Основы информационных технологий

В.П. Гергель

ТЕОРИЯ И ПРАКТИКА ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Учебное пособие

Рекомендовано Советом учебно-методического объединения классических университетов России по прикладной математике и информатике

Интернет-Университет
Информационных Технологий
www.intuit.ru

БИНОМ.
Лаборатория знаний
www.lbz.ru

Москва
2007

УДК 004.42.032.24(07)
ББК 32.973.2-018.2я7
Г37

Гергель В.П.

Г37 Теория и практика параллельных вычислений: учебное пособие / В.П. Гергель. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. – 423 с.: ил., табл. – (Основы информационных технологий).

ISBN 978-5-9556-0096-3 (ИНТУИТ.РУ)
ISBN 978-5-94774-645-7 (БИНОМ.ЛЗ)

Учебное пособие содержит материал для работы в области параллельного программирования. Дается краткая характеристика принципов построения параллельных вычислительных систем, рассматриваются математические модели параллельных алгоритмов и программ для анализа эффективности параллельных вычислений, приводятся примеры конкретных параллельных методов для решения типовых задач вычислительной математики.

Пособие подготовлено по заданию Инновационной образовательной программы Нижегородского госуниверситета им. Н.И. Лобачевского в рамках Национального проекта «Образование».

Для студентов, аспирантов и специалистов, изучающих и практически использующих параллельные компьютерные системы для решения вычислительно трудоемких задач.

УДК 004.42.032.24(07)
ББК 32.973.2-018.2я7

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (495) 157-5272,
e-mail: Lbz@aha.ru, http://www.Lbz.ru

ISBN 978-5-9556-0096-3 (ИНТУИТ.РУ)
ISBN 978-5-94774-645-7 (БИНОМ.ЛЗ)

О проекте

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир – это мир компьютеров и информации. Компьютерная индустрия – самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долго. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов – вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-Университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, – очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать
в Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Об авторе

Гергель Виктор Павлович – доктор технических наук, профессор, декан факультета вычислительной математики и кибернетики Нижегородского государственного университета им. Н.И. Лобачевского. Автор нового научного направления в области математических моделей, методов и программных средств информационного обеспечения процессов поиска рациональных вариантов. Имеет значительный опыт профессиональной деятельности в области разработки сложного программного обеспечения для поддержки процессов выбора решений. Активно занимается учебной и научной работой в области высокопроизводительных параллельных вычислений. Автор более 120 научных работ.

Лекции

Содержание

Введение

Применение параллельных вычислительных систем (ПВС) является стратегическим направлением развития вычислительной техники. Это обстоятельство вызвано не только принципиальным ограничением максимально возможного быстродействия обычных последовательных ЭВМ, но и практически постоянным наличием вычислительных задач, для решения которых возможностей существующих средств вычислительной техники всегда оказывается недостаточно. Так, проблемы «большого вызова» [54] возможностям современной науки и техники: моделирование климата, генная инженерия, проектирование интегральных схем, анализ загрязнения окружающей среды, создание лекарственных препаратов и др. — требуют для своего анализа ЭВМ с производительностью более 1000 миллиардов операций с плавающей запятой в секунду (1 TFlops).

Проблема создания высокопроизводительных вычислительных систем относится к числу наиболее сложных научно-технических задач современности. Ее разрешение возможно только при всемерной концентрации усилий многих талантливых ученых и конструкторов, предполагает использование всех последних достижений науки и техники и требует значительных финансовых инвестиций. Тем не менее достигнутые в последнее время успехи в этой области впечатляют. Так, в рамках принятой в США в 1995 г. программы «Ускоренной стратегической компьютерной инициативы» (Accelerated Strategic Computing Initiative – ASCI) [25] была поставлена задача увеличения производительности суперЭВМ в 3 раза каждые 18 месяцев и достижение уровня производительности в 100 триллионов операций в секунду (100 TFlops) в 2004 г. Одной из наиболее быстродействующих супер-ЭВМ в настоящее время является компьютер SX-6 японской фирмы NEC с быстродействием одного векторного процессора порядка 8 миллиардов операций в секунду (8 GFlops). Достигнутые показатели быстродействия для многопроцессорных систем гораздо выше: например, система ASCI Red фирмы Intel (США, 1997) имеет предельную (пиковую) производительность 1,8 триллионов операций в секунду (1,8 TFlops). Список наиболее быстродействующих вычислительных систем Top 500 на момент издания данного курса лекций возглавляет вычислительный комплекс BlueGene/L, содержащий более 64 тысяч процессоров (!) с общей суммарной пиковой производительностью более чем 380 TFlops.

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

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

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

Другая и, пожалуй, теперь основная причина сдерживания массового распространения параллелизма состоит в том, что для проведения параллельных вычислений необходимо «параллельное» обобщение традиционной — последовательной — технологии решения задач на ЭВМ. Так, численные методы в случае многопроцессорных систем должны проектироваться как системы параллельных и взаимодействующих между собой процессов, допускающих исполнение на независимых процессорах. Применяемые алгоритмические языки и системное программное обеспечение должны обеспечивать создание параллельных программ, организовывать синхронизацию и взаимоисключение асинхронных процессов и т.п.

Среди прочих можно выделить следующий ряд общих проблем, возникающих при использовании параллельных вычислительных систем [22]:

• **Потери производительности для организации параллелизма** – согласно *гипотезе Минского* (Minsky), ускорение, достигаемое при использовании параллельной системы, пропорционально двоичному логарифму от числа процессоров (т.е. при 1000 процессорах возможное ускорение оказывается равным 10).

Комментируя данное высказывание, можно отметить, что, безусловно, подобная оценка ускорения справедлива при распараллеливании определенных алгоритмов. Вместе с тем существует большое количество задач, при параллельном решении которых дости-

гается практически 100%-е использование всех имеющихся процессоров параллельной вычислительной системы.

• **Существование последовательных вычислений** – в соответствии с *законом Амдаля* (Amdahl) ускорение процесса вычислений при использовании p процессоров ограничивается величиной

$$S \le \frac{1}{f + (1-f)/p},$$

где f есть доля последовательных вычислений в применяемом алгоритме обработки данных (т.е., например, при наличии всего 10% последовательных команд в выполняемых вычислениях эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных).

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

• **Зависимость эффективности параллелизма от учета характерных свойств параллельных систем** – в отличие от единственности классической схемы фон Неймана последовательных ЭВМ, параллельные системы характеризуются существенным разнообразием архитектурных принципов построения, и максимальный эффект от параллелизма может быть получен только при полном использовании всех особенностей аппаратуры; как результат, перенос параллельных алгоритмов и программ между разными типами систем становится затруднительным (если вообще возможен).

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

• **Существующее программное обеспечение ориентировано в основном на последовательные ЭВМ** – данное замечание состоит в том, что для большого количества задач уже имеется подготовленное программное

обеспечение и все эти программы ориентированы главным образом на последовательные ЭВМ; как результат, переработка такого количества программ для параллельных систем вряд ли окажется возможной.

> Ответ на данное высказывание сравнительно прост: если существующие программы обеспечивают решение поставленных задач, то, конечно, переработка этих программ не является необходимой. Однако если последовательные программы не позволяют получать решения задач за приемлемое время или же возникает необходимость решения новых задач, то насущной становится разработка нового программного обеспечения и эти программы могут реализовываться для параллельного выполнения.

Подводя итог всем перечисленным проблемам и замечаниям, можно заключить, что параллельные вычисления являются перспективной (и очень привлекательной) областью применения вычислительной техники и представляют собой сложную научно-техническую область деятельности. Тем самым, знание современных тенденций развития ЭВМ и аппаратных средств для достижения параллелизма, умение разрабатывать модели, методы и программы параллельного решения задач обработки данных следует отнести к числу важных квалификационных характеристик современного специалиста по прикладной математике, информатике и вычислительной технике.

Проблематика параллельных вычислений является чрезвычайно широкой областью теоретических исследований и практически выполняемых работ и обычно подразделяется на следующие направления деятельности:

• **разработка параллельных вычислительных систем** – обзор принципов построения параллельных систем приведен в лекции 1 данного курса лекций; дополнительная информация по данному вопросу может быть получена в [11, 14, 16, 24, 28, 29, 47, 59, 76, 77];

• **анализ эффективности параллельных вычислений** для оценки *получаемого ускорения* вычислений и *степени использования* всех возможностей компьютерного оборудования при параллельных способах решения задач – вопросы данного направления работ рассматриваются в лекции 2 (анализ эффективности организации процессов передачи данных как одной из важных составляющих параллельных вычислений выполняется отдельно в лекции 3); проблема анализа эффективности параллельных вычислений широко обсуждается в литературе – см., например, [2, 7, 8, 22, 77];

• **формирование общих принципов разработки параллельных алгоритмов** для решения сложных вычислительно трудоемких задач – возможная практическая методика получения параллельных вычислительных схем приводится в Лекции 4; при рассмотрении данной важной темы может оказаться полезным материал, излагаемый в [2, 32, 63];

• **создание и развитие системного программного обеспечения для параллельных вычислительных систем** – в лекции 5 описывается стандарт MPI

(*Message Passing Interface*), программные реализации которого позволяют разрабатывать параллельные программы и, кроме того, снизить в значительной степени остроту важной проблемы параллельного программирования – обеспечение мобильности (переносимости между разными вычислительными системами) создаваемого прикладного программного обеспечения. Дополнительные сведения в этой области параллельных вычислений могут быть получены, например, в [4, 12, 27, 35, 40 – 42, 57, 63, 69]. Следует отметить, что использование MPI-библиотек для разработки параллельных программ является практически общепринятой технологией параллельного программирования для вычислительных систем с распределенной памятью;

• **создание и развитие параллельных алгоритмов** для решения прикладных задач в разных областях практических приложений – примеры подобных алгоритмов приведены в лекциях 6 – 11 данного курса; более широко сведения о существующих параллельных методах вычислений для различных классов задач представлены в [2, 5, 17, 22, 32, 33, 51, 62, 63, 66, 75, 77]; в этих же лекциях рассмотрены возможные программные реализации рассматриваемых алгоритмов.

В завершение учебного материала в лекции 12 проводится рассмотрение учебно-исследовательской программной системы Параллельная Лаборатория (*ПараЛаб*), которая может быть использована для организации лабораторного практикума по данному курсу для изучения и исследования эффективности параллельных алгоритмов.

Помимо использования системы ПараЛаб, лабораторный практикум включает также выполнение лабораторных работ, направленных на практическое освоение параллельного программирования; каждая лабораторная работа курса имеет детальное описание полного цикла программной реализации рассматриваемого в рамках работы одного из параллельных алгоритмов.

В основе настоящего учебника лежит лекционный материал учебного курса «Многопроцессорные вычислительные системы и параллельное программирование», который читается с 1996 г. в Нижегородском государственном университете на факультете вычислительной математики и кибернетики для студентов, обучаемых по специальности «Прикладная математика и информатика» и новому направлению подготовки «Информационные системы».

Часть необходимого теоретического материала была подготовлена во время научных стажировок в университете г. Трир (Германия, декабрь 1996 г. — январь 1997 г.), университете г. Роскильде и Датском техническом университете (Дания, май 2000 г.), университете г. Делфт (октябрь 2005 г.), университете штата Северная Каролина (США, октябрь – декабрь 2006 г.). Вычислительные эксперименты, необходимость которых возникала при подготовке пособия, были выполнены на высокопроизводительном вы-

числительном кластере Нижегородского университета (оборудование кластера поставлено в ННГУ в рамках Академической программы компании Intel в 2001 г., описание состава и показатели производительности кластера приводятся в лекции 1 учебного курса).

Значительная часть работ по развитию учебно-методического и программного обеспечения данного курса была проведена в 2002 – 2005 гг. в учебно-исследовательской лаборатории «Информационные технологии» (ИТЛаб), созданной в Нижегородском университете при поддержке компании Интел.

Подготовка публикуемого варианта учебных материалов была выполнена в 2005 – 2006 гг. в рамках учебно-исследовательского проекта, который был поддержан компанией Майкрософт, по созданию учебно-методического обеспечения высокопроизводительных кластеров, функционирующих под управлением недавно разработанной системы Microsoft Compute Cluster Server.

Для проведения занятий по курсу «Многопроцессорные вычислительные системы и параллельное программирование» были подготовлены электронные презентации, содержание которых представлено на сайте (http://www.software.unn.ac.ru/ccam/?doc=98).

Данный вариант учебного курса является расширенной и переработанной версией учебного пособия, опубликованного в Нижегородском университете в 2001 г. (второе издание пособия было опубликовано в 2003 г.). За прошедший период учебный курс активно использовался в разных формах подготовки и переподготовки кадров в области современных компьютерных технологий более чем в 20 вузах страны. Материалы учебного курса применялись в проводимых в Нижегородском университете при поддержке компании Интел в 2004 – 2006 гг. Зимних школах по параллельному программированию.

Разработанные учебные материалы послужили для компании Майкрософт одной из важных причин выделения Нижегородского госуниверситета в числе 10 лучших университетов мира в области высокопроизводительных вычислений.

Автор данного курса лекций считает своим приятным долгом выразить благодарность за совместную успешную работу и дружескую поддержку своим коллегам – преподавателям кафедры математического обеспечения ЭВМ факультета вычислительной математики и кибернетики Нижегородского госуниверситета. Автор глубоко признателен Л.В. Нестеренко (Интел), ценные замечания которой значительно способствовали улучшению разрабатываемых учебных материалов. И, безусловно, настоящее издание было подготовлено только благодаря неизменной дружеской поддержке К. Фаенова (Майкрософт) и И.Р. Агамирзяна (Майкрософт).

Лекция 1. Принципы построения параллельных вычислительных систем

Лекция посвящена рассмотрению принципов построения параллельных вычислительных систем (ПВС). Дана краткая характеристика способов достижения параллелизма, приведены примеры ПВС. Приводится классификация параллельных вычислительных систем, рассматриваются типовые топологии сетей передачи данных в ПВС.

Ключевые слова: режим разделения времени, распределенные вычисления, многопроцессорная вычислительная система, сеть передачи данных, классификация Флинна, кластер, мультипроцессор, мультикомпьютер, общая разделяемая и распределенная память, когерентность, топология.

1.1. Пути достижения параллелизма

В общем плане под *параллельными вычислениями* понимаются процессы обработки данных, в которых одновременно могут выполняться несколько операций компьютерной системы. Достижение параллелизма возможно только при выполнении следующих требований к архитектурным принципам построения вычислительной среды:

• **независимость функционирования отдельных устройств ЭВМ** – данное требование относится в равной степени ко всем основным компонентам вычислительной системы: к устройствам ввода-вывода, обрабатывающим процессорам и устройствам памяти;

• **избыточность элементов вычислительной системы** – организация избыточности может осуществляться в следующих основных формах:

— *использование специализированных устройств*, таких, например, как отдельные процессоры для целочисленной и вещественной арифметики, устройства многоуровневой памяти (регистры, кэш);

— *дублирование устройств ЭВМ* путем использования, например, нескольких однотипных обрабатывающих процессоров или нескольких устройств оперативной памяти.

Дополнительной формой обеспечения параллелизма может служить *конвейерная* реализация обрабатывающих устройств, при которой выполнение операций в устройствах представляется в виде исполнения последовательности составляющих операцию подкоманд. Как результат, при вычислениях на таких устройствах на разных стадиях обработки могут находиться одновременно несколько различных элементов данных.

Возможные пути достижения параллелизма детально рассматриваются в [2, 11, 14, 28, 45, 59]; в этих же работах описывается история развития параллельных вычислений и приводятся примеры конкретных параллельных ЭВМ (см. также [24, 76]).

При рассмотрении проблемы организации параллельных вычислений следует различать следующие возможные режимы выполнения независимых частей программы:

— *многозадачный режим* (*режим разделения времени*), при котором для выполнения нескольких процессов используется единственный процессор. Данный режим является псевдопараллельным, когда активным (исполняемым) может быть один, единственный процесс, а все остальные процессы находятся в состоянии ожидания своей очереди; применение режима разделения времени может повысить эффективность организации вычислений (например, если один из процессов не может выполняться из-за ожидания вводимых данных, процессор может быть задействован для выполнения другого, готового к исполнению процесса – см. [73]). Кроме того, в данном режиме проявляются многие эффекты параллельных вычислений (необходимость взаимоисключения и синхронизации процессов и др.), и, как результат, этот режим может быть использован при начальной подготовке параллельных программ;

— *параллельное выполнение*, когда в один и тот же момент может выполняться несколько команд обработки данных. Такой режим вычислений может быть обеспечен не только при наличии нескольких процессоров, но и при помощи конвейерных и векторных обрабатывающих устройств;

— *распределенные вычисления;* данный термин обычно применяют для указания параллельной обработки данных, при которой используется несколько обрабатывающих устройств, достаточно удаленных друг от друга, в которых передача данных по линиям связи приводит к существенным временным задержкам. Как результат, эффективная обработка данных при таком способе организации вычислений возможна только для параллельных алгоритмов с низкой интенсивностью потоков межпроцессорных передач данных. Перечисленные условия являются характерными, например, при организации вычислений в *многомашинных вычислительных комплексах*, образуемых объединением нескольких отдельных ЭВМ с помощью каналов связи локальных или глобальных информационных сетей.

В рамках данного учебного материала основное внимание будет уделяться второму типу организации параллелизма, реализуемому на многопроцессорных вычислительных системах.

1.2. Примеры параллельных вычислительных систем

Разнообразие параллельных вычислительных систем поистине огромно. В каком-то смысле каждая такая система уникальна. В них устанавливаются различные аппаратные составляющие: процессоры (Intel, Power, AMD, HP, Alpha, Nec, Cray, ...), сетевые карты (Ethernet, Myrinet, Infiniband, SCI, ...). Они функционируют под управлением различных операционных систем (версии Unix/Linux, версии Windows, ...) и используют различное прикладное программное обеспечение. Кажется, что найти между ними что-то общее практически невозможно. Конечно же, это не так, и ниже мы попытаемся с общих позиций сформулировать некоторые известные варианты классификаций параллельных вычислительных систем, но прежде рассмотрим несколько примеров.

1.2.1. Суперкомпьютеры

Началом эры суперкомпьютеров с полным правом может считаться 1976 год – год появления первой векторной системы Cray 1. Результаты, показанные этой системой, пусть и на ограниченном в то время наборе приложений, были столь впечатляющи в сравнении с остальными, что система заслуженно получила название «суперкомпьютер» и в течение длительного времени определяла развитие всей индустрии высокопроизводительных вычислений. Однако в результате совместной эволюции архитектур и программного обеспечения на рынке стали появляться системы с весьма кардинально различающимися характеристиками, потому само понятие «суперкомпьютер» стало многозначным, и пересматривать его пришлось неоднократно.

Попытки дать определение термину *суперкомпьютер*, опираясь только на производительность, неизбежно приводят к необходимости постоянно поднимать планку, отделяющую его от рабочей станции или даже обычного настольного компьютера. Так, по определению Оксфордского словаря вычислительной техники 1986 года, для того чтобы получить это гордое название, нужно было иметь производительность в 10 MFlops[1]. Сегодня, как известно, производительность настольных систем на два порядка выше.

Из альтернативных определений наиболее интересны два: экономическое и философское. Первое из них гласит, что суперкомпьютер – это система, цена которой выше 1–2 млн. долларов. Второе – что суперком-

[1] MFlops – million of floating point operations per second – миллион операций над числами с плавающей запятой в секунду, GFlops – миллиард, TFlops – триллион соответственно.

пьютер – это компьютер, мощность которого всего на порядок меньше необходимой для решения современных задач.

1.2.1.1. Программа ASCI

Программа ASCI (http://www.llnl.gov/asci/) – Accelerated Strategic Computing Initiative, поддерживаемая Министерством энергетики США, в качестве одной из основных целей имеет создание суперкомпьютера с производительностью в 100 TFlops.

Первая система серии ASCI – **ASCI Red**, построенная в 1996 г. компанией Intel, стала и первым в мире компьютером с производительностью в 1 TFlops (в дальнейшем производительность системы была доведена до 3 TFlops).

Тремя годами позже появились **ASCI Blue Pacific** от IBM и **ASCI Blue Mountain** от SGI, ставшие первыми на тот момент суперкомпьютерами с быстродействием 3 TFlops.

Наконец, в июне 2000 г. была введена в действие система **ASCI White** (http://www.llnl.gov/asci/platforms/white/) с пиковой производительностью свыше 12 TFlops (реально показанная производительность на тесте LINPACK составила на тот момент 4938 GFlops, позднее данный показатель был доведен до 7304 GFlops).

Аппаратно ASCI White представляет собой систему IBM RS/6000 SP с 512 симметричными мультипроцессорными (SMP) узлами. Каждый узел имеет 16 процессоров, система в целом – 8192 процессора. Оперативная память системы – 4 TB, емкость дискового пространства 180 TB.

Все узлы системы являются симметричными мультипроцессорами IBM RS/6000 POWER3 с 64-разрядной архитектурой. Каждый узел автономен, обладает собственной памятью, операционной системой, локальным диском и 16 процессорами.

Процессоры POWER3 являются суперскалярными 64-разрядными чипами конвейерной организации с двумя устройствами по обработке команд с плавающей запятой и тремя устройствами по обработке целочисленных команд. Они способны выполнять до восьми команд за тактовый цикл и до четырех операций с плавающей запятой за такт. Тактовая частота каждого процессора 375 MHz.

Программное обеспечение ASCI White поддерживает смешанную модель программирования – передача сообщений между узлами и многопотоковость внутри SMP-узла.

Операционная система представляет собой версию UNIX – IBM AIX. AIX поддерживает как 32-, так и 64-разрядные системы RS/6000.

Поддержка параллельного кода на ASCI White включает параллельные библиотеки, отладчики (в частности, TotalView), профилировщики,

утилиты IBM и сервисные программы по анализу эффективности выполнения. Поддерживаются библиотеки MPI, OpenMP, потоки POSIX и транслятор директив IBM. Имеется параллельный отладчик IBM.

1.2.1.2. Система BlueGene

Самый мощный на данный момент суперкомпьютер в мире создан IBM. Точнее говоря, работы над ним еще не закончены. В настоящий момент система имеет полное название «BlueGene/L DD2 beta-System» и представляет собой «первую очередь» полной вычислительной системы. Согласно прогнозам, к моменту ввода в строй ее пиковая производительность достигнет 360 TFlops.

В качестве основных областей применения разработчики называют гидродинамику, квантовую химию, моделирование климата и др.

Текущий вариант системы имеет следующие характеристики:

- 32 стойки по 1024 двухъядерных 32-битных процессора PowerPC 440 с тактовой частотой 0,7 GHz;
- пиковая производительность – порядка 180 TFlops;
- максимальная показанная производительность (на тесте LINPACK) – 135 TFlops.

1.2.1.3. МВС-1000

Один из самых известных в России суперкомпьютеров – Многопроцессорная вычислительная система МВС-1000М – был установлен в Межведомственном суперкомпьютерном центре Российской академии наук.

Работы по созданию МВС-1000М проводились с апреля 2000 года по август 2001 года.

Согласно официальным данным (http://www.jscc.ru) состав системы:

- 384 двухпроцессорных модуля на базе Alpha 21264 с тактовой частотой 667 MHz (кэш L2 4 Mb), собранные в виде 6 базовых блоков, по 64 модуля в каждом;
- управляющий сервер и файл-сервер NetApp F840;
- сети Myrinet 2000 и Fast/Gigabit Ethernet;
- сетевой монитор;
- система бесперебойного электропитания.

Каждый вычислительный модуль имеет по 2 Gb оперативной памяти, HDD 20 Gb, сетевые карты Myrinet (2000 Mbit) и Fast Ethernet (100 Mbit).

При обмене данными между модулями с использованием протоколов MPI на сети Myrinet пропускная способность в МВС-1000М составляет 110 — 150 Mb в секунду.

Программное обеспечение системы составляют:
- операционные системы управляющего и резервного управляющего сервера – ОС Linux RedHat 6.2 с поддержкой SMP;
- операционная система вычислительных модулей – ОС Linux RedHat 6.2 с поддержкой SMP;
- операционная среда параллельного программирования – пакет MPICH for GM;
- программные средства коммуникационных сетей (Myrinet, Fast Ethernet);
- оптимизированные компиляторы языков программирования C, C++, Fortran фирмы Compaq;
- отладчик параллельных программ TotalView;
- средства профилирования параллельных программ;
- средства параллельного администрирования.

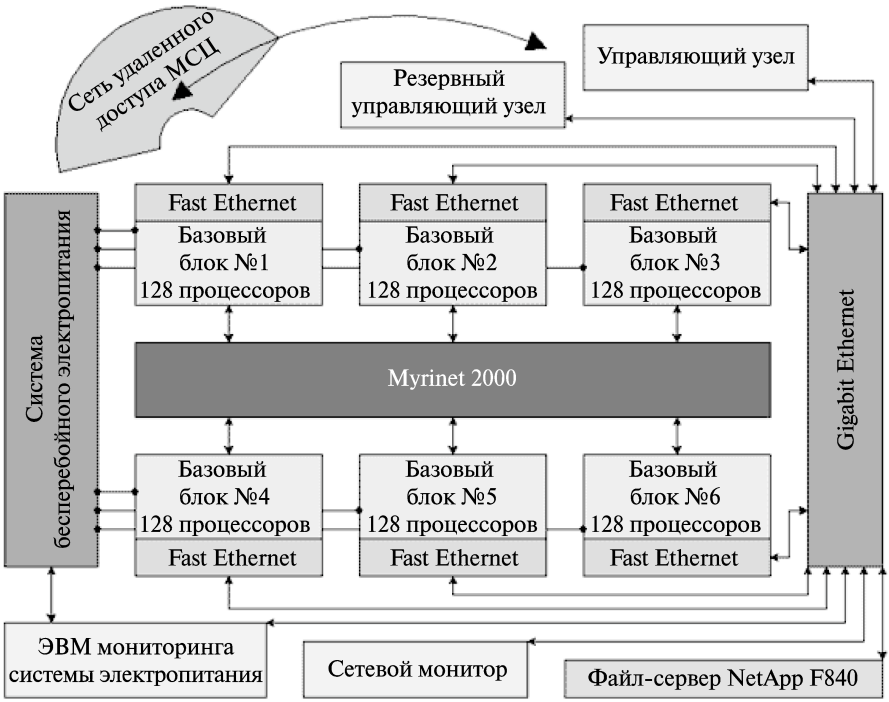


Рис. 1.1. Структура суперкомпьютера МВС-1000М

Обслуживается МВС-1000М двумя основными компонентами:
- подсистемой удаленного управления и непрерывного мониторинга;

• подсистемой коллективного доступа.

