-2$ составляет $2(n-i)+1$ операций. Если применена циклическая схема распределения данных между процессорами, то на i-й итерации каждый процессор будет обрабатывать примерно $(n-i)/s$ строк. С учетом сказанного, общее число операций параллельного варианта прямого хода метода Гаусса определяется выражением

$$T_s^1 = \sum_{i=0}^{n-2} \left\lceil \frac{(n-i)}{s} \right\rceil \left(2(n-i)+1\right).$$

Заметим, что $(n-i)/s$, как правило, не будет целым. Для построения приближенных достаточных оценок не будем учитывать операцию округления, но на каждой итерации введем операции с одной дополнительной строкой:

$$\begin{aligned} T_s^1 &= \sum_{i=0}^{n-2} \left\{ \frac{(n-i)}{s} \right\} \left(2(n-i)+1\right) + \sum_{i=0}^{n-2} \left(2(n-i)+1\right) = \\ &= \frac{1}{s} \sum_{i=0}^{n-2} \left\{ (n-i) + 2(n-i)^2 \right\} + \sum_{i=0}^{n-2} \left(2(n-i)+1\right). \end{aligned} \qquad (11.7)$$

На каждой i-й итерации обратного хода после рассылки очередной вычисленной неизвестной на каждом процессоре обновляются значения правых час-

тей для $\left[(n-i)/s\right]$ еще не обработанных строк из числа закрепленных за ним n/s строк. Поскольку для каждой правой части выполняются 2 операции (умножение и вычитание), общее число операций, необходимых для обновления всех правых частей, составит

$$T_s^2 = \sum_{i=0}^{n-2} 2 \left\lceil \frac{(n-i)}{s} \right\rceil .$$

Кроме обновления правых частей на каждой итерации обратного хода один из процессоров выполняет операцию деления для определения очередной переменной. При этом остальные процессоры, конечно, простаивают, но эти n операций (по числу определяемых переменных) необходимо включить в общие вычислительные затраты:

$$T_s^2 = \sum_{i=0}^{n-2} 2 \left\lceil \frac{(n-i)}{s} \right\rceil + n .$$

Наконец, для построения приближенных достаточных оценок, как и ранее, отменим операцию округления и добавим $2(n-1)$ операций, которые могут дополнительно иметь место на всех $2(n-1)$ итерациях обновления правых частей, если в результате округления на каждой добавляется одна строка. Тогда оценка сверху общих вычислительных затрат для реализации обратного хода составит

$$T_s^2 = \sum_{i=0}^{n-2} 2\frac{(n-i)}{s} + n + 2(n-1) = \frac{1}{s}\sum_{i=0}^{n-2} 2(n-i) + 3n - 2 . \qquad (11.8)$$

С учетом (11.7), (11.8) суммарные затраты на реализацию прямого и обратного хода составят

$$T_s = \frac{1}{s}\sum_{i=0}^{n-2}\left\{2(n-i)^2 + 3(n-i)\right\} + \sum_{i=0}^{n-2}\left\{2(n-i)+1\right\} + 3n - 2 . \qquad (11.9)$$

Как и ранее осуществим замену $j = n - i$, при этом пределы суммирования при записи слагаемых в обратном порядке будут $j = \overline{2,n}$. Тогда (11.9) перепишется в виде

$$T_s = \frac{1}{s}\sum_{i=2}^{n}\left\{2j^2+3j\right\}+\sum_{i=2}^{n}\left\{2j+1\right\}+3n-2. \qquad (11.10)$$

Применяя те же, что и ранее, элементарные формулы для подсчета сумм, получаем

$$T_s = \frac{1}{s}\left(\frac{4n^3+15n^2+11n-30}{6}\right)+\left(n^2+5n-5\right).$$

Если это допустимо, при больших n можно применять приближенную оценку:

$$T_s \approx \frac{1}{s}\left(\frac{2}{3}n^3+\frac{15}{6}n^2\right)+\left(n^2+5n-5\right). \qquad (11.11)$$

Учтем теперь затраты на передачу данных. В прямом ходе на каждой итерации для определения ведущей строки процессоры обмениваются найденными в своей полосе максимальными значениями в столбце с исключаемой переменной. Для выполнения этой операции необходимо $log_2 s$ шагов. С учетом количества итераций общие затраты на передачу максимальных значений

$$T_s^1(comm) = (n-1)\cdot log_2 s\cdot(\alpha + w/\beta), \qquad (11.12)$$

где α – латентность сети передачи данных, β – пропускная способность сети, и w – размер пересылаемого элемента данных.