В летнем списке Top 500 2004 года система МВС-1000М заняла 391-ю позицию с пиковой производительностью 1024 GFlops и максимально показанной на тесте LINPACK 734 GFlops.

1.2.1.4. МВС-15000

В настоящий момент в МСЦ РАН система МВС-1000М заменена на самый мощный суперкомпьютер России МВС-15000 (согласно последней редакции списка Top 50 стран СНГ – http://supercomputers.ru/index.php).

Аппаратная конфигурация вычислительных узлов МВС-15000 включает в себя:

• 2 процессора IBM PowerPC 970 с тактовой частотой 2,2 GHz, кэш L1 96 Kb и кэш L2 512 Kb;
• 4 Gb оперативной памяти на узел;
• 40 Gb жесткий диск IDE;
• 2 встроенных адаптера Gigabit Ethernet;

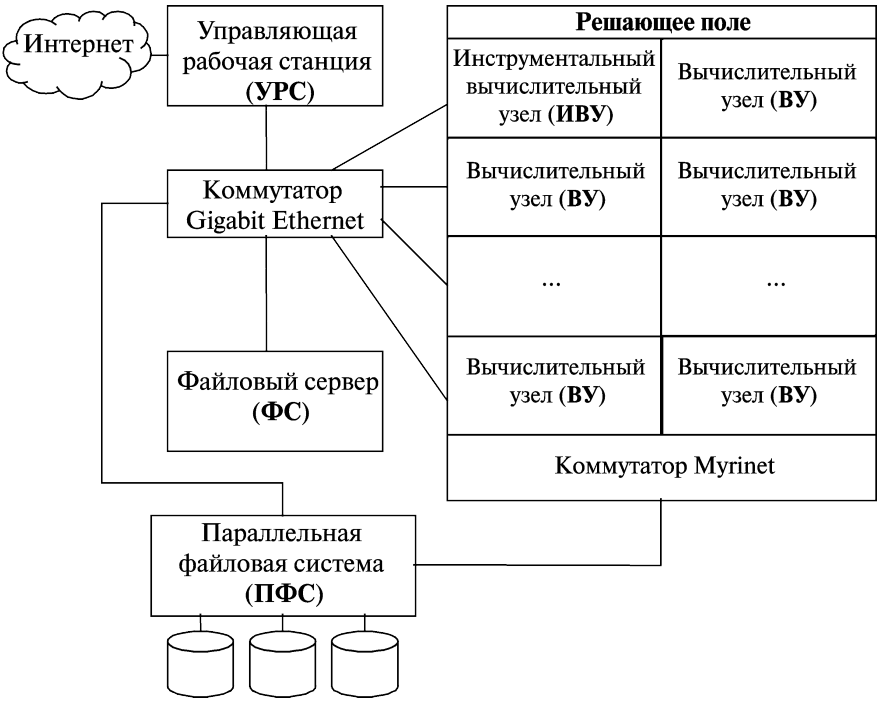


Рис. 1.2. Структурная схема системы МВС-15000

• адаптер Myrinet типа M3S-PCIXD-2-I.

Состав программного обеспечения МВС-15000:

• операционные системы SuSe Linux Enterprise Server версии 8 для платформ x86 и PowerPC;
• пакет GM 2.0.12 в качестве коммуникационной среды Myrinet;
• пакет MPICH-GM в качестве среды параллельного программирования;
• средства управления прохождением задач и их пакетной обработки.

На начало 2005 система МВС-15000 имела общее количество узлов 276 (552 процессора), пиковую производительность 4857,6 GFlops и максимальную (показанную на тесте LINPACK) производительность 3052 GFlops.

1.2.2. Кластеры

Кластер – группа компьютеров, объединенных в локальную вычислительную сеть (ЛВС) и способных работать в качестве единого вычислительного ресурса. Дополнительно предполагается, что для кластера обеспечивается более высокая надежность и эффективность, нежели для ЛВС, и существенно более низкая стоимость в сравнении с другими типами параллельных вычислительных систем (за счет использования типовых аппаратных и программных решений).

Исчисление истории кластеров можно начать от первого проекта, в котором одной из основных целей являлось установление связи между компьютерами, – проекта ARPANET[1]. Именно тогда были заложены первые, оказавшиеся фундаментальными, принципы, приведшие впоследствии к созданию локальных и глобальных вычислительных сетей и, конечно же, всемирной глобальной компьютерной сети Интернет. Правда, с момента ввода в действие сети ARPANET до появления первого кластера должно было пройти более двадцати лет.

Эти годы вместили в себя гигантский скачок в развитии аппаратной базы, появление и воцарение на рынке микропроцессоров и персональных компьютеров, накопление критической массы идей и методов параллельного программирования, что привело, в конечном счете, к решению извечной проблемы уникальности каждой параллельной вычислительной установки – разработке стандартов на создание параллельных программ для систем с общей и распределенной памятью. Добавим к этому дороговизну имевшихся на тот момент решений в области высокопроизводительных си-

[1] ARPANET – проект Агентства передовых исследовательских проектов Министерства обороны США (Defense Advanced Research Projects Agency, DARPA) по созданию компьютерной сети с целью проведения экспериментов в области компьютерных коммуникаций, поддержания связи в условиях ядерного нападения, разработки концепции децентрализованного управления (1966–1969 гг.).

стем, предполагавших использование быстродействующих, а потому специфических компонентов. Также учтем непрерывное улучшение соотношения «цена/производительность» персональных компьютеров. В свете всех этих обстоятельств появление кластеров было неизбежным.

Преимущества нового подхода к созданию вычислительных систем большой мощности, получившие признание практически сразу после первого представления такой системы, со временем только возрастали, поддерживаемые непрерывным ростом производительности типовых компонентов.

В настоящее время в списке Top 500 самых высокопроизводительных систем кластеры составляют большую часть – 294 установки.

1.2.2.1. Кластер Beowulf

Первым в мире кластером, по-видимому, является кластер, созданный под руководством Томаса Стерлинга и Дона Бекера в научно-космическом центре NASA – Goddard Space Flight Center – летом 1994 года. Названный в честь героя скандинавской саги, обладавшего, по преданию, силой тридцати человек, кластер состоял из 16 компьютеров на базе процессоров 486DX4 с тактовой частотой 100 MHz. Каждый узел имел 16 Mb оперативной памяти. Связь узлов обеспечивалась тремя параллельно работавшими 10 Mbit/s сетевыми адаптерами. Кластер функционировал под управлением операционной системы Linux, использовал GNU-компилятор и поддерживал параллельные программы на основе MPI. Процессоры узлов кластера были слишком быстрыми по сравнению с пропускной способностью обычной сети Ethernet, поэтому для балансировки системы Дон Бекер переписал драйверы Ethernet под Linux для создания дублированных каналов и распределения сетевого трафика.

Идея «собери суперкомпьютер своими руками» быстро пришлась по вкусу, в первую очередь академическому сообществу. Использование типовых массово выпускающихся компонентов, как аппаратных, так и программных, вело к значительному уменьшению стоимости разработки и внедрения системы. Вместе с тем производительность получающегося вычислительного комплекса была вполне достаточной для решения существенного количества задач, требовавших большого объема вычислений. Системы класса «кластер Beowulf» стали появляться по всему миру.

Четыре годя спустя в Лос-Аламосской национальной лаборатории (США) астрофизик Майкл Уоррен и другие ученые из группы теоретической астрофизики построили суперкомпьютер Avalon, который представлял собой Linux-кластер на базе процессоров Alpha 21164A с тактовой частотой 533 MHz. Первоначально включавший 68 процессоров, позднее Avalon был расширен до 140. Каждый узел содержал 256 Mb оперативной

памяти, 3 Gb дисковой памяти, Fast Ethernet card. Общая стоимость проекта Avalon составила чуть более 300 тыс. долл.

На момент ввода в строй полной версии (осень 1998 года) с пиковой производительностью в 149 GFlops и показанной на тесте LINPACK производительностью 48,6 GFlops кластер занял 113-е место в списке Top 500.

В том же году на самой престижной конференции в области высокопроизводительных вычислений Supercomputing'98 создатели Avalon получили первую премию в номинации «наилучшее отношение цена/производительность».

В настоящее время под кластером типа Beowulf понимается система, которая состоит из одного серверного узла и одного или более клиентских узлов, соединенных при помощи Ethernet или некоторой другой сети. Это система, построенная из готовых серийно выпускающихся промышленных компонентов, на которых может работать ОС Linux, стандартных адаптеров Ethernet и коммутаторов. Она не содержит специфических аппаратных компонентов и легко воспроизводима. Серверный узел управляет всем кластером и является файл-сервером для клиентских узлов. Он также является консолью кластера и шлюзом во внешнюю сеть. Большие системы Beowulf могут иметь более одного серверного узла, а также, возможно, специализированные узлы, например консоли или станции мониторинга. В большинстве случаев клиентские узлы в Beowulf пассивны. Они конфигурируются и управляются серверными узлами и выполняют только то, что предписано серверным узлом.

1.2.2.2. Кластер AC3 Velocity Cluster

Кластер **AC3 Velocity Cluster**, установленный в Корнельском университете (США) (http://www.tc.cornell.edu), стал результатом совместной деятельности университета и консорциума AC3 (Advanced Cluster Computing Consortium), образованного компаниями Dell, Intel, Microsoft, Giganet и еще 15 производителями ПО с целью интеграции различных технологий для создания кластерных архитектур для учебных и государственных учреждений.

Состав кластера:

- 64 четырехпроцессорных сервера Dell PowerEdge 6350 на базе Intel Pentium III Xeon 500 MHz, 4 GB RAM, 54 GB HDD, 100 Mbit Ethernet card;
- 1 восьмипроцессорный сервер Dell PowerEdge 6350 на базе Intel Pentium III Xeon 550 MHz, 8 GB RAM, 36 GB HDD, 100 Mbit Ethernet card.

Четырехпроцессорные серверы смонтированы по восемь штук на стойку и работают под управлением ОС Microsoft Windows NT 4.0 Server Enterprise Edition. Между серверами установлено соединение на скорости 100 Мбайт/с через Cluster Switch компании Giganet.

Задания в кластере управляются с помощью Cluster ConNTroller, созданного в Корнельском университете. Пиковая производительность AC3 Velocity составляет 122 GFlops при стоимости в 4 – 5 раз меньше, чем у суперкомпьютеров с аналогичными показателями.

На момент ввода в строй (лето 2000 года) кластер с показателем производительности на тесте LINPACK в 47 GFlops занимал 381-ю строку списка Top 500.

1.2.2.3. Кластер NCSA NT Supercluster

В 2000 году в Национальном центре суперкомпьютерных технологий (NCSA – National Center for Supercomputing Applications) на основе рабочих станций Hewlett-Packard Kayak XU PC workstation (http://www.hp.com/desktops/kayak/) был собран еще один кластер, для которого в качестве операционной системы была выбрана ОС Microsoft Windows. Недолго думая, разработчики окрестили его **NT Supercluster** (http://archive.ncsa.uiuc.edu/SCD/Hardware/NTCluster/).

На момент ввода в строй кластер с показателем производительности на тесте LINPACK в 62 GFlops и пиковой производительностью в 140 GFlops занимал 207-ю строку списка Top 500.

Кластер построен из 38 двупроцессорных серверов на базе Intel Pentium III Xeon 550 MHz, 1 Gb RAM, 7.5 Gb HDD, 100 Mbit Ethernet card.

Связь между узлами основана на сети Myrinet (http://www.myri.com/myrinet/index.html).

Программное обеспечение кластера:

- операционная система – Microsoft Windows NT 4.0;
- компиляторы – Fortran77, C/C++;
- уровень передачи сообщений основан на HPVM (http://www-csag.ucsd.edu/projects/clusters.html).

1.2.2.4. Кластер Thunder

В настоящий момент число систем, собранных на основе процессоров корпорации Intel и представленных в списке Top 500, составляет 318. Самый мощный суперкомпьютер, представляющий собой кластер на основе Intel Itanium2, установлен в Ливерморской национальной лаборатории (США).

Аппаратная конфигурация кластера Thunder (http://www.llnl.gov/linux/thunder/):

- 1024 сервера, по 4 процессора Intel Itanium 1.4 GHz в каждом;
- 8 Gb оперативной памяти на узел;
- общая емкость дисковой системы 150 Tb.

Программное обеспечение:

- операционная система CHAOS 2.0;
- среда параллельного программирования MPICH2;
- отладчик параллельных программ TotalView;
- Intel и GNU Fortran, C/C++ компиляторы.

В данное время кластер Thunder занимает 5-ю позицию списка Top 500 (на момент установки – лето 2004 года – занимал 2-ю строку) с пиковой производительностью 22938 GFlops и максимально показанной на тесте LINPACK 19940 Gflops.

1.2.3. Высокопроизводительный вычислительный кластер ННГУ

В качестве следующего примера рассмотрим вычислительный кластер Нижегородского университета, оборудование для которого было передано в рамках Академической программы Интел в 2001 г. В состав кластера входят (см. рис. 1.3):

- 2 вычислительных сервера, каждый из которых имеет 4 процессора Intel Pentium III 700 MHZ, 512 MB RAM, 10 GB HDD, 1 Gbit Ethernet card;
- 12 вычислительных серверов, каждый из которых имеет 2 процессора Intel Pentium III 1000 MHZ, 256 MB RAM, 10 GB HDD, 1 Gbit Ethernet card;
- 12 рабочих станций на базе процессора Intel Pentium 4 1300 MHZ, 256 MB RAM, 10 GB HDD, 10/100 Fast Ethernet card.

Следует отметить, что в результате передачи подобного оборудования Нижегородский госуниверситет оказался первым вузом в Восточной Европе, оснащенным ПК на базе новейшего процессора Intel®Pentium®4. Подобное достижение является дополнительным подтверждением складывающегося плодотворного сотрудничества ННГУ и корпорации Интел.

Важной отличительной особенностью кластера является его неоднородность (*гетерогенность*). В состав кластера входят рабочие места, оснащенные процессорами Intel Pentium 4 и соединенные относительно медленной сетью (100 Мбит), а также вычислительные 2- и 4-процессорные серверы, обмен данными между которыми выполняется при помощи быстрых каналов передачи данных (1000 Мбит). В результате кластер может использоваться не только для решения сложных вычислительно-трудоемких задач, но также и для проведения различных экспериментов по исследованию многопроцессорных кластерных систем и параллельных методов решения научно-технических задач.

В качестве системной платформы для построения кластера выбраны современные операционные системы семейства Microsoft Windows (для

проведения отдельных экспериментов имеется возможность использования ОС Unix). Такой выбор определяется рядом причин:

- операционные системы семейства Microsoft Windows (так же как и ОС Unix) широко используются для построения кластеров; причем если раньше применение ОС Unix для этих целей было преобладающим системным решением, то в настоящее время тенденцией является увеличение числа создаваемых кластеров под управлением ОС Microsoft Windows (см., например, www.tc.cornell.edu/ac3/, www.windowclusters.org и др.);
- разработка прикладного программного обеспечения выполняется преимущественно с использованием ОС Microsoft Windows;
- корпорация Microsoft проявила заинтересованность в создании подобного кластера и передала в ННГУ для поддержки работ все необходимое программное обеспечение (ОС MS Windows 2000 Professional, ОС MS Windows 2000 Advanced Server и др.).

В результате принятых решений программное обеспечение на момент установки кластера являлось следующим:

- вычислительные серверы работают под управлением ОС Microsoft® Windows® 2000 Advanced Server; на рабочих местах разработчиков установлена ОС Microsoft® Windows® 2000 Professional;
- в качестве сред разработки применяются Microsoft® Visual Studio 6.0; для выполнения исследовательских экспериментов возможно использование компилятора Intel® C++ Compiler 5.0, встраиваемого в среду Microsoft® Visual Studio;

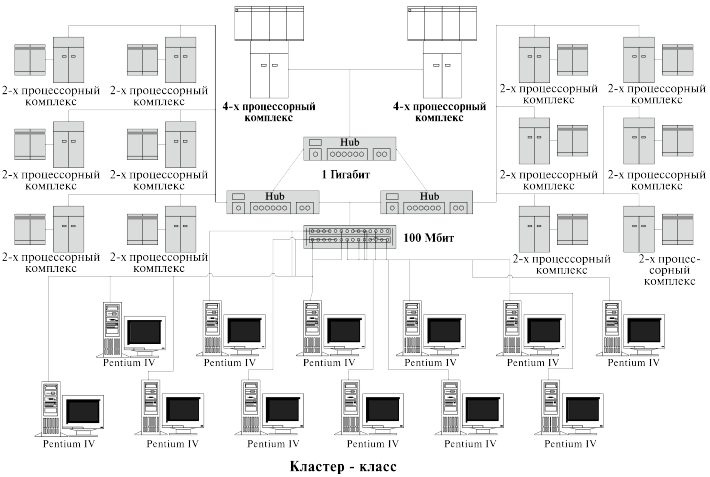


Рис. 1.3. Структура вычислительного кластера Нижегородского университета

- на рабочих местах разработчиков установлены библиотеки:
  - Plapack 3.0 (см. www.cs.utexas.edu/users/plapack);
  - MKL (см. developer.intel.com/software/products/mkl/index.htm);
- в качестве средств передачи данных между процессорами установлены две реализации стандарта MPI:
  - Argonne MPICH (www.unix.mcs.anl.gov/mpi/MPICH/);
  - MP-MPICH (www.lfbs.rwth-aachen.de/~joachim/MP-MPICH.html);
- в опытной эксплуатации находится система разработки параллельных программ DVM (см. www.keldysh.ru/dvm/).

В 2006 году в рамках инновационной образовательной программы Нижегородского университета Приоритетного национального проекта «Образование» была выполнена модернизация вычислительного кластера ННГУ, в результате пиковая производительность кластера была доведена до 3000 Gflops.

1.3. Классификация вычислительных систем

Одним из наиболее распространенных способов классификации ЭВМ является *систематика Флинна* (Flynn), в рамках которой основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия последовательностей (*потоков*) выполняемых команд и обрабатываемых данных. При таком подходе различают следующие основные типы систем (см. [2, 31, 59]):

- **SISD** (Single Instruction, Single Data) – системы, в которых существует одиночный поток команд и одиночный поток данных. К такому типу можно отнести обычные последовательные ЭВМ;
- **SIMD** (Single Instruction, Multiple Data) – системы с одиночным потоком команд и множественным потоком данных. Подобный класс составляют многопроцессорные вычислительные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов; такой архитектурой обладают, например, многопроцессорные системы с единым устройством управления. Этот подход широко использовался в предшествующие годы (системы ILLIAC IV или CM-1 компании Thinking Machines), в последнее время его применение ограничено, в основном, созданием специализированных систем;
- **MISD** (Multiple Instruction, Single Data) – системы, в которых существует множественный поток команд и одиночный поток данных. Относительно этого типа систем нет единого мнения: ряд специалистов считает, что примеров конкретных ЭВМ, соответствующих данному

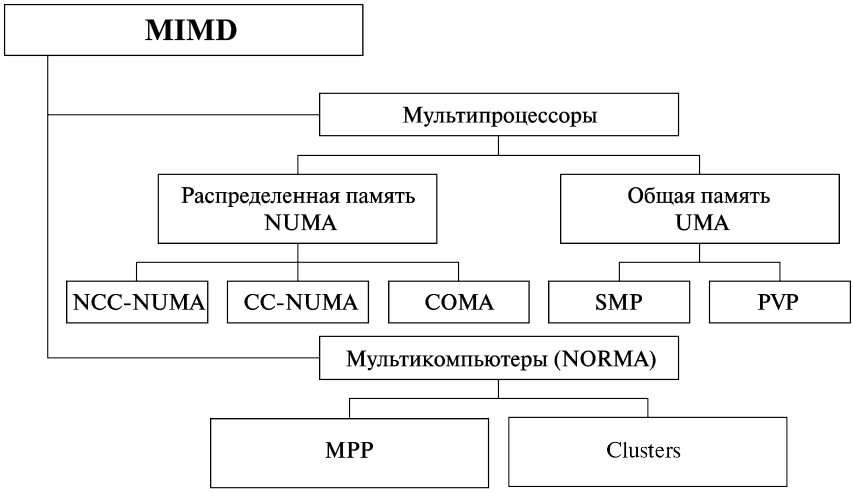


Рис. 1.4. Классификация многопроцессорных вычислительных систем

типу вычислительных систем, не существует и введение подобного класса предпринимается для полноты классификации; другие же относят к данному типу, например, *систолические вычислительные системы* (см. [51, 52]) или системы с конвейерной обработкой данных;

- **MIMD** (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных. К подобному классу относится большинство параллельных многопроцессорных вычислительных систем.

Следует отметить, что хотя систематика Флинна широко используется при конкретизации типов компьютерных систем, такая классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разнородность) оказываются отнесены к одной группе MIMD. Как результат, многими исследователями предпринимались неоднократные попытки детализации систематики Флинна. Так, например, для класса MIMD предложена практически общепризнанная структурная схема (см. [24,75]), в которой дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах (см. рис. 1.4). Такой подход позволяет различать два важных типа многопроцессорных систем – *multiprocessors* (*мультипроцессоры* или системы с общей разделяемой памятью) и *multicomputers* (*мультикомпьютеры* или системы с распределенной памятью).

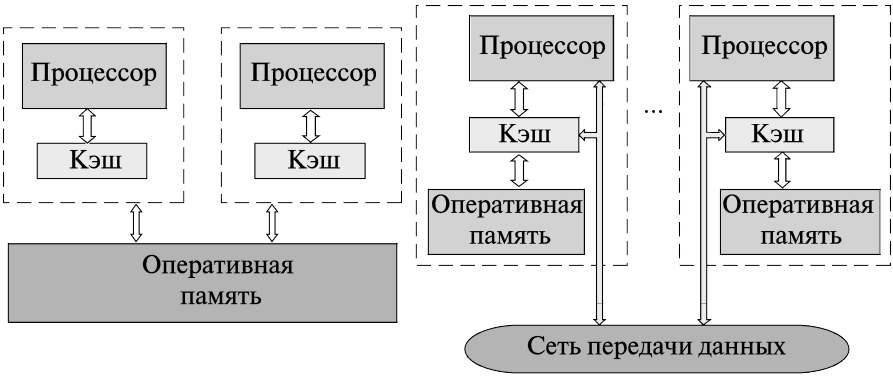


Рис. 1.5. Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным (б) доступом к памяти

1.3.1. Мультипроцессоры

Для дальнейшей систематики **мультипроцессоров** учитывается способ построения общей памяти. Первый возможный вариант – использование единой (централизованной) *общей памяти* (*shared memory*) (см. рис. 1.5 а). Такой подход обеспечивает *однородный доступ к памяти* (*uniform memory access* или *UMA*) и служит основой для построения *векторных параллельных процессоров* (*parallel vector processor* или *PVP*) и симметричных *мультипроцессоров* (*symmetric multiprocessor* или *SMP*). Среди примеров первой группы — суперкомпьютер Cray T90, ко второй группе относятся IBM eServer, Sun StarFire, HP Superdome, SGI Origin и др.

Одной из основных проблем, которые возникают при организации параллельных вычислений на такого типа системах, является доступ с разных процессоров к общим данным и обеспечение, в связи с этим, *однозначности* (*когерентности*) *содержимого разных кэшей* (*cache coherence problem*). Дело в том, что при наличии общих данных копии значений одних и тех же переменных могут оказаться в кэшах разных процессоров. Если в такой ситуации (при наличии копий общих данных) один из процессоров выполнит изменение значения разделяемой переменной, то значения копий в кэшах других процессоров окажутся не соответствующими действительности и их использование приведет к некорректности вычислений. Обеспечение однозначности кэшей обычно реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к пере-

менной потребует обязательного обращения к основной памяти. Следует отметить, что необходимость обеспечения когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при параллельных вычислениях приводит к необходимости *синхронизации взаимодействия* одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить *взаимоисключение* (*mutual exclusion*), чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти) (см. рис. 1.5 б). Такой подход именуется неоднородным доступом к памяти (*non-uniform memory access* или *NUMA*). Среди систем с таким типом памяти выделяют:

- системы, в которых для представления данных используется только локальная кэш-память имеющихся процессоров (*cache-only memory architecture* или *COMA*); примерами являются KSR-1 и DDM;
- системы, в которых обеспечивается когерентность локальных кэшей разных процессоров (*cache-coherent NUMA* или *CC-NUMA*); среди таких систем: SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000;
- системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (*non-cache coherent NUMA* или *NCC-NUMA*); например, система Cray T3E.

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

1.3.2. Мультикомпьютеры

Мультикомпьютеры (многопроцессорные системы с распределенной памятью) уже не обеспечивают общего доступа ко всей имеющейся в системах памяти (*no-remote memory access* или *NORMA*) (см. рис. 1.6). При

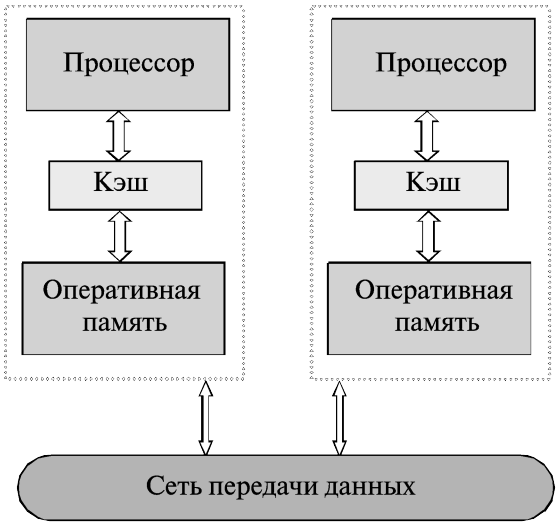


Рис. 1.6. Архитектура многопроцессорных систем с распределенной памятью

всей схожести подобной архитектуры с системами с распределенной общей памятью (рис. 1.5б), мультикомпьютеры имеют принципиальное отличие: каждый процессор системы может использовать только свою локальную память, в то время как для доступа к данным, располагаемым на других процессорах, необходимо явно выполнить *операции передачи сообщений* (*message passing operations*). Данный подход применяется при построении двух важных типов многопроцессорных вычислительных систем (см. рис. 1.4) — *массивно-параллельных систем* (*massively parallel processor* или *MPP*) и *кластеров* (*clusters*). Среди представителей первого типа систем — IBM RS/6000 SP2, Intel PARAGON, ASCI Red, транспьютерные системы Parsytec и др.; примерами кластеров являются, например, системы AC3 Velocity и NCSA NT Supercluster.

Следует отметить чрезвычайно быстрое развитие многопроцессорных вычислительных систем **кластерного типа** – общая характеристика данного подхода приведена, например, в обзоре [19]. Под *кластером* обычно понимается (см. [60,76]) множество отдельных компьютеров, объединенных в сеть, для которых при помощи специальных аппаратно-программных средств обеспечивается возможность унифицированного управления (*single system image*), надежного функционирования (*availability*) и эффективного использования (*performance*). Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров либо же сконструированы из типовых компьютерных элемен-

тов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени устранить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров (*lowly parallel processing*). Тем самым, для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов (*coarse granularity*), что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых данных между компьютерами кластера. Вместе с этим следует отметить, что организация взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным временным задержкам, и это накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

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

В завершение обсуждаемой темы можно отметить, что существуют и другие способы классификации вычислительных систем (достаточно полный обзор подходов представлен в [2,45,59], см. также материалы сайта http://www.parallel.ru/computers/taxonomy/). При рассмотрении темы параллельных вычислений рекомендуется обратить внимание на способ структурной нотации для описания архитектуры ЭВМ, позволяющий с высокой степенью точности описать многие характерные особенности компьютерных систем.

1.4. Характеристика типовых схем коммуникации в многопроцессорных вычислительных системах

В мультикомпьютерах для организации взаимодействия, синхронизации и взаимоисключения параллельно выполняемых процессов используется передача данных между процессорами вычислительной среды. Временные задержки при передаче данных по линиям связи могут оказаться существенными (по сравнению с быстродействием процессоров), и, как результат, коммуникационная трудоемкость алгоритма оказывает заметное влияние на выбор параллельных способов решения задач.

1.4.1. Примеры топологий сети передачи данных

Структура линий коммутации между процессорами вычислительной системы (*топология сети передачи данных*) определяется, как правило, с учетом возможностей эффективной технической реализации. Немаловажную роль при выборе структуры сети играет и анализ интенсивности информационных потоков при параллельном решении наиболее распространенных вычислительных задач. К числу типовых топологий обычно относят следующие схемы коммуникации процессоров (см. рис. 1.7):

- **полный граф** (*completely-connected graph* или *clique*) – система, в которой между любой парой процессоров существует прямая линия связи. Такая топология обеспечивает минимальные затраты при передаче данных, однако является сложно реализуемой при большом количестве процессоров;
- **линейка** (*linear array* или *farm*) – система, в которой все процессоры перенумерованы по порядку и каждый процессор, кроме первого и последнего, имеет линии связи только с двумя соседними (с предыдущим и последующим) процессорами. Такая схема является, с одной стороны, просто реализуемой, с другой стороны, соответствует структуре передачи данных при решении многих вычислительных задач (например, при организации конвейерных вычислений);
- **кольцо** (*ring*) – данная топология получается из линейки процессоров соединением первого и последнего процессоров линейки;
- **звезда** (*star*) – система, в которой все процессоры имеют линии связи с некоторым управляющим процессором. Данная топология является эффективной, например, при организации централизованных схем параллельных вычислений;
- **решетка** (*mesh*) – система, в которой граф линий связи образует прямоугольную сетку (обычно двух- или трехмерную). Подобная топология может быть достаточно просто реализована и, кроме того, эффективно использована при параллельном выполнении многих численных алгоритмов (например, при реализации методов анализа математических моделей, описываемых дифференциальными уравнениями в частных производных);
- **гиперкуб** (*hypercube*) – данная топология представляет собой частный случай структуры решетки, когда по каждой размерности сетки имеется только два процессора (т.е. гиперкуб содержит 2^N процессоров при размерности N). Такой вариант организации сети передачи данных достаточно широко распространен на практике и характеризуется следующим рядом отличительных признаков:

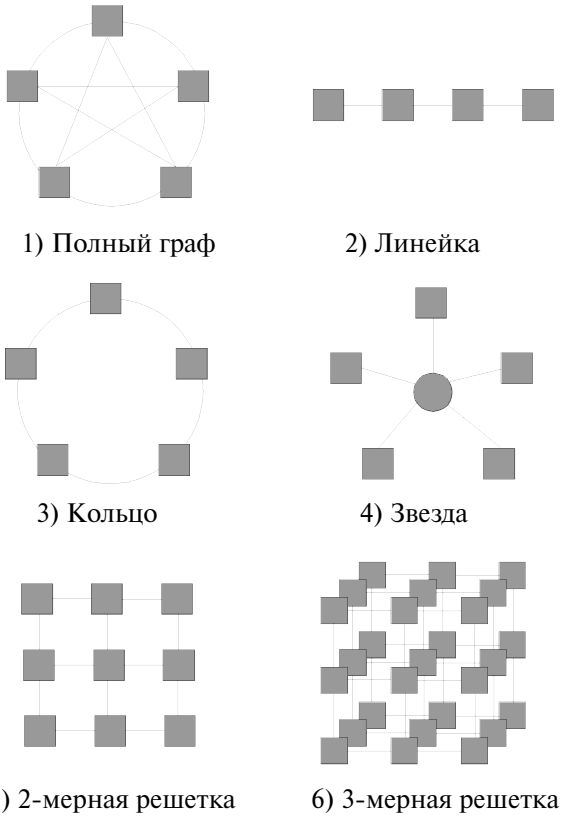


Рис. 1.7. Примеры топологий многопроцессорных вычислительных систем

— 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 различных таких разбиений);
— кратчайший путь между двумя любыми процессорами имеет длину, совпадающую с количеством различающихся битовых значений в номерах процессоров (данная величина известна как *расстояние Хэмминга*).

Дополнительная информация по топологиям многопроцессорных вычислительных систем может быть получена, например, в [2, 11, 24, 28, 45, 59, 76]. При рассмотрении вопроса следует учесть, что схема линий переда-

чи данных может реализовываться на аппаратном уровне, а может быть обеспечена на основе имеющейся физической топологии при помощи соответствующего программного обеспечения. Введение виртуальных (программно-реализуемых) топологий способствует мобильности разрабатываемых параллельных программ и снижает затраты на программирование.

1.4.2. Топология сети вычислительных кластеров

Для построения кластерной системы во многих случаях используют коммутатор (*switch*), через который процессоры кластера соединяются между собой. В этом случае топология сети кластера представляет собой полный граф (рис. 1.7), в соответствии с которым передача данных может быть организована между любыми двумя процессорами сети. При этом, однако, одновременность выполнения нескольких коммуникационных операций является ограниченной – *в любой момент времени каждый процессор может принимать участие только в одной операции приема-передачи данных*. Как результат, параллельно могут выполняться только те коммуникационные операции, в которых взаимодействующие пары процессоров не пересекаются между собой.

1.4.3. Характеристики топологии сети

В качестве основных характеристик топологии сети передачи данных наиболее широко используется следующий ряд показателей:

- *диаметр* – показатель, определяемый как максимальное расстояние между двумя процессорами сети (под расстоянием обычно понимается величина кратчайшего пути между процессорами). Эта величина может характеризовать максимально необходимое время для передачи данных между процессорами, поскольку время передачи обычно прямо пропорционально длине пути;
- *связность* (*connectivity*) – показатель, характеризующий наличие разных маршрутов передачи данных между процессорами сети. Конкретный вид данного показателя может быть определен, например, как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области;
- *ширина бинарного деления* (*bisection width*) – показатель, определяемый как минимальное количество дуг, которое надо удалить для разделения сети передачи данных на две несвязные области одинакового размера;
- *стоимость* – показатель, который может быть определен, например, как общее количество линий передачи данных в многопроцессорной вычислительной системе.

Для сравнения в *таблице 1.1* приводятся значения перечисленных показателей для различных топологий сети передачи данных.

Таблица 1.1. Характеристики топологий сети передачи данных (p – количество процессоров)

Топология	Диаметр	Ширина бисекции	Связность	Стоимость
Полный граф	1	$p^2/4$	p–1	p(p–1)/2
Звезда	2	1	1	p–1
Полное двоичное дерево	2log((p+1)/2)	1	1	p–1
Линейка	p–1	1	1	p–1
Кольцо	$\lfloor p/2 \rfloor$	2	2	p
Решетка N=2	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
Решетка-тор N=2	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	2p
Гиперкуб	log p	p/2	log p	(p log p)/2

1.5. Характеристика системных платформ для построения кластеров

В качестве системной платформы для построения кластеров используют обе наиболее распространенные в настоящий момент операционные системы Unix и Microsoft Windows. Далее в пособии подробно будет рассмотрено решение на основе ОС семейства Microsoft Windows; характеристика подхода на базе ОС Unix может быть получена, например, в [71].

Microsoft Compute Cluster Server 2003 (CCS) представляет собой интегрированную платформу для поддержки высокопроизводительных вычислений на кластерных системах. CCS состоит из операционной системы Microsoft Windows Server 2003 и Microsoft Compute Cluster Pack (CCP) – набора интерфейсов, утилит и инфраструктуры управления. Вместе с CCP поставляется SDK, содержащий необходимые инструменты разработки программ для CCS, включая собственную реализацию MPI (Microsoft MPI). Кроме того, к Microsoft Compute Cluster Server 2003 логически примыкает Microsoft Visual Studio 2005 — интегрированная среда разработки (IDE), содержащая компилятор и отладчик программ, разработанных с использованием технологий MPI и OpenMP.

В качестве вычислительных узлов кластера могут быть применены 64-битные процессоры семейства x86 с, как минимум, 512 Мб оперативной памяти и 4 Гб свободного дискового пространства.

На вычислительных узлах кластера должна быть установлена операционная система Microsoft Windows Server 2003 (Standard, Enterprise или Compute Cluster Edition).

В состав CCP входит Microsoft MPI – версия реализации стандарта MPI 2 от Argonne National Labs. MS MPI совместима с MPICH 2 и поддерживает полнофункциональный API с более чем 160 функциями. MS MPI в Windows Compute Cluster Server 2003 задействует WinSock Direct протокол для наилучшей производительности и эффективного использования центрального процессора. MS MPI может использовать любое Ethernet-соединение, поддерживаемое Windows Server 2003, а также такие соединения, как InfiniBand или Myrinet, с применением WinSock Direct драйверов, поставляемых производителями аппаратного обеспечения. MS MPI поддерживает языки программирования C, Fortran 77 и Fortran 90, а Microsoft Visual Studio 2005 включает в себя параллельный отладчик, работающий с MS MPI. Разработчики могут запустить свое MPI-приложение на нескольких вычислительных узлах, и Visual Studio автоматически соединится с процессами на каждом узле, позволяя разработчику приостанавливать приложение и просматривать значения переменных в каждом процессе отдельно.

Кроме реализации MPI в состав CCP входит удобная система планирования заданий, позволяющая просматривать состояния всех запущенных задач, собирать статистику, назначать запуски программ на определенное время, завершать «зависшие» задачи и пр. В текущей версии работа возможна либо через графический интерфейс, либо через командную строку. В окончательной версии будет предусмотрена возможность обращения к системе и через другие интерфейсы: COM, web-сервис и др.

Windows Computer Cluster Server 2003 поддерживает 5 различных сетевых топологий, при этом каждый узел может иметь от 1 до 3 сетевых карточек. Правильный выбор используемой топологии необходим для оптимального функционирования вычислительного кластера.

1.6. Краткий обзор лекции

В лекции приводится общая характеристика способов организации параллельных вычислений и дается различие между многозадачным, параллельным и распределенным режимами выполнения программ. Для демонстрации возможных подходов рассматривается ряд примеров параллельных вычислительных систем и отмечается существенное разнообразие вариантов построения параллельных систем.

Многообразие компьютерных вычислительных систем приводит к необходимости их классификации. В лекции дается описание одного из наиболее известных способов – систематики Флинна, в основу которой положено понятие потоков команд и данных. Данная классификация является

достаточно простой и понятной, однако в рамках такого подхода почти все многопроцессорные вычислительные системы попадают в одну группу – *класс MIMD*. С целью дальнейшего разделения возможных типов систем в лекции приводится также широко используемая структуризация класса многопроцессорных вычислительных систем, что позволяет выделить две важные группы систем с общей разделяемой и распределенной памятью – *мультипроцессоры* и *мультикомпьютеры*. Наиболее известные примеры систем первой группы — *векторные параллельные процессоры* (*parallel vector processor* или *PVP*) и *симметричные мультипроцессоры* (*symmetric multiprocessor* или *SMP*). К мультикомпьютерам относятся *массивно-параллельные системы* (*massively parallel processor* или *MPP*) и *кластеры* (*clusters*).

Далее в лекции обращается внимание на характеристику сетей передачи данных в многопроцессорных вычислительных системах. Приводятся примеры топологий сетей, отмечаются особенности организации сетей передачи данных в кластерах и обсуждаются параметры топологий, существенно влияющие на коммуникационную сложность методов параллельных вычислений.

В завершение лекции дается общая характеристика системных платформ для построения кластеров.

1.7. Обзор литературы

Дополнительная информация об архитектуре параллельных вычислительных систем может быть получена, например, из [2, 11, 14, 28, 45, 59]; полезная информация содержится также в [24, 76].

В качестве обзора возможных топологий сетей передачи данных в многопроцессорных системах и технологий для их реализации может быть рекомендована, например, работа [29].

Подробное рассмотрение вопросов, связанных с построением и использованием кластерных вычислительных систем, проводится в [24, 76]. Практические рекомендации по построению кластеров для разных системных платформ могут быть найдены в [70, 71].

1.8. Контрольные вопросы

1. В чем заключаются основные способы достижения параллелизма?
2. В чем могут состоять различия параллельных вычислительных систем?
3. Что положено в основу классификации Флинна?
4. В чем состоит принцип разделения многопроцессорных систем на мультипроцессоры и мультикомпьютеры?
5. Какие классы систем известны для мультипроцессоров?

6. В чем состоят положительные и отрицательные стороны симметричных мультипроцессоров?
7. Какие классы систем известны для мультикомпьютеров?
8. В чем состоят положительные и отрицательные стороны кластерных систем?
9. Какие топологии сетей передачи данных наиболее широко используются при построении многопроцессорных систем?
10. В чем состоят особенности сетей передачи данных для кластеров?
11. Каковы основные характеристики сетей передачи данных?
12. Какие системные платформы могут быть использованы для построения кластеров?

1.9. Задачи и упражнения

1. Приведите дополнительные примеры параллельных вычислительных систем.
2. Рассмотрите дополнительные способы классификации компьютерных систем.
3. Рассмотрите способы обеспечения когерентности кэшей в системах с общей разделяемой памятью.
4. Подготовьте обзор программных библиотек, обеспечивающих выполнение операций передачи данных для систем с распределенной памятью.
5. Рассмотрите топологию сети передачи данных в виде двоичного дерева.
6. Выделите эффективно реализуемые классы задач для каждого типа топологий сети передачи данных.

Лекция 2. Моделирование и анализ параллельных вычислений

В лекции описывается модель вычислений в виде графа «операции – операнды». Приводятся основные показатели качества параллельных методов — ускорение (speedup), эффективность (efficiency), стоимость (cost) и масштабируемость (scalability) вычислений. Введенные понятия демонстрируются на примере учебной задачи нахождения частных сумм последовательности числовых значений.

Ключевые слова: модель вычислений, граф «операции – операнды», расписание, время выполнения, ускорение, эффективность, стоимость, масштабируемость, паракомпьютер, закон Амдаля, закон Густавсона – Барсиса, функция изоэффективности, каскадная схема.

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

В данной лекции рассматривается модель вычислений в виде графа «операции – операнды», которая может использоваться для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач, и приводятся оценки эффективности максимально возможного параллелизма, которые могут быть получены в результате анализа имеющихся моделей вычислений. Примеры применения излагаемого теоретического материала приводятся в лекциях 6 – 11 настоящего учебного пособия при изучении параллельных методов решения ряда сложных вычислительно трудоемких задач.

2.1. Модель вычислений в виде графа «операции – операнды»

Для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач может быть использована модель в

виде графа «операции – операнды» (см., например, [2, 22]). Для уменьшения сложности излагаемого материала при построении модели будет предполагаться, что время выполнения любых вычислительных операций является одинаковым и равняется 1 (в тех или иных единицах измерения). Кроме того, принимается, что передача данных между вычислительными устройствами выполняется мгновенно без каких-либо затрат времени (что может быть справедливо, например, при наличии общей разделяемой памяти в параллельной вычислительной системе). Анализ коммуникационной трудоемкости параллельных алгоритмов приводится в следующей лекции.

Представим множество операций, выполняемых в исследуемом алгоритме решения вычислительной задачи, и существующие между операциями информационные зависимости в виде ациклического ориентированного графа

$G = (V, R)$,

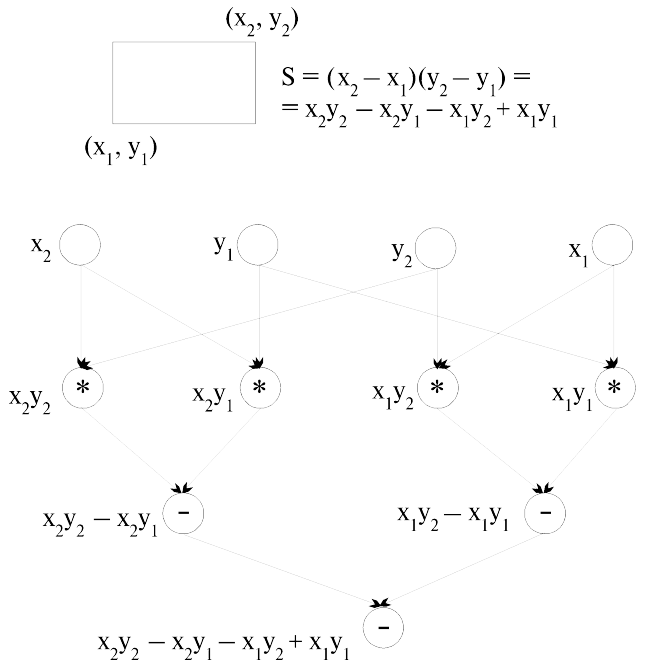


Рис. 2.1. Пример вычислительной модели алгоритма в виде графа «операции – операнды»

где $V = \{1,...,|V|\}$ есть множество вершин графа, представляющих выполняемые операции алгоритма, а R есть множество дуг графа (при этом дуга $r = (i, j)$ принадлежит графу только в том случае, если операция j использует результат выполнения операции i). Для примера на рис. 2.1 показан граф алгоритма вычисления площади прямоугольника, заданного координатами двух противолежащих углов. Как можно заметить по приведенному примеру, для выполнения выбранного алгоритма решения задачи могут быть использованы разные схемы вычислений и построены соответственно разные вычислительные модели. Как будет показано далее, разные схемы вычислений обладают различными возможностями для распараллеливания и, тем самым, при построении модели вычислений может быть поставлена задача выбора наиболее подходящей для параллельного исполнения вычислительной схемы алгоритма.

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

2.2. Описание схемы параллельного выполнения алгоритма

Операции алгоритма, между которыми нет пути в рамках выбранной схемы вычислений, могут быть выполнены параллельно (для вычислительной схемы на рис. 2.1, например, параллельно могут быть реализованы сначала все операции умножения, а затем первые две операции вычитания). Возможный способ описания параллельного выполнения алгоритма может состоять в следующем (см., например, [2, 22]).

Пусть p есть количество процессоров, используемых для выполнения алгоритма. Тогда для параллельного выполнения вычислений необходимо задать множество (*расписание*)

$$H_p = \{(i,P_i,t_i): i \in V\},$$

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

1) $\forall i, j \in V : t_i = t_j \Rightarrow P_i \neq P_j$, т.е. один и тот же процессор не должен назначаться разным операциям в один и тот же момент;

2) $\forall(i,j)\in R \Rightarrow t_j \geq t_i + 1$, т.е. к назначаемому моменту выполнения операции все необходимые данные уже должны быть вычислены.

2.3. Определение времени выполнения параллельного алгоритма

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

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

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

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

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

$$T_p = \min_{G} T_p(G).$$

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

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

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

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

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

где $|\overline{V}|$, напомним, есть количество вершин вычислительной схемы без вершин ввода. Важно отметить, что если при определении оценки ограничиться рассмотрением только одного выбранного алгоритма решения задачи и использовать величину

$$T_1 = \min_G T_1(G),$$

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

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

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

Приведем без доказательства теоретические положения, характеризующие свойства оценок времени выполнения параллельного алгоритма (см. [22]).

Теорема 1. Минимально возможное время выполнения параллельного алгоритма определяется длиной максимального пути вычислительной схемы алгоритма, т.е.

$$T_\infty(G) = d(G).$$

Теорема 2. Пусть для некоторой вершины вывода в вычислительной схеме алгоритма существует путь из каждой вершины ввода. Кроме того, пусть входная степень вершин схемы (количество входящих дуг) не превышает 2. Тогда минимально возможное время выполнения параллельного алгоритма ограничено снизу значением

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

где n есть количество вершин ввода в схеме алгоритма.

Теорема 3. При уменьшении числа используемых процессоров время выполнения алгоритма увеличивается пропорционально величине уменьшения количества процессоров, т.е.

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

Теорема 4. Для любого количества используемых процессоров справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма

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

Теорема 5. Времени выполнения алгоритма, которое сопоставимо с минимально возможным временем T_∞, можно достичь при количестве процессоров порядка $p \sim T_1/T_\infty$, а именно,

$$p \geq T_1 / T_\infty \Rightarrow T_p \leq 2T_\infty .$$

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

$$p < T_1 / T_\infty \Rightarrow \frac{T_1}{p} \leq T_p \leq 2\frac{T_1}{p} .$$

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

1) при выборе вычислительной схемы алгоритма должен использоваться граф с минимально возможным диаметром (см. теорему 1);

2) для параллельного выполнения целесообразное количество процессоров определяется величиной $p \sim T_1/T_\infty$ (см. теорему 5);

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

Для вывода рекомендаций по формированию расписания по параллельному выполнению алгоритма приведем доказательство теоремы 4.

Доказательство теоремы 4. Пусть H_∞ есть расписание для достижения минимально возможного времени выполнения T_∞. Для каждой итерации τ, $0 \leqslant \tau \leqslant T_\infty$, выполнения расписания H_∞ обозначим через n_τ количество операций, выполняемых в ходе итерации τ. Расписание выполнения алгоритма с использованием p процессоров может быть построено следующим образом. Выполнение алгоритма разделим на T_∞ шагов; на каждом шаге τ следует выполнить все n_τ операций, которые выполнялись на итерации τ расписания H_∞. Эти операции могут быть выполнены не более чем за $\lceil n_\tau / p \rceil$ итераций при использовании p процессоров. Как результат, время выполнения алгоритма T_p может быть оценено следующим образом

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

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

2.4. Показатели эффективности параллельного алгоритма

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

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

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

Эффективность (efficiency) использования параллельным алгоритмом процессоров при решении задачи определяется соотношением

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

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

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

• При определенных обстоятельствах ускорение может оказаться больше числа используемых процессоров $S_p(n)>p$ — в этом случае говорят о существовании *сверхлинейного* (*superlinear*) ускорения. Несмотря на парадоксальность таких ситуаций (ускорение превышает число процессоров), на практике сверхлинейное ускорение может иметь место. Одной из причин такого явления может быть неодинаковость условий выполнения последовательной и параллельной программ. Например, при решении за-

дачи на одном процессоре оказывается недостаточно оперативной памяти для хранения всех обрабатываемых данных и тогда становится необходимым использование более медленной внешней памяти (в случае же использования нескольких процессоров оперативной памяти может оказаться достаточно за счет разделения данных между процессорами). Еще одной причиной сверхлинейного ускорения может быть нелинейный характер зависимости сложности решения задачи от объема обрабатываемых данных. Так, например, известный алгоритм пузырьковой сортировки характеризуется квадратичной зависимостью количества необходимых операций от числа упорядочиваемых данных. Как результат, при распределении сортируемого массива между процессорами может быть получено ускорение, превышающее число процессоров (более подробно данный пример рассматривается в лекции 9). Источником сверхлинейного ускорения может быть и различие вычислительных схем последовательного и параллельного методов,

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

При выборе надлежащего параллельного способа решения задачи может оказаться полезной оценка ***стоимости (cost)*** вычислений, определяемой как произведение времени параллельного решения задачи и числа используемых процессоров

$$C_p = p T_p.$$

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

Далее для иллюстрации введенных понятий в следующем пункте будет рассмотрен учебный пример решения задачи вычисления частных сумм для последовательности числовых значений. Кроме того, данные показатели будут использоваться для характеристики эффективности

всех рассматриваемых в лекциях 6 – 11 параллельных алгоритмов при решении ряда типовых задач вычислительной математики.

2.5. Учебный пример. Вычисление частных сумм последовательности числовых значений

Рассмотрим для демонстрации ряда проблем, возникающих при разработке параллельных методов вычислений, сравнительно простую *задачу нахождения частных сумм* последовательности числовых значений

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

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

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

$$S = \sum_{i=1}^{n} x_i .$$

2.5.1. Последовательный алгоритм суммирования

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

S=0,
S=S+x$_1$,...