На каждой итерации прямого хода метода Гаусса выполняется также рассылка выбранной ведущей строки. Сложность данной операции передачи данных определяется величиной

$$T_s^2(comm) = (n-1)\cdot log_2 s\cdot(\alpha + w/\beta). \qquad (11.13)$$

Наконец, при выполнении обратного хода алгоритма Гаусса на каждой итерации осуществляется рассылка всем процессорам значения очередной вычисленной неизвестной. Общее время, затрачиваемое на рассылку:

$$T_s^3(comm) = (n-1)\cdot log_2 s\cdot(\alpha + w/\beta). \qquad (11.14)$$

С учетом (11.9), (11.12), (11.13), (11.14) общая сложность параллельного алгоритма Гаусса составит

$$T_s^3 = \frac{1}{s}\sum_{i=2}^{n}\left(3i + 2i^2\right)\tau + +(n-1)\tau + (n-1)\cdot log_2\, s\cdot\left(3\alpha + w(n+2)/\beta\right), \quad (11.15)$$

где τ – время выполнения одной вычислительной операции.

С использованием приближенных оценок вычислительной сложности последовательного (11.6) и параллельного (11.11) алгоритмов можно построить также оценки ускорения и эффективности:

$$R = \frac{\left(\frac{2}{3}n^3 + \frac{3}{2}n^2\right)\tau}{\frac{1}{s}\left(\frac{2}{3}n^3 + \frac{15}{6}n^2\right)\tau + \left(n^2 + 5n - 5\right)\tau + (n-1)\cdot log_2\, s\cdot\left(\frac{3\alpha + w(n+2)}{\beta}\right)}, \quad (11.16)$$

$$E_s = \frac{\left(\frac{2}{3}n^3 + \frac{3}{2}n^2\right)\tau}{\left(\frac{2}{3}n^3 + \frac{15}{6}n^2\right)\tau + s\left(n^2 + 5n - 5\right)\tau + s(n-1)\cdot log_2\, s\cdot\left(\frac{3\alpha + w(n+2)}{\beta}\right)}. \quad (11.17)$$

Нетрудно заметить, что при достаточно большом n и отсутствии потерь на коммуникации могут достигаться значения ускорения и эффективности близкие к предельно возможным: s и 1 соответственно. При малых n характеристики быстро падают. Например, непосредственным подсчетом легко установить, что при n =5 и s =10 ускорение составит около 5,8.

11.3. Последовательный алгоритм метода сопряженных градиентов

Наряду с методом исключения Гаусса для решения систем линейных уравнений широко используются итерационные алгоритмы, в которых строится последовательность приближений x^0, x^1, ..., x^k, ... к искомому точному решению x^* системы (1). Процесс строится так, что очередное приближение дает оценку точного решения с меньшей погрешностью, и при увеличении числа шагов оценка точного решения может быть получена с любой требуемой точностью.

К преимуществам итерационных методов можно отнести сравнительно малый объем вычислений при решении разреженных систем, возможность быст-

рого получения начального приближения, простота организации параллельных вычислений. Рассмотрим один из наиболее известных итерационных алгоритмов – метод сопряженных градиентов.

Будем полагать, что матрица **A** системы симметричная и положительно определенная, т.е. $\mathbf{A}=\mathbf{A}^{\mathrm{T}}$ и $x^{\mathrm{T}}\mathbf{A}x>0$. При этом функция

$$q(\mathbf{x})=\frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}-\mathbf{x}^T\mathbf{b}+c \qquad (11.10)$$

имеет единственный минимум в точке x^*, совпадающей с точным решением системы (11.1). Метод сопряженных градиентов позволяет найти это решение путем минимизации функции $q(\mathbf{x})$.

Итерация метода сопряженных градиентов состоит в вычислении очередного приближения к точному решению в соответствии с правилом:

$$\mathbf{x}^k=\mathbf{x}^{k-1}+s^k\mathbf{d}^k, \qquad (11.11)$$

т.е. новое приближение $\mathbf{x}^{\mathrm{k}}$ вычисляется путем добавления к приближению $\mathbf{x}^{k-1}$, полученному на предыдущем шаге, вектора направления $\mathbf{d}^k$, умноженного на скалярную величину s^{k} (шаг).