Вычислительная схема данного алгоритма может быть представлена следующим образом (см. рис. 2.2):

$G_1=(V_1,R_1)$,

где $V_1=\{v_{01},...,v_{0n}, v_{11},...,v_{1n}\}$ есть множество операций (вершины $v_{01},...,v_{0n}$ обозначают операции ввода, каждая вершина v_{1i}, $1 \leqslant i \leqslant n$, соответствует прибавлению значения x_i к накапливаемой сумме S), а

$R_1=\{(v_{0i},v_{1i}),(v_{1i},v_{1i+1}), 1 \leqslant i \leqslant n-1\}$

есть множество дуг, определяющих информационные зависимости операций.

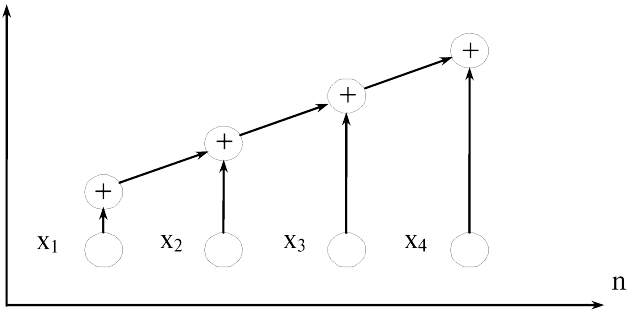


Рис. 2.2. Последовательная вычислительная схема алгоритма суммирования

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

2.5.2. Каскадная схема суммирования

Параллелизм алгоритма суммирования становится возможным только при ином способе построения процесса вычислений, основанном на использовании ассоциативности операции сложения. Получаемый новый вариант суммирования (известный в литературе как *каскадная схема*) состоит в следующем (см. рис. 2.3):

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

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

$$G_2=(V_2,R_2),$$

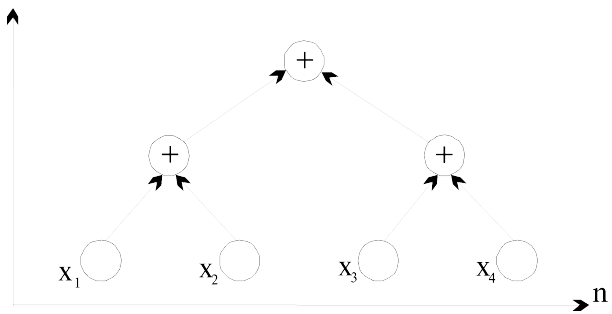


Рис. 2.3. Каскадная схема алгоритма суммирования

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

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

Как нетрудно оценить, количество итераций каскадной схемы оказывается равным величине

$$k=log_2n,$$

а общее количество операций суммирования

$$K_{посл}=n/2+n/4+...+1=n-1$$

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

$$K_{пар}=log_2n.$$

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

$$S_p=T_1/T_p=(n-1)/log_2n,$$
$$E_p=T_1/pT_p=(n-1)/(plog_2n)=(n-1)/((n/2)log_2n),$$

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

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

$$limE_p\to 0 \text{ при } n\to\infty.$$

2.5.3. Модифицированная каскадная схема

Получение асимптотически ненулевой эффективности может быть обеспечено, например, при использовании модифицированной каскадной схемы (см. [22]). Для упрощения построения оценок можно предположить $n=2^k$, $k=2^s$. Тогда в новом варианте каскадной схемы все вычисления производятся в два последовательно выполняемых этапа суммирования (см. рис. 2.4):

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

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

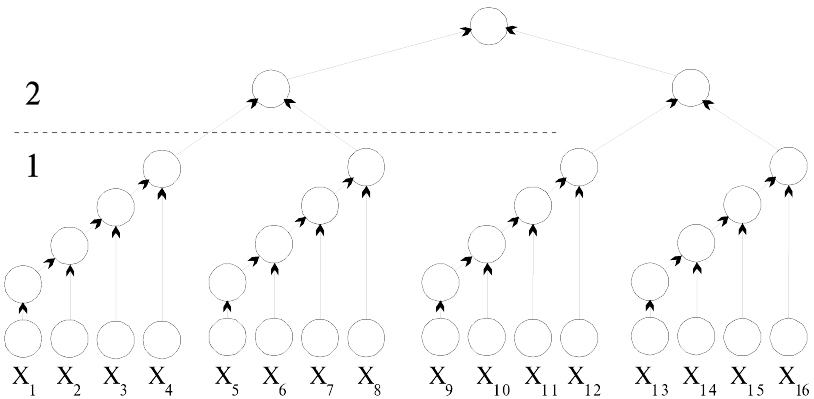


Рис. 2.4. Модифицированная каскадная схема суммирования

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

$$log_2(n/log_2 n) \leqslant log_2 n$$

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

$T_p=2log_2n$, $p=(n/log_2n)$.

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

$S_p=T_1/T_p=(n-1)/2log_2n$,
$E_p=T_1/pT_p=(n-1)/(2(n/log_2n)log_2n)=(n-1)/2n$.

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

$E_p=(n-1)/2n \geqslant 0{,}25$, $limE_p \to 0{,}5$ *при* $n \to \infty$.

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

$C_p=pT_p=(n/log_2n)(2log_2n)$

является пропорциональной времени выполнения последовательного алгоритма.

2.5.4. Вычисление всех частных сумм

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

$T_1=n$.

При параллельном исполнении применение каскадной схемы в явном виде не приводит к желаемым результатам; **достижение эффективного распараллеливания требует привлечения новых подходов (может быть, даже не имеющих аналогов при последовательном программировании) для разработки новых параллельно-ориентированных алгоритмов решения**

задач. Так, для рассматриваемой задачи нахождения всех частных сумм алгоритм, обеспечивающий получение результатов за $log_2 n$ параллельных операций (как и в случае вычисления общей суммы), может состоять в следующем (см. рис. 2.5, а также [22]):

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

• далее на каждой итерации суммирования i, $1 \leqslant i \leqslant log_2 n$, формируется вспомогательный вектор Q путем сдвига вправо вектора S на 2^{i-1} позиций (освобождающиеся при сдвиге позиции слева устанавливаются в нулевые значения); итерация алгоритма завершается параллельной операцией суммирования векторов S и Q.

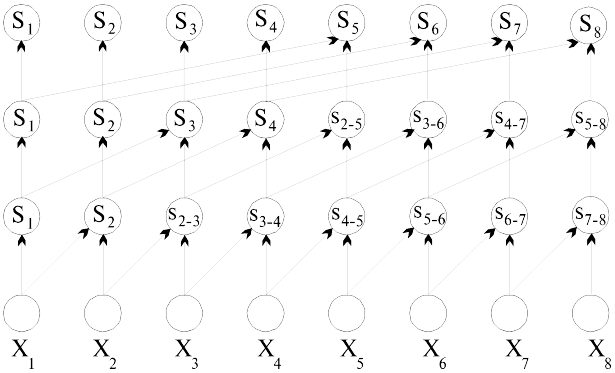


Рис. 2.5. Схема параллельного алгоритма вычисления всех частных сумм (величины S_{i-j} означают суммы значений от i до j элементов числовой последовательности)

Всего параллельный алгоритм выполняется за $log_2 n$ параллельных операций сложения. На каждой итерации алгоритма параллельно выполняются n скалярных операций сложения и, таким образом, общее количество скалярных операций определяется величиной

$$K_{пар} = n log_2 n$$

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

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

$S_p=T_1/T_p=n/log_2n,$
$E_p=T_1/pT_p=n/(plog_2n)=n/(nlog_2n)=1/log_2n.$

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

2.6. Оценка максимально достижимого параллелизма

Оценка качества параллельных вычислений предполагает знание наилучших (*максимально достижимых*) значений показателей ускорения и эффективности, однако получение идеальных величин $S_p=p$ для ускорения и $E_p=1$ для эффективности может быть обеспечено не для всех вычислительно трудоемких задач. Так, для рассматриваемого учебного примера в предыдущем пункте минимально достижимое время параллельного вычисления суммы числовых значений составляет log_2n. Определенное содействие в решении этой проблемы могут оказать теоретические утверждения, приведенные в начале данной лекции. В дополнение к ним рассмотрим еще ряд закономерностей, которые могут быть чрезвычайно полезны при построении оценок максимально достижимого параллелизма[1].

1. Закон Амдаля. Достижению максимального ускорения может препятствовать существование в выполняемых вычислениях последовательных расчетов, которые не могут быть распараллелены. Пусть f есть *доля последовательных вычислений* в применяемом алгоритме обработки данных, тогда в соответствии с *законом Амдаля* (*Amdahl*) ускорение процесса вычислений при использовании p процессоров ограничивается величиной

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

Так, например, при наличии всего 10% последовательных команд в выполняемых вычислениях эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных. В рассмотренном учебном примере вычисления суммы значений для каскадной схемы

[1] Как и ранее, при выводе закономерностей не будут учитываться издержки, связанные с организацией передачи сообщений – анализу коммуникационной сложности параллельных алгоритмов посвящена лекция 3.

доля последовательных расчетов составляет $f=log_2 n/n$ и, как результат, величина возможного ускорения ограничена оценкой $S^*=n/log_2 n$.

Закон Амдаля характеризует одну из самых серьезных проблем в области параллельного программирования (алгоритмов без определенной доли последовательных команд практически не существует). Однако часто доля последовательных действий характеризует не возможность параллельного решения задач, а последовательные свойства применяемых алгоритмов. Поэтому доля последовательных вычислений может быть существенно снижена при выборе более подходящих для распараллеливания методов.

Следует отметить также, что рассмотрение закона Амдаля происходит в предположении, что доля последовательных расчетов f является постоянной величиной и не зависит от параметра n, определяющего вычислительную сложность решаемой задачи. Однако для большого ряда задач доля $f=f(n)$ является убывающей функцией от n, и в этом случае ускорение для фиксированного числа процессоров может быть увеличено за счет увеличения вычислительной сложности решаемой задачи. Данное замечание может быть сформулировано как утверждение, что ускорение $S_p= S_p(n)$ является возрастающей функцией от параметра n (данное утверждение часто именуется *эффект Амдаля*). Так, например, для учебного примера вычисления суммы значений при использовании фиксированного числа процессоров p суммируемый набор данных может быть разделен на блоки размера n/p, для которых сначала параллельно могут быть вычислены частные суммы, а далее эти суммы можно сложить при помощи каскадной схемы. Длительность последовательной части выполняемых операций (минимально возможное время параллельного исполнения) в этом случае составляет

$$T_p=(n/p)+log_2 p,$$

что приводит к оценке доли последовательных расчетов как величины

$$f=(1/p)+log_2 p/n.$$

Как следует из полученного выражения, доля последовательных расчетов f убывает с ростом n и в предельном случае мы получаемом идеальную оценку максимально возможного ускорения $S^*=p$.

2. Закон Густавсона – Барсиса. Оценим максимально достижимое ускорение исходя из имеющейся доли последовательных расчетов в выполняемых параллельных вычислениях:

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

где $\tau(n)$ и $\pi(n)$ есть времена последовательной и параллельной частей выполняемых вычислений соответственно, т.е.

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

С учетом введенной величины g можно получить

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

что позволяет построить оценку для ускорения

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

которая после упрощения приводится к виду *закона Густавсона – Барсиса* (*Gustafson – Barsis's law*) (см. [63]):

$$S_p = g + (1-g)p = p + (1-p)g.$$

Применительно к учебному примеру суммирования значений при использовании p процессоров время параллельного выполнения, как уже отмечалось выше, составляет

$$T_p = (n/p) + \log_2 p,$$

что соответствует последовательной доле

$$g = \frac{\log_2 p}{(n/p)+\log_2 p}.$$

За счет увеличения числа суммируемых значений величина g может быть пренебрежимо малой, обеспечивая получение идеального возможного ускорения $S_p=p$.

При рассмотрении закона Густавсона – Барсиса следует учитывать еще один важный момент. С увеличением числа используемых процессоров темп уменьшения времени параллельного решения задач может падать (после превышения определенного порога). Однако за счет уменьшения времени вычислений сложность решаемых задач может быть увеличена (так, например, для учебной задачи суммирования может быть увеличен размер складываемого набора значений). Оценку получаемого при этом ускорения можно определить при помощи сформулированных закономерностей. Такая аналитическая оценка тем более полезна, поскольку решение

таких более сложных вариантов задач на одном процессоре может оказаться достаточно трудоемким и даже невозможным, например, в силу нехватки оперативной памяти. С учетом указанных обстоятельств оценку ускорения, получаемую в соответствии с законом Густавсона – Барсиса, еще называют *ускорением масштабирования* (*scaled speedup*), поскольку данная характеристика может показать, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач.

2.7. Анализ масштабируемости параллельных вычислений

Целью применения параллельных вычислений во многих случаях является не только уменьшение времени выполнения расчетов, но и обеспечение возможности решения более сложных вариантов задач (таких постановок, решение которых не представляется возможным при использовании однопроцессорных вычислительных систем). Способность параллельного алгоритма эффективно использовать процессоры при повышении сложности вычислений является важной характеристикой выполняемых расчетов. Поэтому параллельный алгоритм называют *масштабируемым* (*scalable*), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении постоянного уровня эффективности использования процессоров. Возможный способ характеристики свойств масштабируемости состоит в следующем.

Оценим *накладные расходы* (*total overhead*), которые имеют место при выполнении параллельного алгоритма

$$T_0 = pT_p - T_1.$$

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

$$T_p = \frac{T_1 + T_0}{p}, \quad S_p = \frac{T_1}{T_p} = \frac{pT_1}{T_1 + T_0}.$$

Применяя полученные соотношения, эффективность использования процессоров можно выразить как

$$E_p = \frac{S_p}{p} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0 / T_1}.$$

Последнее выражение показывает, что если сложность решаемой задачи является фиксированной (T_1=*const*), то при росте числа процессоров эффективность, как правило, будет убывать за счет роста накладных расходов T_0. При фиксации числа процессоров эффективность их использования можно улучшить путем повышения сложности решаемой задачи T_1 (предполагается, что при росте параметра сложности n накладные расходы T_0 увеличиваются медленнее, чем объем вычислений T_1). Как результат, при увеличении числа процессоров в большинстве случаев можно обеспечить определенный уровень эффективности при помощи соответствующего повышения сложности решаемых задач. Поэтому важной характеристикой параллельных вычислений становится соотношение необходимых темпов роста сложности расчетов и числа используемых процессоров.

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

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

Порождаемую последним соотношением зависимость $n=F(p)$ между сложностью решаемой задачи и числом процессоров обычно называют *функцией изоэффективности* (*isoefficiency function*) (см. [51]).

Покажем в качестве иллюстрации вывод функции изоэффективности для учебного примера суммирования числовых значений. В этом случае

$$T_0 = pT_p - T_1 = p((n/p) + \log_2 p) - n = p\log_2 p$$

и функция изоэффективности принимает вид

$$n = Kp\log_2 p\ .$$

Как результат, например, при числе процессоров p=16 для обеспечения уровня эффективности E=0,5 (т.е. K=1) количество суммируемых значений должно быть не менее n=64. Или же, при увеличении числа процессоров с p до q ($q>p$) для обеспечения пропорционального роста ускорения $(S_q/S_p)=(q/p)$ необходимо увеличить число суммируемых значений n в $(q\log_2 q)/(p\log_2 p)$ раз.

2.8. Краткий обзор лекции

В лекции рассматривается модель вычислений в виде графа «операции – операнды», которая может использоваться для описания существующих информационных зависимостей в выбираемых алгоритмах решения задач.

В основу данной модели положен ациклический ориентированный граф, в котором вершины представляют операции, а дуги соответствуют зависимостям операций по данным. При наличии такого графа для определения параллельного алгоритма достаточно задать расписание, в соответствии с которым фиксируется распределение выполняемых операций по процессорам.

Представление вычислений при помощи моделей подобного вида позволяет получить аналитически ряд характеристик разрабатываемых параллельных алгоритмов, среди которых время выполнения, схема оптимального расписания, оценки максимально возможного быстродействия методов решения поставленных задач. Для более простого построения теоретических оценок в лекции рассматривается понятие *паракомпьютера* как параллельной системы с неограниченным количеством процессоров.

Для оценки оптимальности разрабатываемых методов параллельных вычислений в лекции приводятся широко используемые в теории и практике параллельного программирования основные показатели качества — *ускорение* (*speedup*), показывающее, во сколько раз быстрее осуществляется решение задач при использовании нескольких процессоров, и *эффективность* (*efficiency*), которая характеризует долю времени реального использования процессоров вычислительной системы. Важной характеристикой разрабатываемых алгоритмов является *стоимость* (*cost*) вычислений, определяемая как произведение времени параллельного решения задачи и числа используемых процессоров.

Для демонстрации применимости рассмотренных моделей и методов анализа параллельных алгоритмов в лекции рассматривается задача нахождения частных сумм последовательности числовых значений. На данном примере отмечается проблема сложности распараллеливания последовательных алгоритмов, которые изначально не были ориентированы на возможность организации параллельных вычислений. Для выделения «скрытого» параллелизма показывается возможность преобразования исходной последовательной схемы вычислений и приводится получаемая в результате таких преобразований каскадная схема. На примере этой же задачи отмечается возможность введения избыточных вычислений для достижения большего параллелизма выполняемых расчетов.

В завершение лекции рассматривается вопрос построения оценок максимально достижимых значений показателей эффективности. Для получения таких оценок может быть использован *закон Амдаля* (*Amdahl*), позволяющий учесть существование последовательных (нераспараллеливаемых) вычислений в методах решения задач. *Закон Густавсона — Барсиса* (*Gustafson — Barsis's law*) обеспечивает построение оценок *ускорения масштабирования* (*scaled speedup*), применяемое для характеристики того, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач. Для определения зависимости между сложностью решаемой задачи и числом процессоров, при соблюдении которой обеспечивается необходимый уровень эффективности параллельных вычислений, вводится понятие *функции изоэффективности* (*isoefficiency function*).

2.9. Обзор литературы

Дополнительная информация по моделированию и анализу параллельных вычислений может быть получена, например, в [2, 22]), полезная информация содержится также в [51,63].

Рассмотрение учебной задачи суммирования последовательности числовых значений было выполнено в [22].

Впервые закон Амдаля был изложен в работе [18]. Закон Густавсона – Барсиса был опубликован в работе [43]. Понятие функции изоэффективности было предложено в работе [39].

Систематическое изложение (на момент издания работы) вопросов моделирования и анализа параллельных вычислений приводится в [77].

2.10. Контрольные вопросы

1. Как определяется модель «операции — операнды»?
2. Как определяется расписание для распределения вычислений между процессорами?
3. Как определяется время выполнения параллельного алгоритма?
4. Какое расписание является оптимальным?
5. Как определить минимально возможное время решения задачи?
6. Что понимается под паракомпьютером и для чего может оказаться полезным данное понятие?
7. Какие оценки следует использовать в качестве характеристики времени последовательного решения задачи?
8. Как определить минимально возможное время параллельного решения задачи по графу «операнды – операции»?
9. Какие зависимости могут быть получены для времени параллельного решения задачи при увеличении или уменьшении числа используемых процессоров?
10. При каком числе процессоров могут быть получены времена выполнения параллельного алгоритма, сопоставимые по порядку с оценками минимально возможного времени решения задачи?
11. Как определяются понятия ускорения и эффективности?
12. Возможно ли достижение сверхлинейного ускорения?
13. В чем состоит противоречивость показателей ускорения и эффективности?
14. Как определяется понятие стоимости вычислений?
15. В чем состоит понятие стоимостно-оптимального алгоритма?
16. В чем заключается проблема распараллеливания последовательного алгоритма суммирования числовых значений?
17. В чем состоит каскадная схема суммирования? С какой целью рассматривается модифицированный вариант данной схемы?

18. В чем состоит различие показателей ускорения и эффективности для рассматриваемых вариантов каскадной схемы суммирования?
19. В чем состоит параллельный алгоритм вычисления всех частных сумм последовательности числовых значений?
20. Как формулируется закон Амдаля? Какой аспект параллельных вычислений позволяет учесть данный закон?
21. Какие предположения используются для обоснования закона Густавсона – Барсиса?
22. Как определяется функция изоэффективности?
23. Какой алгоритм является масштабируемым? Приведите примеры методов с разным уровнем масштабируемости.

2.11. Задачи и упражнения

1. Разработайте модель и выполните оценку показателей ускорения и эффективности параллельных вычислений:

• для задачи скалярного произведения двух векторов

$$y = \sum_{i=1}^{N} a_i b_i \ ;$$

• для задачи поиска максимального и минимального значений для заданного набора числовых данных

$$y_{\min} = \min_{1 \le i \le N} a_i, \ y_{\max} = \max_{1 \le i \le N} a_i \ ;$$

• для задачи нахождения среднего значения для заданного набора числовых данных

$$y = \frac{1}{N} \sum_{i=1}^{N} a_i \ .$$

2. Выполните в соответствии с законом Амдаля оценку максимально достижимого ускорения для задач п. 1.
3. Выполните оценку ускорения масштабирования для задач п.1.
4. Выполните построение функций изоэффективности для задач п. 1.
5. Разработайте модель и выполните полный анализ эффективности параллельных вычислений (ускорение, эффективность, максимально достижимое ускорение, ускорение масштабирования, функция изоэффективности) для задачи умножения матрицы на вектор.

Лекция 3. Оценка коммуникационной трудоемкости параллельных алгоритмов

Лекция посвящена вопросам анализа информационных потоков, возникающих при выполнении параллельных алгоритмов. Дается общая характеристика механизмов передачи данных, проводится анализ трудоемкости основных операций обмена информацией, рассматриваются методы логического представления структуры многопроцессорных вычислительных систем.

Ключевые слова: алгоритмы маршрутизации, методы передачи данных, передача данных от одного процессора всем остальным процессорам сети, передача данных от всех процессоров всем процессорам сети, прием сообщений на каждом процессоре от всех процессоров сети, задача редукции, операция циклического сдвига, топология сети передачи даных, оценка трудоемкости коммуникационных операций.

3.1. Общая характеристика механизмов передачи данных

3.1.1. Алгоритмы маршрутизации

Алгоритмы маршрутизации определяют путь передачи данных от процессора – источника сообщения до процессора, к которому сообщение должно быть доставлено. Среди возможных способов решения данной задачи различают:

- *оптимальные*, определяющие всегда наикратчайшие пути передачи данных, и *неоптимальные* алгоритмы маршрутизации;
- *детерминированные* и *адаптивные* методы выбора маршрутов (адаптивные алгоритмы определяют пути передачи данных в зависимости от существующей загрузки коммуникационных каналов).

К числу наиболее распространенных оптимальных алгоритмов относится класс *методов покоординатной маршрутизации* (*dimension-ordered routing*), в которых поиск путей передачи данных осуществляется поочередно для каждой размерности топологии сети коммуникации. Так, для двумерной решетки такой подход приводит к маршрутизации, при которой передача данных сначала выполняется по одному направлению (например, по горизонтали до достижения вертикали, на которой располагается процессор назначения), а затем данные передаются вдоль другого направления (данная схема известна под названием *алгоритма XY-маршрутизации*).

Для гиперкуба покоординатная схема маршрутизации может состоять, например, в циклической передаче данных процессору, определяемому первой различающейся битовой позицией в номерах процессоров — того, на котором сообщение располагается в данный момент времени, и того, на который оно должно быть передано.

3.1.2. Методы передачи данных

Время передачи данных между процессорами определяет *коммуникационную составляющую* (*communication latency*) длительности выполнения параллельного алгоритма в многопроцессорной вычислительной системе. Основной набор параметров, описывающих время передачи данных, состоит из следующего ряда величин:

- *время начальной подготовки* ($t_н$) характеризует длительность подготовки сообщения для передачи, поиска маршрута в сети и т. п.;
- *время передачи служебных данных* ($t_с$) между двумя соседними процессорами (т. е. для процессоров, между которыми имеется физический канал передачи данных). К служебным данным может относиться заголовок сообщения, блок данных для обнаружения ошибок передачи и т. п.;
- *время передачи одного слова данных* по одному каналу передачи данных ($t_к$). Длительность подобной передачи определяется полосой пропускания коммуникационных каналов в сети.

К числу наиболее распространенных методов передачи данных относятся два основных способа коммуникации (см., например, [51]). Первый из них ориентирован на *передачу сообщений* (*метод передачи сообщений* или *МПС*) как неделимых (атомарных) блоков информации (*store-and-forward routing* или *SFR*). При таком подходе процессор, содержащий сообщение для передачи, готовит весь объем данных для передачи, определяет процессор, которому следует направить данные, и запускает операцию пересылки данных. Процессор, которому направлено сообщение, в первую очередь осуществляет прием полностью всех пересылаемых данных и только затем приступает к пересылке принятого сообщения далее по маршруту. Время пересылки данных $t_{пд}$ для метода передачи сообщения размером m байт по маршруту длиной l определяется выражением:

$$t_{пд} = t_н + (mt_к + t_с)l \ . \qquad (3.1)$$

При достаточно длинных сообщениях временем передачи служебных данных можно пренебречь и выражение для времени передачи данных может быть записано в более простом виде:

$$t_{пд} = t_н + mt_к l \ . \qquad (3.2)$$

Второй способ коммуникации основывается на представлении пересылаемых сообщений в виде блоков информации меньшего размера – пакетов, в результате чего передача данных может быть сведена к *передаче пакетов* (*метод передачи пакетов* или *МПП*). При таком методе коммуникации (*cut-through routing* или *CTR*) принимающий процессор может осуществлять пересылку данных по дальнейшему маршруту непосредственно сразу после приема очередного пакета, не дожидаясь завершения приема данных всего сообщения. Время пересылки данных при использовании метода передачи пакетов определяется выражением:

$$t_{пд} = t_н + mt_к + t_c l \ . \qquad (3.3)$$

Сравнивая полученные выражения, можно заметить, что в большинстве случаев метод передачи пакетов приводит к более быстрой пересылке данных; кроме того, данный подход снижает потребность в памяти для хранения пересылаемых данных при организации приема-передачи сообщений, а для передачи пакетов могут использоваться одновременно разные коммуникационные каналы. С другой стороны, реализация пакетного метода требует разработки более сложного аппаратного и программного обеспечения сети, может увеличить накладные расходы (время подготовки и время передачи служебных данных). Кроме того, при передаче пакетов возможно возникновение конфликтных ситуаций (дедлоков).

3.2. Анализ трудоемкости основных операций передачи данных

При всем разнообразии выполняемых операций передачи данных при параллельных способах решения сложных научно-технических задач, определенные процедуры взаимодействия процессоров сети могут быть отнесены к числу основных коммуникационных действий, либо наиболее широко распространенных в практике параллельных вычислений, либо тех, к которым могут быть сведены многие другие процессы приема-передачи сообщений. Важно отметить также, что в рамках подобного базового набора для большинства операций коммуникации существуют процедуры, обратные по действию исходным операциям (так, например, операции передачи данных от одного процессора всем имеющимся процессорам сети соответствует операция приема в одном процессоре сообщений от всех остальных процессоров). Как результат, рассмотрение коммуникационных процедур целесообразно выполнять попарно, поскольку во многих случаях алгоритмы выполнения прямой и обратной операций могут быть получены исходя из одинаковых предпосылок.

Рассмотрение основных операций передачи данных в этой лекции будет осуществляться на примере таких топологий сети, как кольцо, двумерная решетка и гиперкуб. Для двумерной решетки будет предполагаться также, что между граничными процессорами в строках и столбцах решетки имеются каналы передачи данных (т. е. топология сети представляет собой тор). Как и ранее, величина m будет означать размер сообщения в словах, значение p определяет количество процессоров в сети, а переменная N задает размерность топологии гиперкуба.

3.2.1. Передача данных между двумя процессорами сети

Трудоемкость данной коммуникационной операции может быть получена путем подстановки длины максимального пути (диаметра сети) в выражения для времени передачи данных при разных методах коммуникации (см. п. 3.1.2) – см. табл. 3.1.

Таблица 3.1. Время передачи данных между двумя процессорами

Топология	Передача сообщений	Передача пакетов
Кольцо	$t_н + mt_к \lfloor p/2 \rfloor$	$t_н + mt_к + t_c \lfloor p/2 \rfloor$
Решетка-тор	$t_н + 2mt_к \left\lfloor \sqrt{p}/2 \right\rfloor$	$t_н + mt_к + 2t_c \left\lfloor \sqrt{p}/2 \right\rfloor$
Гиперкуб	$t_н + mt_к \log_2 p$	$t_н + mt_к + t_c \log_2 p$

3.2.2. Передача данных от одного процессора всем остальным процессорам сети

Операция передачи данных (одного и того же сообщения) от одного процессора всем остальным процессорам сети (*one-to-all broadcast* или *single-node broadcast*) является одним из наиболее часто выполняемых коммуникационных действий. Двойственная ей операция – прием на одном процессоре сообщений от всех остальных процессоров сети (*single-node accumulation*). Подобные операции используются, в частности, при реализации матрично-векторного умножения, решении систем линейных уравнений методом Гаусса, решении задачи поиска кратчайших путей и др.

Простейший способ реализации операции рассылки состоит в ее выполнении как последовательности попарных взаимодействий процессоров сети. Однако при таком подходе большая часть пересылок является избыточной и возможно применение более эффективных алгоритмов

коммуникации. Изложение материала будет проводиться сначала для метода передачи сообщений, затем – для пакетного способа передачи данных (см. п. 3.1.2).

Передача сообщений. Для **кольцевой топологии** процессор – источник рассылки может инициировать передачу данных сразу двум своим соседям, которые, в свою очередь, приняв сообщение, организуют пересылку далее по кольцу. Трудоемкость выполнения операции рассылки в этом случае будет определяться соотношением:

$$t_{пд} = (t_н + mt_к)\lceil p/2 \rceil. \qquad (3.4)$$

Для топологии типа **решетка-тор** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети. Так, рассылка может быть выполнена в виде двухэтапной процедуры. На первом этапе организуется передача сообщения всем процессорам сети, располагающимся на той же горизонтали решетки, что и процессор – инициатор передачи. На втором этапе процессоры, получившие копию данных на первом этапе, рассылают сообщения по своим соответствующим вертикалям. Оценка длительности операции рассылки в соответствии с описанным алгоритмом определяется соотношением:

$$t_{пд} = 2(t_н + mt_к)\lceil \sqrt{p}/2 \rceil. \qquad (3.5)$$

Для гиперкуба рассылка может быть выполнена в ходе N-этапной процедуры передачи данных. На первом этапе процессор-источник сообщения передает данные одному из своих соседей (например, по первой размерности) – в результате после первого этапа есть два процессора, имеющих копию пересылаемых данных (данный результат можно интерпретировать также как разбиение исходного гиперкуба на два таких одинаковых по размеру гиперкуба размерности N-1, что каждый из них имеет копию исходного сообщения). На втором этапе два процессора, задействованные на первом этапе, пересылают сообщение своим соседям по второй размерности и т. д. В результате такой рассылки время операции оценивается при помощи выражения:

$$t_{пд} = (t_н + mt_к)\log_2 p. \qquad (3.6)$$

Сравнивая полученные выражения для длительности выполнения операции рассылки, можно отметить, что наилучшие показатели имеет топология типа гиперкуб; более того, можно показать, что данный результат является наилучшим для выбранного способа коммуникации с помощью передачи сообщений.

Передача пакетов. Для топологии типа **кольцо** алгоритм рассылки может быть получен путем логического представления кольцевой структуры сети в виде гиперкуба. В результате на этапе рассылки процессор – источник сообщения передает данные процессору, находящемуся на расстоянии $p/2$ от исходного процессора. Далее, на втором этапе оба процессора, уже имеющие рассылаемые данные после первого этапа, передают сообщения процессорам, находящимся на расстоянии $p/4$, и т. д. Трудоемкость выполнения операции рассылки при таком методе передачи данных определяется соотношением:

$$t_{пд} = \sum_{i=1}^{\log_2 p} (t_н + mt_к + t_c p/2^i) = (t_н + mt_к)\log_2 p + t_c(p-1) \qquad (3.7)$$

(как и ранее, при достаточно больших сообщениях временем передачи служебных данных можно пренебречь).

Для топологии типа **решетка-тор** алгоритм рассылки может быть получен из способа передачи данных, примененного для кольцевой структуры сети, в соответствии с тем же способом обобщения, что и в случае использования метода передачи сообщений. Получаемый в результате такого обобщения алгоритм рассылки характеризуется следующим соотношением для оценки времени выполнения:

$$t_{пд} = (t_н + mt_к)\log_2 p + 2t_c(\sqrt{p} - 1). \qquad (3.8)$$

Для **гиперкуба** алгоритм рассылки (и, соответственно, временные оценки длительности выполнения) при передаче пакетов не отличается от варианта для метода передачи сообщений.

3.2.3. Передача данных от всех процессоров всем процессорам сети

Операция передачи данных от всех процессоров всем процессорам сети (*all-to-all broadcast* или *multinode broadcast*) является естественным обобщением одиночной операции рассылки, двойственная ей операция – прием сообщений на каждом процессоре от всех процессоров сети (*multinode accumulation*). Подобные операции широко используются, например, при реализации матричных вычислений.

Возможный способ реализации операции множественной рассылки состоит в выполнении соответствующего набора операций одиночной рассылки. Однако такой подход не является оптимальным для многих топологий сети, поскольку часть необходимых операций одиночной рассылки потенциально может быть выполнена параллельно. Как и ранее, материал будет рассматриваться раздельно для разных методов передачи данных (см. п. 3.1.2).

Передача сообщений. Для **кольцевой топологии** каждый процессор может инициировать рассылку своего сообщения одновременно (в какомлибо выбранном направлении по кольцу). В любой момент каждый процессор выполняет прием и передачу данных, завершение операции множественной рассылки произойдет через p-1 цикл передачи данных. Длительность выполнения операции рассылки оценивается соотношением:

$$t_{nд} = (t_н + mt_к)(p-1). \qquad (3.9)$$

Для топологии типа **решетка-тор** множественная рассылка сообщений может быть выполнена при помощи алгоритма, получаемого обобщением способа передачи данных для кольцевой структуры сети. Схема обобщения состоит в следующем. На первом этапе организуется передача сообщений раздельно по всем процессорам сети, располагающимся на одних и тех же горизонталях решетки (в результате на каждом процессоре одной и той же горизонтали формируются укрупненные сообщения размера $m\sqrt{p}$, объединяющие все сообщения горизонтали). Время выполнения этапа:

$$t'_{nд} = (t_н + mt_к)(\sqrt{p}-1).$$

На втором этапе рассылка данных выполняется по процессорам сети, образующим вертикали решетки. Длительность этого этапа:

$$t''_{nд} = (t_н + m\sqrt{p}t_к)(\sqrt{p}-1).$$

Общая длительность операции рассылки определяется соотношением:

$$t_{nд} = 2t_н(\sqrt{p}-1) + mt_к(p-1). \qquad (3.10)$$

Для **гиперкуба** алгоритм множественной рассылки сообщений может быть получен путем обобщения ранее описанного способа передачи данных для топологии типа решетки на размерность гиперкуба *N*. В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе i, $1 \le i \le N$, выполнения алгоритма функционируют все процессоры сети, которые обмениваются своими данными со своими соседями по i-ой размерности и формируют объединенные сообщения. Время операции рассылки может быть получено при помощи выражения:

$$t_{nд} = \sum_{i=1}^{\log_2 p} (t_н + 2^{i-1} mt_к) = t_н \log_2 p + mt_к(p-1). \qquad (3.11)$$

Передача пакетов. Применение более эффективного для **кольцевой структуры** и топологии типа **решетка-тор** метода передачи данных не приводит к какому-либо улучшению времени выполнения операции множественной рассылки, поскольку обобщение алгоритмов выполнения операции одиночной рассылки на случай множественной рассылки приводит к перегрузке каналов передачи данных (т. е. к существованию ситуаций, когда в один и тот же момент для передачи по одной и той же линии имеется несколько ожидающих пересылки пакетов данных). Перегрузка каналов приводит к задержкам при пересылках данных, что и не позволяет проявиться всем преимуществам метода передачи пакетов.

Широко распространенным примером операции множественной рассылки является *задача редукции* (*reduction*), которая определяется в общем виде как процедура выполнения той или иной обработки данных, получаемых на каждом процессоре в ходе множественной рассылки (в качестве примера такой задачи может быть рассмотрена проблема вычисления суммы значений, находящихся на разных процессорах, и рассылки полученной суммы по всем процессорам сети). Способы решения задачи редукции могут состоять в следующем:

- непосредственный подход заключается в выполнении операции множественной рассылки и последующей затем обработке данных на каждом процессоре в отдельности;
- более эффективный алгоритм может быть получен в результате применения операции одиночного приема данных на отдельном процессоре, выполнения на этом процессоре действий по обработке данных и рассылки полученного результата обработки всем процессорам сети;
- наилучший же способ решения задачи редукции состоит в совмещении процедуры множественной рассылки и действий по обработке данных, когда каждый процессор сразу же после приема очередного сообщения реализует требуемую обработку полученных данных (например, выполняет сложение полученного значения с имеющейся на процессоре частичной суммой). Время решения задачи редукции при таком алгоритме реализации в случае, например, когда размер пересылаемых данных имеет единичную длину (m=1) и топология сети имеет структуру гиперкуба, определяется выражением:

$$t_{пд} = (t_н + t_к)\log_2 p. \qquad (3.12)$$

Другим типовым примером использования операции множественной рассылки является задача нахождения частных сумм последовательности значений S_i (в англоязычной литературе эта задача известна под названием *prefix sum problem*)

$$S_k = \sum_{i=1}^{k} x_i, \quad 1 \leqslant k \leqslant p \tag{3.13}$$

(будем предполагать, что количество значений совпадает с количеством процессоров, значение x_i располагается на i-м процессоре и результат S_k должен получаться на процессоре с номером k).

Алгоритм решения данной задачи также может быть получен при помощи конкретизации общего способа выполнения множественной операции рассылки, когда процессор выполняет суммирование полученного значения (но только в том случае, если процессор – отправитель значения имеет меньший номер, чем процессор-получатель).

3.2.4. Обобщенная передача данных от одного процессора всем остальным процессорам сети

Общий случай передачи данных от одного процессора всем остальным процессорам сети состоит в том, что все рассылаемые сообщения являются различными (*one-to-all personalized communication* или *single-node scatter*). Двойственная операция передачи для данного типа взаимодействия процессоров – обобщенный прием сообщений (*single-node gather*) на одном процессоре от всех остальных процессоров сети (отличие данной операции от ранее рассмотренной процедуры сборки данных на одном процессоре состоит в том, что обобщенная операция сборки не предполагает какого-либо взаимодействия сообщений (например, редукции) в процессе передачи данных).

Трудоемкость операции обобщенной рассылки сопоставима со сложностью выполнения процедуры множественной передачи данных. Процессор – инициатор рассылки посылает каждому процессору сети сообщение размера m, и, тем самым, нижняя оценка длительности выполнения операции характеризуется величиной $mt_k(p-1)$.

Проведем более подробный анализ трудоемкости обобщенной рассылки для случая топологии типа **гиперкуб**. Возможный способ выполнения операции состоит в следующем. Процессор – инициатор рассылки передает половину своих сообщений одному из своих соседей (например, по первой размерности) – в результате исходный гиперкуб становится разделенным на два гиперкуба половинного размера, в каждом из которых содержится ровно половина исходных данных. Далее, действия по рассылке сообщений могут быть повторены, и общее количество повторений определяется исходной размерностью гиперкуба. Длительность операции обобщенной рассылки может быть охарактеризована соотношением:

$$t_{пд} = t_н \log_2 p + mt_к (p-1) \tag{3.14}$$

(как и отмечалась выше, трудоемкость операции совпадает с длительностью выполнения процедуры множественной рассылки).

3.2.5. Обобщенная передача данных от всех процессоров всем процессорам сети

Обобщенная передача данных от всех процессоров всем процессорам сети (*total exchange*) представляет собой наиболее общий случай коммуникационных действий. Необходимость выполнения подобных операций возникает в параллельных алгоритмах быстрого преобразования Фурье, транспонирования матриц и др.

Выполним краткую характеристику возможных способов выполнения обобщенной множественной рассылки для разных методов передачи данных (см. п. 3.1.2).

Передача сообщений. Общая схема алгоритма для **кольцевой топологии** состоит в следующем. Каждый процессор производит передачу всех своих исходных сообщений своему соседу (в каком-либо выбранном направлении по кольцу). Далее процессоры осуществляют прием направленных к ним данных, затем среди принятой информации выбирают свои сообщения, после чего выполняют дальнейшую рассылку оставшейся части данных. Длительность выполнения подобного набора передач данных оценивается при помощи выражения:

$$t_{n\partial} = (t_н + \frac{1}{2} mpt_к)(p-1). \qquad (3.15)$$

Способ получения алгоритма рассылки данных для топологии типа **решетка-тор** является тем же самым, что и в случае рассмотрения других коммуникационных операций. На первом этапе организуется передача сообщений раздельно по всем процессорам сети, располагающимся на одних и тех же горизонталях решетки (каждому процессору по горизонтали передаются только те исходные сообщения, что должны быть направлены процессорам соответствующей вертикали решетки). После завершения этапа на каждом процессоре собираются p сообщений, предназначенных для рассылки по одной из вертикалей решетки. На втором этапе рассылка данных выполняется по процессорам сети, образующим вертикали решетки. Общая длительность всех операций рассылок определяется соотношением:

$$t_{n\partial} = (2t_н + mpt_к)(\sqrt{p}-1). \qquad (3.16)$$

Для **гиперкуба** алгоритм обобщенной множественной рассылки сообщений может быть получен путем обобщения способа выполнения

операции для топологии типа решетка на размерность гиперкуба *N*. В результате такого обобщения схема коммуникации состоит в следующем. На каждом этапе *i*, $1 \le i \le N$, выполнения алгоритма функционируют все процессоры сети, которые обмениваются своими данными со своими соседями по *i*-й размерности и формируют объединенные сообщения. При организации взаимодействия двух соседей канал связи между ними рассматривается как связующий элемент двух равных по размеру подгиперкубов исходного гиперкуба, и каждый процессор пары посылает другому процессору только те сообщения, что предназначены для процессоров соседнего подгиперкуба. Время операции рассылки может быть получено при помощи выражения:

$$t_{пд} = (t_н + \frac{1}{2} m p t_к) \log_2 p \qquad (3.17)$$

(кроме затрат на пересылку, каждый процессор выполняет $mp\log_2 p$ операций по сортировке своих сообщений перед обменом информацией со своими соседями).

Передача пакетов. Как и в случае множественной рассылки, применение метода передачи пакетов не приводит к улучшению временных характеристик для операции обобщенной множественной рассылки. Рассмотрим как пример более подробно выполнение данной коммуникационной операции для сети с топологией типа **гиперкуб**. В этом случае рассылка может быть выполнена за p-1 итерацию. На каждой итерации все процессоры разбиваются на взаимодействующие пары процессоров, причем это разбиение на пары может быть выполнено таким образом, чтобы передаваемые между разными парами сообщения не использовали одни и те же пути передачи данных. Как результат, общая длительность операции обобщенной рассылки может быть определена в соответствии с выражением:

$$t_{пд} = (t_н + m t_к)(p-1) + \frac{1}{2} t_c p \log_2 p \,. \qquad (3.18)$$

3.2.6. Циклический сдвиг

Частный случай обобщенной множественной рассылки есть *процедура перестановки* (*permutation*), представляющая собой операцию перераспределения информации между процессорами сети, в которой каждый процессор передает сообщение определенному неким способом другому процессору сети. Конкретный вариант перестановки – *циклический q-сдвиг* (*cirlular q-shift*), при котором каждый процессор *i*, $1 \le i \le N$, передает данные процессору с номером $(i+q) \bmod p$. Подобная операция сдвига используется, например, при организации матричных вычислений.

Поскольку выполнение циклического сдвига для кольцевой топологии может быть обеспечено при помощи простых алгоритмов передачи данных, рассмотрим возможные способы выполнения данной коммуникационной операции только для топологий решетка-тор и гиперкуб при разных методах передачи данных (см. п. 3.1.2).

Передача сообщений. Общая схема алгоритма циклического сдвига для топологии типа **решетка-тор** состоит в следующем. Пусть процессоры перенумерованы по строкам решетки от 0 до p-1. На первом этапе организуется циклический сдвиг с шагом $q \bmod \sqrt{p}$ по каждой строке в отдельности (если при реализации такого сдвига сообщения передаются через правые границы строк, то после выполнения каждой такой передачи необходимо осуществить компенсационный сдвиг вверх на 1 для процессоров первого столбца решетки). На втором этапе реализуется циклический сдвиг вверх с шагом $\lfloor q/\sqrt{p} \rfloor$ для каждого столбца решетки. Общая длительность всех операций рассылок определяется соотношением:

$$t_{n\partial} = (t_н + mt_к)(2\lfloor \sqrt{p}/2 \rfloor + 1). \qquad (3.19)$$

Для гиперкуба алгоритм циклического сдвига может быть получен путем логического представления топологии гиперкуба в виде кольцевой структуры. Для получения такого представления установим взаимно-однозначное соответствие между вершинами кольца и гиперкуба. Необходимое соответствие может быть получено, например, при помощи известного кода Грея. Более подробное изложение механизма установки такого соответствия осуществляется в подразделе 3.3; для наглядности на рис. 3.1 приводится вид гиперкуба для размерности N=3 с указанием для каждого процессора гиперкуба соответствующей вершины кольца. Положительным свойством выбора такого соответствия является тот факт, что для любых двух вершин в кольце, расстояние между которыми равно l=2^i для некоторого значения i, путь между соответствующими вершинами в гиперкубе содержит только две линии связи (за исключением случая i=0, когда путь в гиперкубе имеет единичную длину).

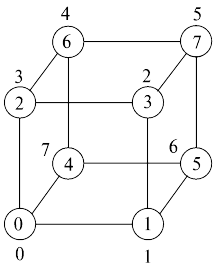


Рис. 3.1. Схема отображения гиперкуба на кольцо (в кружках приведены номера процессоров гиперкуба)

Представим величину сдвига q в виде двоичного кода. Количество ненулевых позиций кода определяет количество этапов в схеме реализации операции циклического сдвига. На каждом этапе выполняется операция сдвига с величиной шага, задаваемой наиболее старшей ненулевой позицией значения q (например, при исходной величине сдвига $q=5=101_2$ на первом этапе выполняется сдвиг с шагом 4, на втором этапе шаг сдвига равен 1). Выполнение каждого этапа (кроме сдвига с шагом 1) состоит в передаче данных по пути, включающему две линии связи. Как результат, верхняя оценка для длительности выполнения операции циклического сдвига определяется соотношением:

$$t_{пд} = (t_н + mt_к)(2\log_2 p - 1). \qquad (3.20)$$

Передача пакетов. Использование пересылки пакетов может повысить эффективность выполнения операции циклического сдвига для топологии **гиперкуб**. Реализация всех необходимых коммуникационных действий в этом случае может быть обеспечена путем отправления каждым процессором всех пересылаемых данных непосредственно процессорам назначения. Применение метода покоординатной маршрутизации (см. п. 3.1.1) позволит избежать коллизий при использовании линий передачи данных (в каждый момент времени для каждого канала будет существовать не более одного готового для отправки сообщения). Длина наибольшего пути при такой рассылке данных определяется как $\log_2 p - \gamma(q)$, где $\gamma(q)$ есть наибольшее целое значение j такое, что 2^j есть делитель величины сдвига q. Тогда длительность операции циклического сдвига может быть охарактеризована при помощи выражения

$$t_{пд} = t_н + mt_к + t_c(\log_2 p - \gamma(q)) \qquad (3.21)$$

(при достаточно больших размерах сообщений временем передачи служебных данных можно пренебречь и время выполнения операции может быть определено как $t_{пд}=t_н+mt_к$).

3.3. Методы логического представления топологии коммуникационной среды

Как показало рассмотрение основных коммуникационных операций в подразделе 3.1, ряд алгоритмов передачи данных допускает более простое изложение при использовании вполне определенных топологий сети межпроцессорных соединений. Кроме того, многие методы коммуникации могут быть получены при помощи того или иного логического представления исследуемой топологии. Как результат, важным моментом при организации параллельных вычислений является возможность *логи-*

ческого представления разнообразных топологий на основе конкретных (физических) межпроцессорных структур.

Способы логического представления (отображения) топологий характеризуются следующими тремя основными характеристиками:

- *уплотнение дуг* (*congestion*), выражаемое как максимальное количество дуг логической топологии, которые отображаются в одну линию передачи физической топологии;
- *удлинение дуг* (*dilation*), определяемое как путь максимальной длины физической топологии, на который отображается дуга логической топологии;
- *увеличение вершин* (*expansion*), вычисляемое как отношение количества вершин в логической и физической топологиях.

Для рассматриваемых в рамках пособия топологий ограничимся изложением вопросов отображения топологий кольца и решетки на гиперкуб. Предлагаемые ниже подходы для логического представления топологий характеризуются единичными показателями уплотнения и удлинения дуг.

3.3.1. Представление кольцевой топологии в виде гиперкуба

Установление соответствия между кольцевой топологией и гиперкубом может быть выполнено при помощи *двоичного рефлексивного кода Грея G*(*i, N*) (*binary reflected Gray code*), определяемого в соответствии с выражениями:

$$G(0,1)=0, \quad G(1,1)=1,$$

$$G(i,s+1)=\begin{cases} G(i,s), & i<2^s, \\ 2^s+G(2^{s+1}-1-i,s), & i\geq 2^s, \end{cases} \qquad (3.22)$$

где i задает номер значения в коде Грея, а N есть длина этого кода. Для иллюстрации подхода на рис. 3.2 показывается отображение кольцевой топологии на гиперкуб для сети из p=8 процессоров.

Важное свойство кода Грея: соседние значения $G(i,N)$ и $G(i+1,N)$ имеют только одну различающуюся битовую позицию. Как результат, соседние вершины в кольцевой топологии отображаются на соседние процессоры в гиперкубе.

3.3.2. Отображение топологии решетки на гиперкуб

Отображение топологии решетки на гиперкуб может быть выполнено в рамках подхода, использованного для кольцевой структуры сети.

Код Грея для N=1	Код Грея для N=2	Код Грея для N=3	Номера процессоров	
			гиперкуба	кольца
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

Рис. 3.2. Отображение кольцевой топологии на гиперкуб при помощи кода Грея

Тогда для отображения решетки $2^r \times 2^s$ на гиперкуб размерности $N=r+s$ можно принять правило, что элементу решетки с координатами (i, j) соответствует процессор гиперкуба с номером:

$$G(i,r)||G(j,s),$$

где операция || означает конкатенацию кодов Грея.

3.4. Оценка трудоемкости операций передачи данных для кластерных систем

Для кластерных вычислительных систем (см. п. 1.2.2) одним из широко применяемых способов построения коммуникационной среды является использование *концентраторов* (*hub*) или *коммуникаторов* (*switch*) для объединения процессорных узлов кластера в единую вычислительную сеть. В этих случаях топология сети кластера представляет собой *полный граф*, в котором, однако, имеются определенные ограничения на одновременность выполнения коммуникационных операций. Так, при использовании концентраторов передача данных в каждый текущий момент может выполняться только между двумя процессорными узлами; коммуникаторы могут обеспечивать взаимодействие нескольких непересекающихся пар процессоров.

Другое часто применяемое решение при создании кластеров состоит в использовании *метода передачи пакетов* (часто реализуемого на основе стека протоколов TCP/IP) в качестве основного способа выполнения коммуникационных операций.

Если выбрать для дальнейшего анализа кластеры данного распространенного типа (топология в виде полного графа, пакетный способ передачи сообщений), то трудоемкость операции коммуникации между двумя

процессорными узлами может быть оценена в соответствии с выражением (*модель А*)

$$t_{пд}(m) = t_н + m^* t_к + t_с; \tag{3.23}$$

оценка подобного вида следует из соотношений для метода передачи пакетов при единичной длине пути передачи данных, т. е. при l=1. Отмечая возможность подобного подхода, вместе с этим можно заметить, что в рамках рассматриваемой модели время подготовки данных $t_н$ предполагается постоянным (не зависящим от объема передаваемых данных), время передачи служебных данных $t_с$ не зависит от количества передаваемых пакетов и т. п. Эти предположения не в полной мере соответствуют действительности, и временные оценки, получаемые в результате использования модели, могут не обладать необходимой точностью.