Перед началом выполнения алгоритма полагают

$\mathbf{x}^0=0$ и $\mathbf{d}^0=0$ и $\mathbf{g}^0=\mathbf{b}$.

Шаг 1: Вычисление градиента:

$\mathbf{g}^k=\mathbf{A}\mathbf{x}^{k-1}-\mathbf{b}$.

Шаг 2: Вычисление вектора направления:

$$\mathbf{d}^k=-\mathbf{g}^k+\frac{\left(\left(\mathbf{g}^k\right)^T\mathbf{g}^k\right)}{\left(\left(\mathbf{g}^{k-1}\right)^T\mathbf{g}^{k-1}\right)}\mathbf{d}^{k-1},$$

где $(\mathbf{g}^{\mathrm{T}}\mathbf{g})$ – скалярное произведение векторов.

Шаг 3: Вычисление величины смещения по выбранному направлению:

$$s^k=\frac{\left(\left(\mathbf{d}^k\right)^T\mathbf{g}^k\right)}{\left(\left(\mathbf{d}^k\right)^T\cdot\mathbf{A}\cdot\mathbf{d}^k\right)}.$$

Шаг 4: Вычисление нового приближения по соотношению (11.11).

Описанный алгоритм содержит две операции умножения матрицы на вектор, четыре операции скалярного произведения и пять операций (сложение и умножение на скаляр) над векторами. Общее количество операций на одной итерации

$t_1=2n^2+13n$.

Известно, что точное решение достигается после выполнения n итераций алгоритма метода сопряженных градиентов. Следовательно, всего необходимо выполнить

$$T_1 = 2n^3 + 13n^2 \qquad (11.12)$$

операций, т.е. сложность описанного алгоритма имеет порядок $O(n^3)$.

11.4 Организация параллельных вычислений

Поскольку итерации алгоритма сопряженных градиентов должны выполняться последовательно, распараллеливание вычислений возможно лишь в пределах одной итерации. Основные вычислительные затраты на каждой итерации связаны с умножением матрицы **A** на векторы **x** и **d**. Для организации параллельных вычислений в данном случае могут использоваться схемы, рассмотренные в лекции 10.

Кроме того, должны выполняться операции над векторами: скалярное произведение, сложение, умножение на скаляр. Организация этих вычислений должна согласовываться с принятым способом (типом декомпозиции) умножения матрицы на вектор. Общая рекомендация может заключаться в том, что при малом размере векторов возможно дублирование данных, при большой размерности системы уравнений целесообразно осуществлять разделение на блоки.

Анализ эффективности

Анализ эффективности параллельного алгоритма проведем для варианта с ленточным горизонтальным разделением матрицы, в предположении, что все

обрабатываемые векторы дублируются. Вычислительная сложность последовательного алгоритма дается соотношением (11.12). Определим вычислительные затраты при параллельной реализации метода сопряженных градиентов.

Вычислительная сложность параллельного умножения матрицы на вектор при ленточном разделении по строкам

$$T_s^1 = 2n\left\lceil n/s \right\rceil \cdot (2n-1).$$

С учетом дополнительных операций над векторами при условии их дублирования общая вычислительная сложность параллельного алгоритма составит

$$T_s^1 = n(2 \cdot \left\lceil n/s \right\rceil \cdot (2n-1) + 13n).$$

Если $\lceil n/s \rceil$ – целое число, ускорение и эффективность соответственно

$$R = \frac{2n^3 + 13n^2}{\frac{4n^3 - 2n^2}{s} + 13n^2},$$

$$E_s = \frac{2n^3 + 13n^2}{4n^3 - 2n^2 + 13sn^2}.$$

С учетом информационного взаимодействия процессоров при умножении матрицы на вектор сложность параллельных вычислений

$$T_s = \left(\left(4n^3 - 2n^2\right)/s + 13n \right) + 2n\left(\alpha \cdot \left\lceil \log s \right\rceil + w\left(n/s\right)\left(s-1\right)/\beta\right).$$

Тогда окончательно оценки ускорения и эффективности

$$R = \frac{2n^3 + 13n^2}{\frac{4n^3 - 2n^2}{s} + 13n^2 + 2n\left(\alpha \cdot \left\lceil \log s \right\rceil + w\left(n/s\right)\left(s-1\right)/\beta\right)}, \qquad (11.13)$$

$$E_s = \frac{2n^3 + 13n^2}{4n^3 - 2n^2 + 13sn^2 + 2sn\left(\alpha \cdot \left\lceil \log s \right\rceil + w\left(n/s\right)\left(s-1\right)/\beta\right)}. \qquad (11.14)$$