С учетом приведенных замечаний, схема построения временных оценок может быть уточнена; в рамках новой расширенной модели трудоемкость передачи данных между двумя процессорами определяется в соответствии со следующими выражениями (*модель В*):

$$t_{пд} = \begin{cases} t_{нач_0} + m \cdot t_{нач_1} + (m + V_с) \cdot t_к, & n = 1 \\ t_{нач_0} + (V_{\max} - V_с) \cdot t_{нач_1} + (m + V_с \cdot n) \cdot t_к, & n > 1 \end{cases}, \tag{3.24}$$

где $n = \lceil m/(V_{max} - V_с) \rceil$ есть количество пакетов, на которое разбивается передаваемое сообщение, величина V_{max} определяет максимальный размер пакета, который может быть доставлен в сети (по умолчанию для операционной системы MS Windows в сети Fast Ethernet V_{max} =1500 байт), а $V_с$ есть объем служебных данных в каждом из пересылаемых пакетов (для протокола TCP/IP, ОС Windows 2000 и сети Fast Ethernet $V_с$=78 байт). Поясним также, что в приведенных соотношениях константа $t_{нач_0}$ характеризует аппаратную составляющую латентности и зависит от параметров используемого сетевого оборудования, значение $t_{нач_1}$ задает время подготовки одного байта данных для передачи по сети. Как результат, величина латентности

$$t_н = t_{нач_0} + v \cdot t_{нач_1} \tag{3.25}$$

увеличивается линейно в зависимости от объема передаваемых данных. При этом предполагается, что подготовка данных для передачи второго и всех последующих пакетов может быть совмещена с пересылкой по сети предшествующих пакетов и латентность, тем самым, не может превышать величины:

$$t_н = t_{нач_0} + (V_{\max} - V_c) \cdot t_{нач_1}. \qquad (3.26)$$

Помимо латентности, в предлагаемых выражениях для оценки трудоемкости коммуникационной операции можно уточнить также правило вычисления времени передачи данных

$$(m + V_c \cdot n) \cdot t_к, \qquad (3.27)$$

что позволяет теперь учитывать эффект увеличения объема передаваемых данных при росте числа пересылаемых пакетов за счет добавления служебной информации (заголовков пакетов).

Завершая анализ проблемы построения теоретических оценок трудоемкости коммуникационных операций, следует отметить, что для практического применения перечисленных моделей необходимо выполнить оценку значений параметров используемых соотношений. В этом отношении полезным может оказаться использование и более простых способов вычисления временных затрат на передачу данных – одной из известных схем подобного вида является подход, в котором трудоемкость операции коммуникации между двумя процессорными узлами кластера оценивается в соответствии с выражением:

$$t_{пд}(m) = t_н + m t_к, \qquad (3.28)$$

это модель *C*, предложенная **Хокни** (*the Hockney model*) – см., например, [46].

Для проверки адекватности рассмотренных моделей реальным процессам передачи данных приведем результаты выполненных экспериментов в сети многопроцессорного кластера Нижегородского университета (компьютеры IBM PC Pentium 4 1300 МГц и сеть Fast Etherrnet). При проведении экспериментов для реализации коммуникационных операций использовалась библиотека MPI.

Часть экспериментов была выполнена для оценки параметров моделей:

- значение латентности $t_н$ для моделей *A* и *C* определялось как время передачи сообщения нулевой длины;
- величина пропускной способности *R* оценивалась максимальным значением скорости передачи данных, наблюдавшимся в экспериментах, т. е. величиной

$$R = \max_m (t_{пд}(m)/m),$$

и полагалось $t_к = 1/R$;

- значения величин $t_{нач_0}$ и $t_{нач_1}$ оценивались при помощи линейной аппроксимации времен передачи сообщений размера от 0 до V_{max}.

В ходе экспериментов осуществлялась передача данных между двумя узлами кластера, размер передаваемых сообщений варьировался от 0 до 8 Мб. Для получения более точных оценок выполнение каждой операции осуществлялось многократно (более 100 000 раз), после чего полученные результаты усреднялись. Для иллюстрации ниже приведен результат одного эксперимента, при проведении которого размер передаваемых сообщений изменялся от 2000 до 60 000 байт.

В табл. 3.2 приводится ряд числовых данных по погрешности рассмотренных моделей трудоемкости коммуникационных операций (величина погрешности дается в виде относительного отклонения от реального времени выполнения операции передачи данных).

Таблица 3.2. Погрешность моделей трудоемкости операций передачи данных (по результатам вычислительных экспериментов)

Объем сообщения (байт)	Время передачи (мкс)	Погрешность теоретической оценки времени передачи данных, %		
		Модель A	Модель B	Модель C
2000	495	33,45	7,93	34,80
10 000	1184	13,91	1,70	14,48
20 000	2055	8,44	0,44	8,77
30 000	2874	4,53	-1,87	4,76
40 000	3758	4,04	-1,38	4,22
50 000	4749	5,91	1,21	6,05
60 000	5730	6,97	2,73	7,09

Как можно заметить по результатам проведенных экспериментов, оценки трудоемкости операций передачи данных по модели *B* имеют меньшую погрешность.

Вместе с этим важно отметить, что для предварительного анализа временных затрат на выполнение коммуникационных операций точности модели *C* может оказаться достаточно. Кроме того, данная модель имеет наиболее простой вид среди всех рассмотренных. С учетом последнего обстоятельства, далее во всех последующих лекциях для оценки трудоемкости операций передачи данных будет применяться именно модель *C* (модель Хокни), при этом для модели будет использоваться форма записи, приведенная к обозначениям, которые приняты в работе Хокни [46]:

$$t_{n\partial}(m) = \alpha + m/\beta, \tag{3.29}$$

где α есть *латентность* сети передачи данных (т. е. $\alpha=t_н$), а β обозначает *пропускную способность* сети (т. е. $\beta=R=1/t_к$).

3.5. Краткий обзор лекции

Данная лекция посвящена оценке коммуникационной сложности параллельных алгоритмов.

В подразделе 3.1 представлена общая характеристика алгоритмов маршрутизации и методов передачи данных. Для подробного рассмотрения выделены **метод передачи сообщений** и **метод передачи пакетов**, для которых определены оценки времени выполнения коммуникационных операций.

В подразделе 3.2 определены основные типы операций передачи данных, выполняемых в ходе параллельных вычислений. К основным коммуникационным операциям относятся:

- передача данных между процессорами сети;
- передача данных от одного процессора всем остальным процессорам сети и двойственная ей операция приема на одном процессоре сообщений от всех остальных процессоров сети;
- передача данных от всех процессоров всем процессорам сети и двойственная ей операция приема сообщений на каждом процессоре от всех процессоров сети;
- обобщенная[1] передача данных от одного процессора всем остальным процессорам сети и обратная операция обобщенного приема сообщений на одном процессоре от всех остальных процессоров сети;
- обобщенная передача данных от всех процессоров всем процессорам сети.

Для всех перечисленных операций передачи данных рассмотрены алгоритмы их выполнения на примере топологий кольца, решетки и гиперкуба. Для каждого из представленных алгоритмов приведены оценки их временной трудоемкости как для метода передачи сообщений, так и для метода передачи пакетов.

В подразделе 3.3 рассмотрены **методы логического представления топологий** на основе конкретных (физических) межпроцессорных структур. Использование логических топологий позволяет получить более простое изложение для ряда алгоритмов передачи данных, снизить затраты на реализацию коммуникационных операций и т. п.

[1] Обобщение операции состоит в том, что разным процессорам рассылаются различные сообщения; для обратной операции приема все собираемые сообщения также являются разными.

В подразделе 3.4 более подробно обсуждаются модели, при помощи которых могут быть получены оценки времени выполнения операций передачи данных для кластерных вычислительных систем. Точность формирования временных оценок сравнивается при помощи проведения вычислительных экспериментов. По результатам экспериментов определена наиболее точная модель (модель *B*). Кроме того, отмечается, что для предварительного анализа временной трудоемкости коммуникационных операций целесообразно использовать более простую модель – модель *C* (**модель Хокни**).

3.6. Обзор литературы

В качестве дополнительного учебного материала для данной лекции могут быть рекомендованы работы [51, 63].

Вопросы построения моделей для оценки времени выполнения коммуникационных операций широко обсуждаются в литературе. При изучении лекции могут быть полезны работы [5, 28, 68]. *Модель Хокни* впервые была опубликована в [46]. Модель *B* из подраздела 3.4 представлена в работе [3].

3.7. Контрольные вопросы

1. Какие основные характеристики используются для оценки топологии сети передачи данных? Приведите значения характеристик для конкретных типов коммуникационных структур (полный граф, линейка, решетка и др.).
2. Какие основные методы применяются при маршрутизации передаваемых данных по сети?
3. В чем состоят основные методы передачи данных? Приведите для этих методов аналитические оценки времени выполнения.
4. Какие операции передачи данных могут быть выделены в качестве основных?
5. В чем состоят алгоритмы выполнения передачи данных от одного процессора всем процессорам сети для топологий кольца, решетки и гиперкуба? Приведите оценки временной трудоемкости для этих алгоритмов.
6. В чем состоят алгоритмы выполнения передачи данных от всех процессоров всем процессорам сети для топологий кольца, решетки и гиперкуба? Приведите оценки временной трудоемкости для этих алгоритмов.
7. В чем состоят возможные алгоритмы выполнения операции редукции? Какой из алгоритмов является наилучшим по времени выполнения?

8. В чем состоит алгоритм выполнения операции циклического сдвига?
9. В чем состоит полезность использования логических топологий? Приведите примеры алгоритмов логического представления структуры коммуникационной сети.
10. В чем состоит различие моделей для оценки времени выполнения операций передачи данных в кластерных вычислительных системах? Какая модель является более точной? Какая модель может быть использована для предварительного анализа временной трудоемкости коммуникационных операций?

3.8. Задачи и упражнения

1. Разработайте алгоритмы выполнения основных операций передачи данных для топологии сети в виде 3-мерной решетки.
2. Разработайте алгоритмы выполнения основных операций передачи данных для топологии сети в виде двоичного дерева.
3. Примените модель *B* из подраздела 3.4 для оценки временной сложности операций передачи данных. Сравните получаемые показатели.
4. Примените модель *C* из подраздела 3.4 для оценки временной сложности операций передачи данных. Сравните получаемые показатели.
5. Разработайте алгоритмы логического представления двоичного дерева для различных физических топологий сети.

Лекция 4. Принципы разработки параллельных методов

В лекции рассматриваются базовые принципы разработки параллельных алгоритмов. Описываются основные понятия, подробно разбираются все этапы создания и анализа параллельных алгоритмов. Приводится пример применения обсуждаемых методов.

Ключевые слова: граф «подзадачи – сообщения», граф «процессы – каналы», процесс, канал, выделение подзадач, определение информационных зависимостей, масштабирование и распределение подзадач по процессорам, параллелизм по данным, функциональный параллелизм.

Разработка алгоритмов (а в особенности методов параллельных вычислений) для решения сложных научно-технических задач часто представляет собой значительную проблему. Для снижения сложности рассматриваемой темы оставим в стороне математические аспекты разработки и доказательства сходимости алгоритмов – эти вопросы в той или иной степени изучаются в ряде «классических» математических учебных курсов. Здесь же мы будем полагать, что вычислительные схемы решения задач, рассматриваемых далее в качестве примеров, уже известны[1]. С учетом высказанных предположений последующие действия для определения эффективных способов организации параллельных вычислений могут состоять в следующем:

- выполнить анализ имеющихся вычислительных схем и осуществить их разделение (*декомпозицию*) на части (*подзадачи*), которые могут быть реализованы в значительной степени независимо друг от друга;
- выделить для сформированного набора подзадач *информационные взаимодействия*, которые должны осуществляться в ходе решения исходной поставленной задачи;
- определить необходимую (или доступную) для решения задачи вычислительную систему и выполнить *распределение* имеющего набора подзадач между процессорами системы.

[1] Несмотря на то, что для многих научно-технических задач на самом деле известны не только последовательные, но и параллельные методы решения, данное предположение является, конечно, очень сильным, поскольку для новых возникающих задач, требующих для своего решения большого объема вычислений, процесс разработки алгоритмов составляет существенную часть всех выполняемых работ.

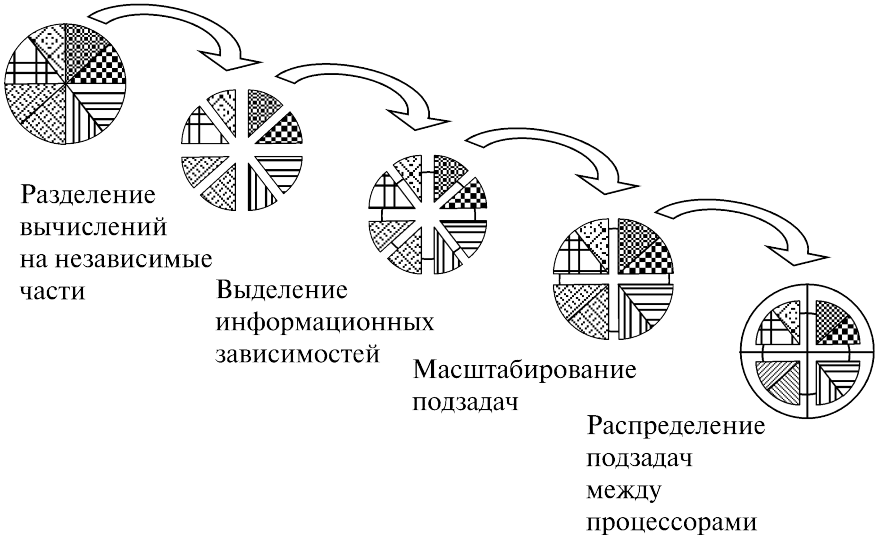


Рис. 4.1. Общая схема разработки параллельных алгоритмов

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

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

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

масштабирование разрабатываемого алгоритма и выделены в качестве отдельного этапа проектирования параллельных вычислений.

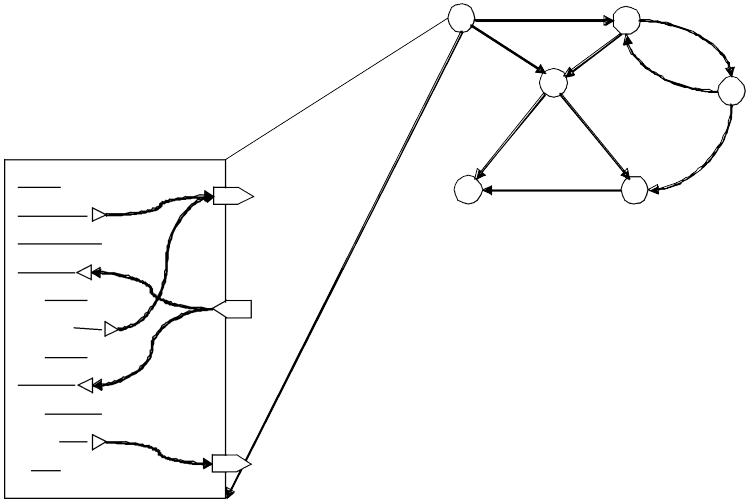
Чтобы применить получаемый в конечном итоге параллельный метод, необходимо выполнить разработку *программ* для решения сформированного набора подзадач и разместить разработанные программы по процессорам в соответствии с выбранной схемой распределения подзадач. Для проведения вычислений программы запускаются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

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

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

4.1. Моделирование параллельных программ

Рассмотренная схема проектирования и реализации параллельных вычислений дает способ понимания параллельных алгоритмов и программ. На стадии проектирования параллельный метод может быть представлен в виде *графа «подзадачи – сообщения»*, который представляет собой не что иное, как укрупненное (агрегированное) представление графа информационных зависимостей (графа «операции – операнды» – см. лекцию 2). Аналогично на стадии выполнения для описания параллельной программы может быть использована модель в виде *графа «процессы – ка-*



◁ [▷] — операции приема (передачи)

 — входные (выходные) каналы для взаимодействия процессов

Рис. 4.2. Модель параллельной программы в виде графа «процессы – каналы»

налы», в которой вместо подзадач используется понятие процессов, а информационные зависимости заменяются каналами передачи сообщений. Дополнительно на этой модели может быть показано распределение процессов по процессорам вычислительной системы, если количество подзадач превышает число процессоров – см. рис. 4.2.

Использование двух моделей параллельных вычислений[1] позволяет лучше разделить проблемы, которые проявляются при разработке параллельных методов. Первая модель – граф «подзадачи – сообщения» – позволяет сосредоточиться на вопросах выделения подзадач одинаковой вычислительной сложности, обеспечивая при этом низкий уровень информационной зависимости между подзадачами. Вторая модель – граф «процессы – каналы» – концентрирует внимание на вопросах распределения подзадач по процессорам, обеспечивая еще одну возможность снижения трудоемкости информационных взаимодействий между подзадачами за

[1] В [32] рассматривается только одна модель – модель «задача – канал» для описания параллельных вычислений, которая занимает некоторое промежуточное положение по сравнению с изложенными здесь моделями. Так, в модели «задача – канал» не учитывается возможность использования одного процессора для решения нескольких подзадач одновременно.

счет размещения на одних и тех же процессорах интенсивно взаимодействующих процессов. Кроме того, эта модель позволяет лучше анализировать эффективность разработанного параллельного метода и обеспечивает возможность более адекватного описания процесса выполнения параллельных вычислений.

Дадим дополнительные пояснения для используемых понятий в модели «процессы – каналы»:

- под *процессом* будем понимать выполняемую на процессоре *программу*, которая использует для своей работы часть локальной *памяти* процессора и содержит ряд *операций приема/передачи* данных для организации информационного взаимодействия с другими выполняемыми процессами параллельной программы;
- *канал передачи данных* с логической точки зрения может рассматриваться как *очередь сообщений*, в которую один или несколько процессов могут отправлять пересылаемые данные и из которой процесс-адресат может извлекать сообщения, отправляемые другими процессами.

В общем случае, можно считать, что каналы возникают динамически в момент выполнения первой операции приема/передачи с каналом. По степени общности канал может соответствовать одной или нескольким командам приема данных процесса-получателя; аналогично, при передаче сообщений канал может использоваться одной или несколькими командами передачи данных одного или нескольких процессов. Для снижения сложности моделирования и анализа параллельных методов будем предполагать, что емкость каналов является неограниченной и, как результат, операции передачи данных выполняются практически без задержек простым копированием сообщений в канал. С другой стороны, операции приема сообщений могут приводить к задержкам (*блокировкам*), если запрашиваемые из канала данные еще не были отправлены процессами – источниками сообщений.

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

4.2. Этапы разработки параллельных алгоритмов

Рассмотрим более подробно изложенную выше методику разработки параллельных алгоритмов. В значительной степени данная методика опирается на подход, впервые разработанный в [32], и, как отмечалось ранее, включает этапы выделения подзадач, определения информационных

зависимостей, масштабирования и распределения подзадач по процессорам вычислительной системы (см. рис. 4.1). Для демонстрации приводимых рекомендаций далее будет использоваться учебная задача поиска максимального значения среди элементов матрицы A (такая задача возникает, например, при численном решении систем линейных уравнений для определения ведущего элемента метода Гаусса):

$$y = \max_{1 \le i, j \le N} a_{ij} . \qquad (4.1)$$

Данная задача носит полностью иллюстративный характер, и после рассмотрения этапов разработки в оставшейся части лекции будет приведен более полный пример использования данной методики для разработки параллельных алгоритмов. Кроме того, данная схема разработки будет применена и при изложении всех рассматриваемых далее методов параллельных вычислений.

4.2.1. Разделение вычислений на независимые части

Выбор способа разделения вычислений на независимые части основывается на анализе вычислительной схемы решения исходной задачи. Требования, которым должен удовлетворять выбираемый подход, обычно состоят в обеспечении равного объема вычислений в выделяемых подзадачах и минимума информационных зависимостей между этими подзадачами (при прочих равных условиях нужно отдавать предпочтение редким операциям передачи сообщений большего размера по сравнению с частыми пересылками данных небольшого объема). В общем случае, проведение анализа и выделение задач представляет собой достаточно сложную проблему – ситуацию помогает разрешить существование двух часто встречающихся типов вычислительных схем (см. рис. 4.3).

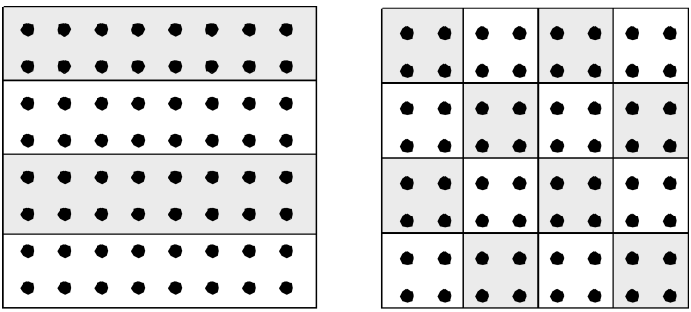


Рис. 4.3. Разделение данных матрицы: а) ленточная схема, б) блочная схема

Для большого класса задач вычисления сводятся к выполнению однотипной обработки большого набора данных – к такому классу задач относятся, например, матричные вычисления, численные методы решения уравнений в частных производных и др. В этом случае говорят, что существует **параллелизм по данным**, и выделение подзадач сводится к разделению имеющихся данных. Так, например, для рассматриваемой учебной задачи поиска максимального значения при формировании подзадач исходная матрица может быть разделена на отдельные строки (или последовательные группы строк) – так называемая *ленточная схема* разделения данных (см. рис. 4.3) – либо на прямоугольные наборы элементов – *блочная схема* разделения данных. Для большого количества решаемых задач разделение вычислений по данным приводит к порождению одно-, дву- и трехмерных наборов подзадач, в которых информационные связи существуют только между ближайшими соседями (такие схемы обычно именуются сетками или решетками).

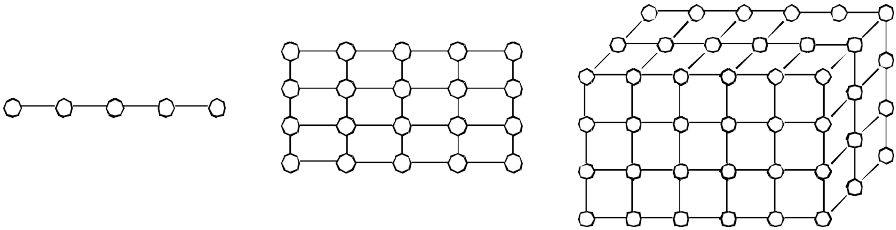


Рис. 4.4. Регулярные одно-, дву- и трехмерные структуры базовых подзадач после декомпозиции данных

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

Важный вопрос при выделении подзадач состоит в выборе нужного *уровня декомпозиции* вычислений. Формирование максимально возможного количества подзадач обеспечивает использование предельно достижимого уровня параллелизма решаемой задачи, однако затрудняет анализ параллельных вычислений. Применение при декомпозиции вычислений

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

Для рассматриваемой учебной задачи достаточный уровень декомпозиции может состоять, например, в разделении матрицы на множество отдельных строк и получении на этой основе набора подзадач поиска максимальных значений в отдельных строках; порождаемая при этом структура информационных связей соответствует линейному графу (см. рис. 4.5).

Для оценки корректности этапа разделения вычислений на независимые части можно воспользоваться контрольным списком вопросов, предложенных в [32]:

- выполненная декомпозиция не увеличивает объем вычислений и необходимый объем памяти?
- возможна ли при выбранном способе декомпозиции равномерная загрузка всех имеющихся процессоров?
- достаточно ли выделенных частей процесса вычислений для эффективной загрузки имеющихся процессоров (с учетом возможности увеличения их количества)?

4.2.2. Выделение информационных зависимостей

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

При проведении анализа информационных зависимостей между подзадачами следует различать (предпочтительные формы информационного взаимодействия выделены подчеркиванием):

- *локальные* и *глобальные схемы* передачи данных – для локальных схем передачи данных в каждый момент времени выполняются только между небольшим числом подзадач (располагаемых, как правило, на соседних процессорах), для глобальных операций передачи данных в процессе коммуникации принимают участие все подзадачи;
- *структурные* и *произвольные способы взаимодействия* – для структурных способов организация взаимодействий приводит к формированию некоторых стандартных схем коммуникации (например, в виде кольца, прямоугольной решетки и т. д.), для произвольных структур взаимодействия схема выполняемых операций передач данных не носит характера однородности;
- *статические* или *динамические схемы* передачи данных – для статических схем моменты и участники информационного взаимодействия фиксируются на этапах проектирования и разработки параллельных программ, для динамического варианта взаимодействия структура операции передачи данных определяется в ходе выполняемых вычислений;
- *синхронные* и *асинхронные способы взаимодействия* – для синхронных способов операции передачи данных выполняются только при готовности всех участников взаимодействия и завершаются только после полного окончания всех коммуникационных действий, при асинхронном выполнении операций участники взаимодействия могут не дожидаться полного завершения действий по передаче данных. Для представленных способов взаимодействия достаточно сложно выделить предпочтительные формы организации передачи

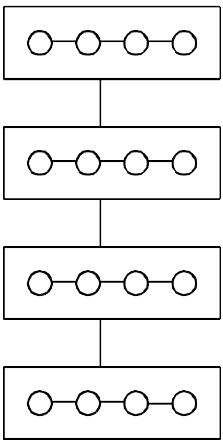


Рис. 4.5. Структура информационных связей учебной задачи

данных: синхронный вариант, как правило, более прост для применения, в то время как асинхронный способ часто позволяет существенно снизить временные задержки, вызванные операциями информационного взаимодействия.

Как уже отмечалось в предыдущем пункте, для учебной задачи поиска максимального значения при использовании в качестве базовых элементов подзадач поиска максимальных значений в отдельных строках исходной матрицы структура информационных связей имеет вид, представленный на рис. 4.5.

Для оценки правильности этапа выделения информационных зависимостей можно воспользоваться контрольным списком вопросов, предложенным в [32]:

- соответствует ли вычислительная сложность подзадач интенсивности их информационных взаимодействий?
- является ли одинаковой интенсивность информационных взаимодействий для разных подзадач?
- является ли схема информационного взаимодействия локальной?
- не препятствует ли выявленная информационная зависимость параллельному решению подзадач?

4.2.3. Масштабирование набора подзадач

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

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

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

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

Список контрольных вопросов, предложенный в [32] для оценки правильности этапа масштабирования, выглядит следующим образом:

- не ухудшится ли локальность вычислений после масштабирования имеющегося набора подзадач?
- имеют ли подзадачи после масштабирования одинаковую вычислительную и коммуникационную сложность?
- соответствует ли количество задач числу имеющихся процессоров?
- зависят ли параметрически правила масштабирования от количества процессоров?

4.2.4. Распределение подзадач между процессорами

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

Основной показатель успешности выполнения данного этапа – *эффективность использования процессоров*, определяемая как относительная доля времени, в течение которого процессоры использовались для вычислений, связанных с решением исходной задачи. Пути достижения хороших результатов в этом направлении остаются прежними: как и ранее, необходимо обеспечить равномерное распределение вычислительной нагрузки между процессорами и минимизировать количество сообщений, передаваемых между ними. Точно так же как и на предшествующих этапах проектирования, оптимальное решение проблемы распределения подзадач между процессорами основывается на анализе информационной связности графа «подзадачи – сообщения». Так, в частности, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных.

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

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

Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений может изменяться в ходе решения задачи. Причиной этого могут быть, например, неоднородные сетки при решении уравнений в частных производных, разреженность матриц и т. п.[1]. Кроме того, используемые на этапах проектирования оценки вычислительной сложности решения подзадач могут иметь приближенный характер, и, наконец, количество подзадач может изменяться в ходе вычислений. В таких ситуациях может потребоваться перераспределение базовых подзадач между процессорами уже непосредственно в ходе выполнения параллельной программы (или, как обычно говорят, придется выполнить *динамическую балансировку* вычислительной нагрузки). Данные вопросы являются одними из наиболее сложных (и наиболее интересных) в области параллельных вычислений – к сожалению, их рассмотрение выходит за рамки нашего учебного материала (дополнительная информация может быть получена, например, в [24,75]).

В качестве примера дадим краткую характеристику широко используемого способа динамического управления распределением вычислительной нагрузки, обычно именуемого *схемой «менеджер – исполнитель»* (*the manager-worker scheme*). При использовании данного подхода предполагается, что подзадачи могут возникать и завершаться в ходе вычислений, при этом информационные взаимодействия между подзадачами ли-

[1] Можно отметить, что даже для нашей простой учебной задачи может наблюдаться различная вычислительная сложность сформированных базовых задач. Так, например, количество операций при поиске максимального значения для строки, в которой максимальное значение имеет первый элемент, и строки, в которой значения являются упорядоченными по возрастанию, будет различаться в два раза.

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

Предложенный в [32] перечень контрольных вопросов для проверки этапа распределения подзадач состоит в следующем:

- не приводит ли распределение нескольких задач на один процессор к росту дополнительных вычислительных затрат?
- существует ли необходимость динамической балансировки вычислений?
- не является ли процессор-менеджер «узким» местом при использовании схемы «менеджер – исполнитель»?

4.3. Параллельное решение гравитационной задачи *N* тел

Многие задачи в области физики сводятся к операциям обработки данных для каждой пары объектов имеющейся физической системы. Такой задачей является, в частности, проблема, широко известная в литературе как гравитационная задача *N* тел (или просто *задача N тел*) – см., например, [5]. В самом общем виде задача может быть описана следующим образом.

Пусть дано большое количество тел (планет, звезд и т. д.), для каждого из которых известна масса, начальное положение и скорость. Под действием гравитации положение тел меняется, и требуемое решение задачи состоит в моделировании динамики изменения системы *N* тел на протяжении некоторого задаваемого интервала времени. Для проведения такого моделирования заданный интервал времени обычно разбивается на временные отрезки небольшой длительности и далее на каждом шаге моделирования вычисляются силы, действующие на каждое тело, а затем обновляются скорости и положения тел.

Очевидный алгоритм решения задачи *N* тел состоит в рассмотрении на каждом шаге моделирования всех пар объектов физической системы и выполнении для каждой получаемой пары всех необходимых расчетов.

Как результат, при таком подходе время выполнения одной итерации моделирования будет составлять[1]

$$T_1 = \tau\, N(N-1)/2\,, \tag{4.2}$$

где τ есть время перевычисления параметров одной пары тел.

Как следует из приведенного описания, вычислительная схема рассмотренного алгоритма является сравнительно простой, что позволяет использовать задачу N тел в качестве еще одной наглядной демонстрации применения методики разработки параллельных алгоритмов.

4.3.1. Разделение вычислений на независимые части

Выбор способа разделения вычислений не вызывает каких-либо затруднений – очевидный подход состоит в выборе в качестве базовой подзадачи всего набора вычислений, связанных с обработкой данных какого-либо одного тела физической системы.

4.3.2. Выделение информационных зависимостей

Выполнение вычислений, связанных с каждой подзадачей, становится возможным только в случае, когда в подзадачах имеются данные (положение и скорости передвижения) обо всех телах физической системы. Как результат, перед началом каждой итерации моделирования каждая подзадача должна получить все необходимые сведения от всех других подзадач системы. Такая процедура передачи данных, как отмечалось в лекции 3, именуется *операцией сбора данных* (*single-node gather*). В рассматриваемом алгоритме данная операция должна быть выполнена для каждой подзадачи – такой вариант передачи данных обычно именуется *операцией обобщенного сбора данных* (*multi-node gather* или *all gather*).

Определение требований к необходимым результатам информационного обмена не приводит к однозначному установлению нужного информационного обмена между подзадачами – достижение требуемых результатов может быть обеспечено при помощи разных алгоритмов выполнения операции обобщенного сбора данных.

Наиболее простой способ выполнения необходимого информационного обмена состоит в реализации последовательности шагов, на каждом из которых все имеющиеся подзадачи разбиваются попарно и обмен данными осуществляется между подзадачами образовавшихся пар. При

[1] Следует отметить, что для решения задачи N тел существуют и более эффективные последовательные алгоритмы, однако их изучение может потребовать достаточно больших усилий. С учетом данного обстоятельства для дальнейшего рассмотрения выбирается именно данный «очевидный» (но не самый быстрый) метод.

надлежащей организации попарного разделения подзадач (N-1)-кратное повторение описанных действий приведет к полной реализации требуемой операции сбора данных.

Рассмотренный выше метод организации информационного обмена является достаточно трудоемким – для сбора всех необходимых данных требуется провести N-1 итерацию, на каждой из которых выполняется одновременно $N/2$ операций передачи данных. Для сокращения требуемого количества итераций можно обратить внимание на факт, что после выполнения первого шага операции сбора данных подзадачи будут уже содержать не только свои данные, но и данные подзадач, с которыми они образовывали пары. Как результат, на второй итерации сбора данных можно будет образовывать пары подзадач для обмена данными сразу о двух телах физической системы – тем самым, после завершения второй итерации каждая подзадача будет содержать сведения о четырех телах системы и т. д. Как можно заметить, данный способ реализации обменов позволяет завершить необходимую процедуру за $\log_2 N$ итераций. Следует отметить, что при этом объем пересылаемых данных в каждой операции обмена удваивается от итерации к итерации: на первой итерации между подзадачами пересылаются данные об одном теле системы, на второй – о двух телах и т. д.

4.3.3. Масштабирование и распределение подзадач по процессорам

Как правило, число тел физической системы N значительно превышает количество процессоров p. Поэтому рассмотренные ранее подзадачи следует укрупнить, объединив в рамках одной подзадачи вычисления для группы из N/p тел. После проведения подобной агрегации число подзадач и количество процессоров будет совпадать и при распределении подзадач между процессорами останется лишь обеспечить наличие прямых коммуникационных линий между процессорами с подзадачами, у которых имеются информационные обмены при выполнении операции сбора данных.

4.3.4. Анализ эффективности параллельных вычислений

Оценим эффективность разработанных способов параллельных вычислений для решения задачи N тел. Поскольку предложенные варианты отличаются только методами выполнения информационных обменов, для сравнения подходов достаточно определить длительность операции обобщенного сбора данных. Используем для оценки времени передачи сообщений модель, предложенную Хокни (см. лекцию 3), — тогда длительность выполнения операции сбора данных для первого варианта параллельных вычислений может быть выражена как

$$T_p^1(comm) = (p-1)(\alpha + m(N/p)/\beta), \qquad (4.3)$$

где α и β есть параметры модели Хокни (латентность и пропускная способность сети передачи данных), а m задает объем пересылаемых данных для одного тела физической системы.

Для второго способа информационного обмена, как уже отмечалось ранее, объем пересылаемых данных на разных итерациях операции сбора данных различается. На первой итерации объем пересылаемых сообщений составляет Nm/p, на второй этот объем увеличивается вдвое и оказывается равным $2Nm/p$ и т. д. В общем случае, для итерации с номером i объем сообщений оценивается как $2^{i-1}Nm/p$. Как результат, длительность выполнения операции сбора данных в этом случае может быть определена при помощи следующего выражения:

$$T_p^2(comm) = \sum_{i=1}^{\log p}(\alpha + 2^{i-1} m(N/p)/\beta) = \alpha \log p + m\,(N/p)(p-1)/\beta. \qquad (4.4)$$

Сравнение полученных выражений показывает, что второй разработанный способ параллельных вычислений имеет существенно более высокую эффективность, требует меньших коммуникационных затрат и допускает лучшую масштабируемость при увеличении количества используемых процессоров.

4.4. Краткий обзор лекции

В лекции была рассмотрена методика разработки параллельных алгоритмов, предложенная в [32]. Она включает в себя этапы выделения подзадач, определения информационных зависимостей, масштабирования и распределения подзадач по процессорам вычислительной системы. При использовании методики предполагается, что вычислительная схема решения рассматриваемой задачи уже является известной. Основные требования, которые должны быть обеспечены при разработке параллельных алгоритмов, состоят в равномерной загрузке процессоров при низком информационном взаимодействии сформированного множества подзадач.

Для описания получаемых в ходе разработки вычислительных параллельных схем рассмотрены две модели. Первая из них – модель «подзадачи – сообщения» может быть использована на стадии проектирования параллельных алгоритмов, вторая — модель «процессы – каналы» – может быть применена на стадии реализации методов в виде параллельных программ.

В завершение раздела показывается применение рассмотренной методики разработки параллельных алгоритмов на примере решения гравитационной задачи N тел.

4.5. Обзор литературы

Рассмотренная в лекции методика разработки параллельных алгоритмов впервые была предложена в [32]. В этой работе изложение методики проводится более детально, кроме того, в ней содержится несколько примеров ее использования для разработки параллельных методов для решения ряда вычислительных задач.

Полезной при рассмотрении вопросов проектирования и разработки параллельных алгоритмов может оказаться также работа [63].

Гравитационная задача N тел более подробно рассматривается в [5].

4.6. Контрольные вопросы

1. В чем состоят исходные предположения для возможности применения рассмотренной в лекции методики разработки параллельных алгоритмов?
2. Каковы основные этапы проектирования и разработки методов параллельных вычислений?
3. Как определяется модель «подзадачи – сообщения»?
4. Как определяется модель «процессы – каналы»?
5. Какие основные требования должны быть обеспечены при разработке параллельных алгоритмов?
6. В чем состоят основные действия на этапе выделения подзадач?
7. Каковы основные действия на этапе определения информационных зависимостей?
8. В чем состоят основные действия на этапе масштабирования имеющегося набора подзадач?
9. В чем состоят основные действия на этапе распределения подзадач по процессорам вычислительной системы?
10. Как происходит динамическое управление распределением вычислительной нагрузки при помощи схемы «менеджер – исполнитель»?
11. Какой метод параллельных вычислений был разработан для решения гравитационной задачи N тел?
12. Какой способ выполнения операции обобщенного сбора данных является более эффективным?

4.7. Задачи и упражнения

1. Разработайте схему параллельных вычислений, используя рассмотренную в разделе методику проектирования и разработки параллельных методов:

- для задачи поиска максимального значения среди минимальных элементов строк матрицы (такая задача имеет место для решения матричных игр)

$$y = \max_{1 \le i \le N} \min_{1 \le j \le N} a_{ij}$$

(обратите особое внимание на ситуацию, когда число процессоров превышает размер матрицы, т. е. $p > N$);

- для задачи вычисления определенного интеграла с использованием метода прямоугольников

$$y = \int_a^b f(x)dx \approx h \sum_{i=0}^{N-1} f_i,\ f_i = f(x_i),\ x_i = a + i\,h,\ h = (b-a)/N$$

(описание методов интегрирования дано, например, в [47]).

2. Разработайте схему параллельных вычислений для задачи умножения матрицы на вектор, используя рассмотренную в разделе методику проектирования и разработки параллельных методов.

Лекция 5. Параллельное программирование на основе MPI

В лекции рассматривается стандарт для программирования в системах с распределенной памятью MPI. Дается обзор истории возникновения и развития стандарта, а также перечисляются его основные возможности. Приводятся примеры программ, использующих рассматриваемый стандарт.

Ключевые слова: многопроцессорная система, распределенная память, MPI, параллельная программа, процесс, передача сообщений, парные и коллективные операции пересылки данных, режимы передачи, группы процессов, производный тип данных, виртуальные топологии.

В вычислительных системах с распределенной памятью (см. рис. 1.6) процессоры работают независимо друг от друга. Для организации параллельных вычислений в таких условиях необходимо иметь возможность *распределять вычислительную нагрузку и организовать информационное взаимодействие* (*передачу данных*) между процессорами.

Решение всех перечисленных вопросов и обеспечивает интерфейс передачи данных (*message passing interface – MPI*).

В общем плане, для распределения вычислений между процессорами необходимо проанализировать алгоритм решения задачи, выделить информационно независимые фрагменты вычислений, провести их программную реализацию и затем разместить полученные части программы на разных процессорах. В рамках MPI принят более простой подход – для решения поставленной задачи разрабатывается одна программа и эта единственная программа запускается одновременно на выполнение на всех имеющихся процессорах. При этом для того чтобы избежать идентичности вычислений на разных процессорах, можно, во-первых, подставлять разные данные для программы на разных процессорах, а во-вторых, использовать имеющиеся в MPI средства для идентификации процессора, на котором выполняется программа (тем самым предоставляется возможность организовать различия в вычислениях в зависимости от используемого программой процессора).

Подобный способ организации параллельных вычислений получил наименование *модели «одна программа множество процессов»* (*single program multiple processes or SPMP*[1]).

[1] В литературе чаще упоминается модель «одна программа множество данных» (*single program multiple data* or *SPMD*). Применительно к MPI более логичным представляется использование сочетания SPMP.

Для организации информационного взаимодействия между процессорами в самом минимальном варианте достаточно операций приема и передачи данных (при этом, конечно, должна существовать техническая возможность коммуникации между процессорами – *каналы* или *линии связи*). В MPI существует целое множество операций передачи данных. Они обеспечивают разные способы пересылки данных, реализуют практически все рассмотренные в лекции 3 коммуникационные операции. Именно данные возможности являются наиболее сильной стороной MPI (об этом, в частности, свидетельствует и само название MPI).

Следует отметить, что попытки создания программных средств передачи данных между процессорами начали предприниматься практически сразу с появлением локальных компьютерных сетей – ряд таких средств, представлен, например, в работах [2, 5, 24] и многих других. Однако подобные средства часто были неполными и, самое главное, являлись несовместимыми. Таким образом, одна из самых серьезных проблем в программировании – переносимость программ при переводе программного обеспечения на другие компьютерные системы – проявлялась при разработке параллельных программ в максимальной степени. Как результат, уже с 90-х годов стали предприниматься усилия по стандартизации средств организации передачи сообщений в многопроцессорных вычислительных системах. Началом работ, непосредственно приведших к появлению MPI, послужило проведение рабочего совещания по стандартам для передачи сообщений в среде распределенной памяти (the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992). По итогам совещания была образована рабочая группа, позднее преобразованная в международное сообщество MPI Forum, результатом деятельности которых явилось создание и принятие в 1994 г. стандарта *интерфейса передачи сообщений* (*message passing interface – MPI*) версии 1.0. В последующие годы стандарт MPI последовательно развивался. В 1997 г. был принят стандарт MPI версии 2.0.

Итак, теперь можно пояснить, что означает понятие MPI. Во-первых, MPI – это стандарт, которому должны удовлетворять средства организации передачи сообщений. Во-вторых, MPI – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI. Так, по стандарту, эти программные средства должны быть организованы в виде библиотек программных функций (*библиотеки MPI*) и должны быть доступны для наиболее широко используемых алгоритмических языков C и Fortran. Подобную «двойственность» MPI следует учитывать при использовании терминологии. Как правило, аббревиатура MPI применяется при упоминании стандарта, а сочетание «библиотека MPI» указывает на ту или иную программ-

ную реализацию стандарта. Однако достаточно часто для краткости обозначение MPI используется и для библиотек MPI, и, тем самым, для правильной интерпретации термина следует учитывать контекст.

Вопросы, связанные с разработкой параллельных программ с применением MPI, достаточно широко рассмотрены в литературе – краткий обзор полезных материалов содержится в конце данной лекции. Здесь же, еще не приступая к изучению MPI, приведем ряд его важных положительных моментов:

- MPI позволяет в значительной степени снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами – параллельная программа, разработанная на алгоритмическом языке C или Fortran с использованием библиотеки MPI, как правило, будет работать на разных вычислительных платформах;
- MPI содействует повышению эффективности параллельных вычислений, поскольку в настоящее время практически для каждого типа вычислительных систем существуют реализации библиотек MPI, в максимальной степени учитывающие возможности компьютерного оборудования;
- MPI уменьшает, в определенном плане, сложность разработки параллельных программ, т. к., с одной стороны, большая часть рассмотренных в лекции 3 основных операций передачи данных предусматривается стандартом MPI, а с другой стороны, уже имеется большое количество библиотек параллельных методов, созданных с использованием MPI.

5.1. MPI: основные понятия и определения

Рассмотрим ряд понятий и определений, являющихся основополагающими для стандарта MPI.

5.1.1. Понятие параллельной программы

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (*модель SPMP*). Данный про-

граммный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска параллельной программы на всех используемых процессорах. Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках C или Fortran с применением той или иной реализации библиотеки MPI.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений не может меняться без применения специальных, но редко задействуемых средств динамического порождения процессов и управления ими, появившихся в стандарте MPI версии 2.0. Все процессы программы последовательно перенумерованы от 0 до p-1, где p есть общее количество процессов. *Номер процесса именуется рангом процесса*.

5.1.2. Операции передачи данных

Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются *парные* (*point-to-point*) операции между двумя процессами и *коллективные* (*collective*) коммуникационные действия для одновременного взаимодействия нескольких процессов.

Для выполнения парных операций могут использоваться разные режимы передачи, среди которых синхронный, блокирующий и др. – полное рассмотрение возможных режимов передачи будет выполнено в подразделе 5.3.

Как уже отмечалось ранее, в стандарт MPI включено большинство основных коллективных операций передачи данных – см. подразделы 5.2 и 5.4.

5.1.3. Понятие коммуникаторов

Процессы параллельной программы объединяются в *группы*. Другим важным понятием MPI, описывающим набор процессов, является понятие **коммуникатора**. Под коммуникатором в MPI понимается специально создаваемый служебный объект, который объединяет в своем составе группу процессов и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных.

Парные операции передачи данных выполняются только для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов одного коммуникатора. Как результат, указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммуникаторы. Один и тот же процесс может принадлежать разным группам и коммуникаторам. Все имеющиеся в параллельной программе процессы входят в состав конструируемого по умолчанию коммуникатора с идентификатором `MPI_COMM_WORLD`.

В версии 2.0 стандарта появилась возможность создавать глобальные коммуникаторы (*intercommunicator*), объединяющие в одну структуру пару групп при необходимости выполнения коллективных операций между процессами из разных групп.

Подробное рассмотрение возможностей MPI для работы с группами и коммуникаторами будет выполнено в подразделе 5.6.

5.1.4. Типы данных

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать *тип пересылаемых данных*. MPI содержит большой набор *базовых типов данных*, во многом совпадающих с типами данных в алгоритмических языках C и Fortran. Кроме того, в MPI имеются возможности создания новых *производных типов* данных для более точного и краткого описания содержимого пересылаемых сообщений.

Подробное рассмотрение возможностей MPI для работы с производными типами данных будет выполнено в подразделе 5.5.

5.1.5. Виртуальные топологии

Как уже отмечалось ранее, парные операции передачи данных могут быть выполнены между любыми процессами одного и того же коммуникатора, а в коллективной операции принимают участие все процессы коммуникатора. Логическая топология линий связи между процессами имеет структуру полного графа (независимо от наличия реальных физических каналов связи между процессорами).

Вместе с этим (и это уже отмечалось в лекции 3), для изложения и последующего анализа ряда параллельных алгоритмов целесообразно логическое представление имеющейся коммуникационной сети в виде тех или иных топологий.

В MPI имеется возможность представления множества процессов в виде решетки произвольной размерности (см. рис. 1.7). При этом граничные процессы решеток могут быть объявлены соседними, и, тем самым, на основе решеток могут быть определены структуры типа *тор*.

Кроме того, в MPI имеются средства и для формирования логических (виртуальных) топологий любого требуемого типа. Подробное рас-

смотрение возможностей MPI для работы с топологиями будет выполнено в подразделе 5.7.

И, наконец, последний ряд замечаний перед началом рассмотрения MPI:

- описание функций и все приводимые примеры программ будут представлены на алгоритмическом языке C; особенности использования MPI для алгоритмического языка Fortran будут даны в п. 5.8.1;
- краткая характеристика имеющихся реализаций библиотек MPI и общее описание среды выполнения MPI-программ будут рассмотрены в п. 5.8.2;
- основное изложение возможностей MPI будет ориентировано на стандарт версии 1.2 (так называемый *MPI*-1), нововведения стандарта версии 2.0 будут представлены в п. 5.8.3.

Приступая к изучению MPI, можно отметить, что, с одной стороны, MPI достаточно сложен – в стандарте MPI предусматривается наличие более чем 120 функций. С другой стороны, структура MPI является тщательно продуманной – разработка параллельных программ может быть начата уже после рассмотрения всего лишь 6 функций MPI. Все дополнительные возможности MPI могут осваиваться по мере роста сложности разрабатываемых алгоритмов и программ. Именное в таком стиле – от простого к сложному – и будет далее представлен весь учебный материал по MPI.

5.2. Введение в разработку параллельных программ с использованием MPI

5.2.1. Основы MPI

Приведем минимально необходимый набор функций MPI, достаточный для разработки сравнительно простых параллельных программ.

5.2.1.1. Инициализация и завершение MPI-программ

Первой вызываемой функцией MPI должна быть функция:

```
int MPI_Init(int *argc, char ***argv),
```

где

— **argc** — указатель на количество параметров командной строки,

— **argv** — параметры командной строки,

применяемая для инициализации среды выполнения MPI-программы. Параметрами функции являются количество аргументов в командной

строке и адрес указателя на массив символов текста самой командной строки.

Последней вызываемой функцией MPI обязательно должна являться функция:

```
int MPI_Finalize(void).
```

Как результат, можно отметить, что структура параллельной программы, разработанная с использованием MPI, должна иметь следующий вид:

```
#include "mpi.h"
int main(int argc, char *argv[]) {
 <программный код без использования функций MPI>
 MPI_Init(&agrc, &argv);
 <программный код с использованием функций MPI>
 MPI_Finalize();
 <программный код без использования функций MPI>
 return 0;
}
```

Следует отметить:

- файл *mpi.h* содержит определения именованных констант, прототипов функций и типов данных библиотеки MPI;
- функции `MPI_Init` и `MPI_Finalize` являются обязательными и должны быть выполнены (и только один раз) каждым процессом параллельной программы;
- перед вызовом `MPI_Init` может быть использована функция `MPI_Initialized` для определения того, был ли ранее выполнен вызов `MPI_Init`, а после вызова `MPI_Finalize` — `MPI_Finalized`[1] аналогичного предназначения.

Рассмотренные примеры функций дают представление синтаксиса именования функций в MPI. Имени функции предшествует префикс MPI, далее следует одно или несколько слов названия, первое слово в имени функции начинается с заглавного символа, слова разделяются знаком подчеркивания. Названия функций MPI, как правило, поясняют назначение выполняемых функцией действий.

5.2.1.2. Определение количества и ранга процессов

Определение *количества процессов* в выполняемой параллельной программе осуществляется при помощи функции:

[1] Эта функция появилась только в стандарте MPI версии 2.0.

```
int MPI_Comm_size(MPI_Comm comm, int *size),
```

где

— **comm** — коммуникатор, размер которого определяется,
— **size** — определяемое количество процессов в коммуникаторе.

Для определения ранга процесса используется функция:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank),
```

где

— **comm** — коммуникатор, в котором определяется ранг процесса,
— **rank** — ранг процесса в коммуникаторе.

Как правило, вызов функций `MPI_Comm_size` и `MPI_Comm_rank` выполняется сразу после `MPI_Init` для получения общего количества процессов и ранга текущего процесса:

```
#include "mpi.h"
int main(int argc, char *argv[]) {
 int ProcNum, ProcRank;
 <программный код без использования функций MPI>
 MPI_Init(&agrc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
   <программный код с использованием функций MPI>
 MPI_Finalize();
 <программный код без использования функций MPI>
 return 0;
}
```

Следует отметить:

- коммуникатор `MPI_COMM_WORLD`, как отмечалось ранее, создается по умолчанию и представляет все процессы выполняемой параллельной программы;
- ранг, получаемый при помощи функции `MPI_Comm_rank`, является рангом процесса, выполнившего вызов этой функции, т. е. переменная *ProcRank* примет различные значения у разных процессов.

5.2.1.3. Передача сообщений

Для *передачи сообщения* процесс-отправитель должен выполнить функцию:

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm),
```

где

— **buf** — адрес буфера памяти, в котором располагаются данные отправляемого сообщения;
— **count** — количество элементов данных в сообщении;
— **type** — тип элементов данных пересылаемого сообщения;
— **dest** — ранг процесса, которому отправляется сообщение;
— **tag** — значение-тег, используемое для идентификации сообщения;
— **comm** — коммуникатор, в рамках которого выполняется передача данных.