Представляет интерес проанализировать предельно достижимые в этом методе показатели ускорения и эффективности. Из (11.13) легко установить, что если потери на передачу данных отсутствуют, то при сравнительно малом *n* и

достаточно большом s, таких что можно пренебречь первым слагаемым в знаменателе (11.13), предельно достижимые ускорение и эффективность соответственно

$$R \approx 1 + \frac{2}{13} n,$$

$$E_s \approx \frac{1}{s}.$$

Если, наоборот, n велико, а s мало настолько, что

$$R \approx \frac{2n^3 + 13n^2}{4n^3 / s} = \frac{1}{2} s,$$

$$E_s \approx \frac{1}{2}.$$

Ясно, что в других случаях показатели ускорения и эффективности будут принимать промежуточные значения. Это означает, что в данном случае не может быть достигнуто высокое ускорение, а предельно достижимая эффективность 0,5, т.е. при достаточно больших n метод значительно проигрывает методу Гаусса. Следовательно, метод сопряженных градиентов является не вполне подходящим для параллельной реализации на многопроцессорной вычислительной системе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Барский, А.Б. Параллельные процессы в вычислительных системах. Планирование и организация / А.Б. Барский. – М.: Радио и связь, 1990. – 256 с.
2. Воеводин, В.В. Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. СПб.: БХВ-Петербург, 2002.
3. Гергель, В.П. Теория и практика параллельных вычислений: учеб. пособие/ В.П. Гергель.- М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007. – 423 с.
4. Гладких, Б.А. Информатика от абака до Интернета. Введение в специальность: учеб. пособие / Б.А. Гладких. – Томск: Изд-во НТЛ, 2005. – 484 с.
5. Головашкин, Д.Л. Методы параллельных вычислений: учеб. пособие / Д.Л. Головашкин. – Самара: Изд-во СГАУ, 2002. Ч. I. – 92 с.
6. Головашкин, Д.Л. Методы параллельных вычислений: учеб. пособие / Д.Л. Головашкин, С.П. Головашкина. – Самара: Изд-во СГАУ, 2003. Ч. II. – 103 с.
7. Введение в программирование для параллельных ЭВМ и кластеров: учеб. пособие / В.В. Кравчук, С.Б. Попов, А.Ю. Привалов [и др.]. – Самара: Самар. науч. центр РАН, Самар. гос. аэрокосм. ун-т, 2000. – 87 с.
8. Тоффоли, Т. Машины клеточных автоматов / Т. Тоффоли, Н. Марголус. – М.: Мир, 1991.
9. Фурсов, В.А. Технология итерационного планирования распределения ресурсов гетерогенного кластера / В.А. Фурсов, В.А. Шустов, С.А. Скуратов. Тр. Всерос. науч. конф. "Высокопроизводительные вычисления и их приложения"; Тр. Всерос. науч. конф. г. Черноголовка , 30 октября - 2 ноября 2000г. – С. 43-46.
10. Bruce P. Lester, The Art of Parallel Programming. Prentice Hall, Englewood Cliffs, New Jersey, 1993, 376 p.
11. Computational Science: Ensuring America`s Competitiveness. President`s Information Tehnology Advisory Committee. May 27, 2005.

ОГЛАВЛЕНИЕ

Учебное издание

Гергель Виктор Павлович
Фурсов Владимир Алексеевич

ЛЕКЦИИ ПО ПАРАЛЛЕЛЬНЫМ ВЫЧИСЛЕНИЯМ
Учебное пособие

Редактор Т. К. К р е т и н и н а
Доверстка Т. Е. П о л о в н е в а

Подписано в печать 16.10.09. Формат 60×84 1/16.
Бумага офсетная. Печать офсетная.
Печ. л. 10,25. Тираж 100 экз. Заказ .

Самарский государственный
аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

Изд-во Самарского государственного
аэрокосмического университета.
443086 Самара, Московское шоссе, 34.

**ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П.КОРОЛЕВА»**

В.П. Гергель, В.А. Фурсов

ЛЕКЦИИ
ПО ПАРАЛЛЕЛЬНЫМ ВЫЧИСЛЕНИЯМ

С А М А Р А 2009