Для указания типа пересылаемых данных в MPI имеется ряд базовых типов, полный список которых приведен в табл. 5.1.

Таблица 5.1. Базовые (предопределенные) типы данных MPI для алгоритмического языка C

Тип данных MPI	Тип данных C
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Следует отметить:

• отправляемое сообщение определяется через указание блока памяти (буфера), в котором это сообщение располагается. Используемая для указания буфера триада

```
(buf, count, type)
```

входит в состав параметров практически всех функций передачи данных;

- процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции MPI_Send;
- параметр *tag* используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное положительное целое число[1] (см. также описание функции MPI_Recv).

Сразу же после завершения функции MPI_Send процесс-отправитель может начать повторно использовать буфер памяти, в котором располагалось отправляемое сообщение. Также следует понимать, что в момент завершения функции MPI_Send состояние самого пересылаемого сообщения может быть совершенно различным: сообщение может располагаться в процессе-отправителе, может находиться в состоянии передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции MPI_Recv. Тем самым, завершение функции MPI_Send означает лишь, что операция передачи начала выполняться и пересылка сообщения рано или поздно будет выполнена.

Пример использования функции будет представлен после описания функции MPI_Recv.

5.2.1.4. Прием сообщений

Для *приема сообщения* процесс-получатель должен выполнить функцию:

```
int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
 int tag, MPI_Comm comm, MPI_Status *status),
```

где

— **buf, count, type** — буфер памяти для приема сообщения, назначение каждого отдельного параметра соответствует описанию в MPI_Send;
— **source** — ранг процесса, от которого должен быть выполнен прием сообщения;
— **tag** — тег сообщения, которое должно быть принято для процесса;
— **comm** — коммуникатор, в рамках которого выполняется передача данных;
— **status** – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Следует отметить:

[1] Максимально возможное целое число, однако, не может быть больше, чем определяемая реализацией константа MPI_TAG_UB.

- буфер памяти должен быть достаточным для приема сообщения. При нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения; с другой стороны, принимаемое сообщение может быть и короче, чем размер приемного буфера, в таком случае изменятся только участки буфера, затронутые принятым сообщением;
- типы элементов передаваемого и принимаемого сообщения должны совпадать;
- при необходимости приема сообщения от любого процесса-отправителя для параметра *source* может быть указано значение MPI_ANY_SOURCE (в отличие от функции передачи MPI_Send, которая отсылает сообщение строго определенному адресату);
- при необходимости приема сообщения с любым тегом для параметра *tag* может быть указано значение MPI_ANY_TAG (опять-таки, при использовании функции MPI_Send должно быть указано конкретное значение тега);
- в отличие от параметров «процесс-получатель» и «тег», параметр «коммуникатор» не имеет значения, означающего «любой коммуникатор»;
- параметр *status* позволяет определить ряд характеристик принятого сообщения:

— status.MPI_SOURCE — ранг процесса – отправителя принятого сообщения;
— status.MPI_TAG — тег принятого сообщения.

Приведенные значения MPI_ANY_SOURCE и MPI_ANY_TAG иногда называют *джокерами*.

Значение переменной *status* позволяет определить количество элементов данных в принятом сообщении при помощи функции:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype type,
  int *count),
```

где

— **status** — статус операции MPI_Recv;
— **type** — тип принятых данных;
— **count** — количество элементов данных в сообщении.

Вызов функции MPI_Recv не обязан быть согласованным со временем вызова соответствующей функции передачи сообщения MPI_Send – прием

сообщения может быть инициирован до момента, в момент или после момента начала отправки сообщения.

По завершении функции MPI_Recv в заданном буфере памяти будет располагаться принятое сообщение. Принципиальный момент здесь состоит в том, что функция MPI_Recv является *блокирующей* для процесса-получателя, т. е. его выполнение приостанавливается до завершения работы функции. Таким образом, если по каким-то причинам ожидаемое для приема сообщение будет отсутствовать, выполнение параллельной программы будет блокировано.

5.2.1.5. Первая параллельная программа с использованием MPI

Рассмотренный набор функций оказывается достаточным для разработки параллельных программ[1]. Приводимая ниже программа является стандартным начальным примером для алгоритмического языка С.

Программа 5.1. Первая параллельная программа с использованием MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
  int ProcNum, ProcRank, RecvRank;
  MPI_Status Status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if ( ProcRank == 0 ){
  // Действия, выполняемые только процессом с рангом 0
    printf("\n Hello from process %3d", ProcRank);
    for (int i = 1; i < ProcNum; i++ ) {
      MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
      printf("\n Hello from process %3d", RecvRank);
    }
  }
  else // Сообщение, отправляемое всеми процессами,
       // кроме процесса с рангом 0
```

[1] Как было обещано ранее, количество функций MPI, необходимых для начала разработки параллельных программ, оказалось равным шести.

```
    MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

Как следует из текста программы, каждый процесс определяет свой ранг, после чего действия в программе разделяются. Все процессы, кроме процесса с рангом 0, передают значение своего ранга нулевому процессу. Процесс с рангом 0 сначала печатает значение своего ранга, а далее последовательно принимает сообщения с рангами процессов и также печатает их значения. При этом важно отметить, что порядок приема сообщений заранее не определен и зависит от условий выполнения параллельной программы (более того, этот порядок может изменяться от запуска к запуску). Так, возможный вариант результатов печати процесса 0 может состоять в следующем (для параллельной программы из четырех процессов):

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

Такой «плавающий» вид получаемых результатов существенным образом усложняет разработку, тестирование и отладку параллельных программ, т. к. в этом случае исчезает один из основных принципов программирования – повторяемость выполняемых вычислительных экспериментов. Как правило, если это не приводит к потере эффективности, следует обеспечивать однозначность расчетов и при использовании параллельных вычислений. Для рассматриваемого простого примера можно восстановить постоянство получаемых результатов при помощи задания ранга процесса-отправителя в операции приема сообщения:

```
MPI_Recv(&RecvRank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD,
  &Status).
```

Указание ранга процесса-отправителя регламентирует порядок приема сообщений, и, как результат, строки печати будут появляться строго в порядке возрастания рангов процессов (повторим, что такая регламентация в отдельных ситуациях может приводить к замедлению выполняемых параллельных вычислений).

Следует отметить еще один важный момент: разрабатываемая с применением MPI программа, как в данном частном варианте, так и в самом

общем случае, используется для порождения всех процессов параллельной программы а значит, должна определять вычисления, выполняемые всеми этими процессами. Можно сказать, что MPI-программа является некоторой *«макропрограммой»*, различные части которой используются разными процессами. Так, например, в приведенном примере программы выделенные рамкой участки программного кода не выполняются одновременно ни одним из процессов. Первый выделенный участок с функцией приема `MPI_Recv` исполняется только процессом с рангом 0, второй участок с функцией передачи `MPI_Send` задействуется всеми процессами, за исключением нулевого процесса.

Для разделения фрагментов кода между процессами обычно используется подход, примененный в только что рассмотренной программе, – при помощи функции `MPI_Comm_rank` определяется ранг процесса, а затем в соответствии с рангом выделяются необходимые для процесса участки программного кода. Наличие в одной и той же программе фрагментов кода разных процессов также значительно усложняет понимание и, в целом, разработку MPI-программы. Как результат, можно рекомендовать при увеличении объема разрабатываемых программ выносить программный код разных процессов в отдельные программные модули (функции). Общая схема MPI-программы в этом случае будет иметь вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

Во многих случаях, как и в рассмотренном примере, выполняемые действия являются отличающимися только для процесса с рангом 0. В этом случае общая схема MPI-программы принимает более простой вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoManagerProcess();
else DoWorkerProcesses();
```

В завершение обсуждения примера поясним примененный в MPI подход для контроля правильности выполнения функций. Все функции MPI (кроме `MPI_Wtime` и `MPI_Wtick`) возвращают в качестве своего значения *код завершения*. При успешном выполнении функции возвращаемый код равен `MPI_SUCCESS`. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций. Для выяснения типа обнаруженной ошибки используются предопределенные именованные константы, среди которых:

— MPI_ERR_BUFFER — неправильный указатель на буфер;
— MPI_ERR_TRUNCATE — сообщение превышает размер приемного буфера;
— MPI_ERR_COMM — неправильный коммуникатор;
— MPI_ERR_RANK — неправильный ранг процесса и др.

Полный список констант для проверки кода завершения содержится в файле mpi.h. Однако, по умолчанию, возникновение любой ошибки во время выполнения функции MPI приводит к немедленному завершению параллельной программы. Для того чтобы иметь возможность проанализировать возвращаемый код завершения, необходимо воспользоваться предоставляемыми MPI функциями по созданию обработчиков ошибок и управлению ими, рассмотрение которых не входит в материал данной лекции.

5.2.2. Определение времени выполнение MPI-программы

Практически сразу же после разработки первых параллельных программ возникает необходимость определения времени выполнения вычислений для оценки достигаемого ускорения процессов решения задач за счет использования параллелизма. Используемые обычно средства для измерения времени работы программ зависят, как правило, от аппаратной платформы, операционной системы, алгоритмического языка и т. п. Стандарт MPI включает определение специальных функций для измерения времени, применение которых позволяет устранить зависимость от среды выполнения параллельных программ.

Получение *текущего момента времени* обеспечивается при помощи функции:

```
double MPI_Wtime(void),
```

результат ее вызова есть количество секунд, прошедшее от некоторого определенного момента времени в прошлом. Этот момент времени в прошлом, от которого происходит отсчет секунд, может зависеть от среды реализации библиотеки MPI, и, тем самым, для ухода от такой зависимости функцию MPI_Wtime следует использовать только для определения длительности выполнения тех или иных фрагментов кода параллельных программ. Возможная схема применения функции MPI_Wtime может состоять в следующем:

```
double t1, t2, dt;
t1 = MPI_Wtime();
…
```

```
t2 = MPI_Wtime();
dt = t2 - t1;
```

Точность измерения времени также может зависеть от среды выполнения параллельной программы. Для определения текущего значения точности может быть использована функция:

```
double MPI_Wtick(void),
```

позволяющая определить время в секундах между двумя последовательными показателями времени аппаратного таймера примененной компьютерной системы.

5.2.3. Начальное знакомство с коллективными операциями передачи данных

Функции `MPI_Send` и `MPI_Recv`, рассмотренные в п. 5.2.1, обеспечивают возможность выполнения *парных операций* передачи данных между двумя процессами параллельной программы. Для выполнения коммуникационных *коллективных операций*, в которых принимают участие все процессы коммуникатора, в MPI предусмотрен специальный набор функций. В данном подразделе будут рассмотрены три такие функции, широко применяемые даже при разработке сравнительно простых параллельных программ; полное же представление коллективных операций будет дано в подразделе 5.4.

Для демонстрации применения рассматриваемых функций MPI будет использоваться учебная *задача суммирования* элементов вектора x (см. подраздел 2.5):

$$S = \sum_{i=1}^{n} x_i \ .$$

Разработка параллельного алгоритма для решения данной задачи не вызывает затруднений: необходимо разделить данные на равные блоки, передать эти блоки процессам, выполнить в процессах суммирование полученных данных, собрать значения вычисленных частных сумм на одном из процессов и сложить значения частичных сумм для получения общего результата решаемой задачи. При последующей разработке демонстрационных программ данный рассмотренный алгоритм будет несколько упрощен: процессам программы будет передаваться весь суммируемый вектор, а не отдельные блоки этого вектора.

5.2.3.1. Передача данных от одного процесса всем процессам программы

Первая задача при выполнении рассмотренного параллельного алгоритма суммирования состоит в необходимости передачи значений вектора x всем процессам параллельной программы. Конечно, для решения этой задачи можно воспользоваться рассмотренными ранее функциями парных операций передачи данных:

```
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
for (int i = 1; i < ProcNum; i++)
 MPI_Send(&x, n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

Однако такое решение будет крайне неэффективным, поскольку повторение операций передачи приводит к суммированию затрат (латентностей) на подготовку передаваемых сообщений. Кроме того, как показано в лекции 3, данная операция может быть выполнена за $\log_2 p$ итераций передачи данных.

Достижение эффективного выполнения операции передачи данных от одного процесса всем процессам программы (*широковещательная рассылка данных*) может быть обеспечено при помощи функции MPI:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root,
 MPI_Comm comm),
```

где

- **buf, count, type** — буфер памяти с отправляемым сообщением (для процесса с рангом 0) и для приема сообщений (для всех остальных процессов);
- **root** — ранг процесса, выполняющего рассылку данных;
- **comm** — коммуникатор, в рамках которого выполняется передача данных.

Функция `MPI_Bcast` осуществляет рассылку данных из буфера *buf*, содержащего *count* элементов типа *type*, с процесса, имеющего номер *root*, всем процессам, входящим в коммуникатор *comm* (см. рис. 5.1).

Следует отметить:

- функция `MPI_Bcast` определяет коллективную операцию, и, тем самым, при выполнении необходимых рассылок данных вызов функции `MPI_Bcast` должен быть осуществлен всеми процессами указываемого коммуникатора (см. далее пример программы);
- указываемый в функции `MPI_Bcast` буфер памяти имеет различное назначение у разных процессов: для процесса с рангом *root*, кото-

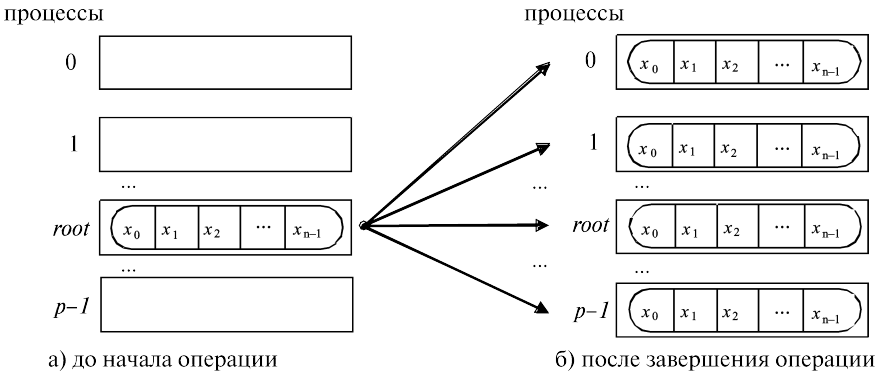


Рис. 5.1. Общая схема операции передачи данных от одного процесса всем процессам

рым осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение, а для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных;

- все коллективные операции «несовместимы» с парными операциями — так, например, принять широковещательное сообщение, отосланное с помощью `MPI_Bcast`, функцией `MPI_Recv` нельзя, для этого можно задействовать только `MPI_Bcast`.

Приведем программу для решения учебной задачи суммирования элементов вектора с использованием рассмотренной функции.

Программа 5.2. Параллельная программа суммирования числовых значений

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[]){
 double x[100], TotalSum, ProcSum = 0.0;
 int  ProcRank, ProcNum, N=100, k, i1, i2;
 MPI_Status Status;

 // Инициализация
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
```

```
// Подготовка данных
if ( ProcRank == 0 ) DataInitialization(x,N);

// Рассылка данных на все процессы
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Вычисление частичной суммы на каждом из процессов
// на каждом процессе суммируются элементы вектора x от i1 до i2
k = N / ProcNum;
i1 = k *   ProcRank;
i2 = k * ( ProcRank + 1 );
if ( ProcRank == ProcNum-1 ) i2 = N;
for ( int i = i1; i < i2; i++ )
  ProcSum  = ProcSum + x[i];

// Сборка частичных сумм на процессе с рангом 0
if ( ProcRank == 0 ) {
  TotalSum = ProcSum;
  for ( int i=1; i < ProcNum; i++ ) {
    MPI_Recv(&ProcSum,1,MPI_DOUBLE,MPI_ANY_SOURCE,0,
      MPI_COMM_WORLD, &Status);
    TotalSum = TotalSum + ProcSum;
  }
}
else // Все процессы отсылают свои частичные суммы
  MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

// Вывод результата
if ( ProcRank == 0 )
  printf("\nTotal Sum = %10.2f",TotalSum);
MPI_Finalize();
return 0;
}
```

В приведенной программе функция `DataInitialization` осуществляет подготовку начальных данных. Необходимые данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел — подготовка этой функции предоставляется как задание для самостоятельной разработки.

5.2.3.2. Передача данных от всех процессов одному процессу. Операция редукции

В рассмотренной программе суммирования числовых значений имеющаяся процедура сбора и последующего суммирования данных являет-

ся примером часто выполняемой коллективной операции передачи данных от всех процессов одному процессу. В этой операции над собираемыми значениями осуществляется та или иная обработка данных (для подчеркивания последнего момента данная операция еще именуется *операцией редукции данных*). Как и ранее, реализация операции редукции при помощи обычных парных операций передачи данных является неэффективной и достаточно трудоемкой. Для наилучшего выполнения действий, связанных с редукцией данных, в MPI предусмотрена функция:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm),
```

где

— **sendbuf** — буфер памяти с отправляемым сообщением;
— **recvbuf** — буфер памяти для результирующего сообщения (только для процесса с рангом root);
— **count** — количество элементов в сообщениях;
— **type** — тип элементов сообщений;
— **op** — операция, которая должна быть выполнена над данными;
— **root** — ранг процесса, на котором должен быть получен результат;
— **comm** — коммуникатор, в рамках которого выполняется операция.

В качестве операций редукции данных могут быть использованы предопределенные в MPI операции – см. табл. 5.2.

Помимо данного стандартного набора операций могут быть определены и новые дополнительные операции непосредственно самим пользователем библиотеки MPI – см., например, [4, 40 – 42, 57].

Общая схема выполнения операции сбора и обработки данных на одном процессе показана на рис. 5.2. Элементы получаемого сообщения на процессе *root* представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, т. е.:

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, 0 \le j < n \ ,$$

где $\otimes$ есть операция, задаваемая при вызове функции `MPI_Reduce` (для пояснения на рис. 5.3 показан пример выполнения функции редукции данных).

Следует отметить:

- функция `MPI_Reduce` определяет коллективную операцию, и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора. При этом все вызовы функции должны содержать одинаковые значения параметров *count, type, op, root, comm*;

Таблица 5.2. Базовые (предопределенные) типы операций MPI для функций редукции данных

Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции «И» над значениями сообщений
MPI_BAND	Выполнение битовой операции «И» над значениями сообщений
MPI_LOR	Выполнение логической операции «ИЛИ» над значениями сообщений
MPI_BOR	Выполнение битовой операции «ИЛИ» над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего «ИЛИ» над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего «ИЛИ» над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

- передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом *root*;

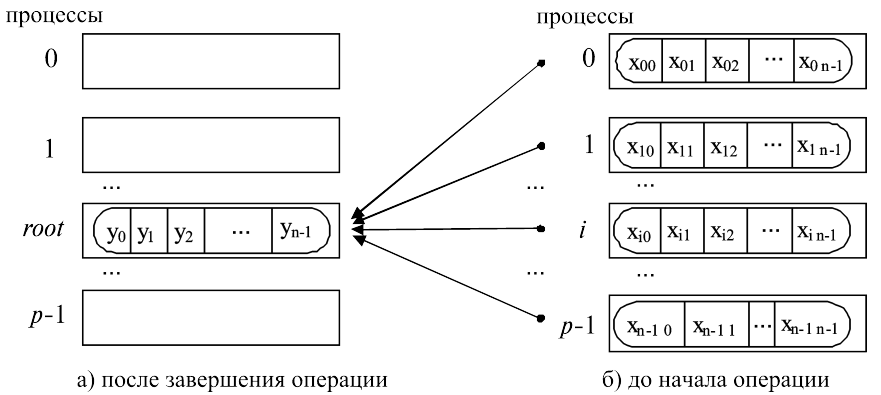
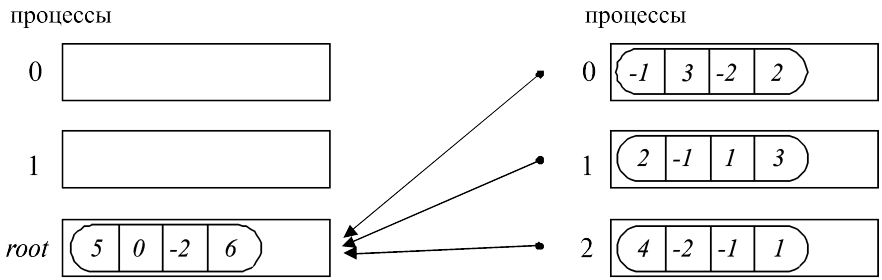


Рис. 5.2. Общая схема операции сбора и обработки на одном процессе данных от всех процессов

- выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования `MPI_SUM`, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно;
- не все сочетания типа данных *type* и операции *op* возможны, разрешенные сочетания перечислены в табл. 5.3.

Таблица 5.3. Разрешенные сочетания операции типа операнда в операции редукции

Операция	**Допустимый тип операндов для алгоритмического языка С**
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD	Целый, вещественный
MPI_LAND, MPI_LOR, MPI_LXOR	Целый
MPI_BAND, MPI_BOR, MPI_BXOR	Целый, байтовый
MPI_MINLOC, MPI_MAXLOC	Целый, вещественный



а) после завершения операции б) до начала операции

Рис. 5.3. Пример выполнения операции редукции при суммировании пересылаемых данных для трех процессов (в каждом сообщении 4 элемента, сообщения собираются на процессе с рангом 2)

Применим полученные знания для переработки ранее рассмотренной программы суммирования: как можно увидеть, весь программный код, выделенный рамкой, может быть теперь заменен на вызов одной лишь функции `MPI_Reduce`:

```
// Сборка частичных сумм на процессе с рангом 0
MPI_Reduce(&ProcSum, &TotalSum, 1, MPI_DOUBLE, MPI_SUM, 0,
 MPI_COMM_WORLD);
```

5.2.3.3. Синхронизация вычислений

В ряде ситуаций независимо выполняемые в процессах вычисления необходимо синхронизировать. Так, например, для измерения времени начала работы параллельной программы необходимо, чтобы для всех процессов одновременно были завершены все подготовительные действия, перед окончанием работы программы все процессы должны завершить свои вычисления и т. п.

Синхронизация процессов, т. е. одновременное достижение процессами тех или иных точек процесса вычислений, обеспечивается при помощи функции MPI:

```
int MPI_Barrier(MPI_Comm comm),
```

где

— **comm** — коммуникатор, в рамках которого выполняется операция.

Функция `MPI_Barrier` определяет коллективную операцию, и, тем самым, при использовании она должна вызываться всеми процессами используемого коммуникатора. При вызове функции `MPI_Barrier` выполнение процесса блокируется, продолжение вычислений процесса произойдет только после вызова функции `MPI_Barrier` всеми процессами коммуникатора.

5.2.3.4. Аварийное завершение параллельной программы

Для корректного завершения параллельной программы в случае непредвиденных ситуаций необходимо использовать функцию:

```
int MPI_Abort(MPI_Comm comm, int errorcode),
```

где

— **comm** — коммуникатор, процессы которого необходимо аварийно остановить;
— **errorcode** — код возврата из параллельной программы.

Эта функция корректно прерывает выполнение параллельной программы, оповещая об этом событии среду MPI, в отличие от функций

стандартной библиотеки алгоритмического языка С, таких, как *abort* или *terminate*. Обычное ее использование заключается в следующем:

```
MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
```

5.3. Операции передачи данных между двумя процессами

Продолжим начатое в п. 5.2.1 изучение функций MPI для выполнения операций передачи данных между процессами параллельной программы.

5.3.1. Режимы передачи данных

Рассмотренная ранее функция `MPI_Send` обеспечивает так называемый *стандартный* (*standard*) режим отправки сообщений, при котором (см. также п. 5.2.1.3):

- на время выполнения функции процесс – отправитель сообщения блокируется;
- после завершения функции буфер может быть использован повторно;
- состояние отправленного сообщения может быть различным – сообщение может располагаться на процессе-отправителе, может находиться в состоянии передачи, может храниться на процессе-получателе или же может быть принято процессом-получателем при помощи функции `MPI_Recv`.

Кроме стандартного режима в MPI предусматриваются следующие дополнительные режимы передачи сообщений:

- *синхронный* (*synchronous*) режим состоит в том, что завершение функции отправки сообщения происходит только при получении от процесса-получателя подтверждения о начале приема отправленного сообщения. Отправленное сообщение или полностью принято процессом-получателем, или находится в состоянии приема;
- *буферизованный* (*buffered*) режим предполагает использование дополнительных системных или задаваемых пользователем буферов для копирования в них отправляемых сообщений. Функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер;
- *режим передачи по готовности* (*ready*) может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

Для именования функций отправки сообщения для разных режимов выполнения в MPI применяется название функции `MPI_Send`, к которому как префикс добавляется начальный символ названия соответствующего режима работы, т. е.:

— **MPI_Ssend** – функция отправки сообщения в синхронном режиме;
— **MPI_Bsend** – функция отправки сообщения в буферизованном режиме;
— **MPI_Rsend** – функция отправки сообщения в режиме по готовности.

Список параметров всех перечисленных функций совпадает с составом параметров функции `MPI_Send`.

Для применения буферизованного режима передачи может быть создан и передан MPI буфер памяти, используемая для этого функция имеет вид:

```
int MPI_Buffer_attach(void *buf, int size),
```

где

— **buf** — адрес буфера памяти;
— **size** — размер буфера.

После завершения работы с буфером он должен быть отключен от MPI при помощи функции:

```
int MPI_Buffer_detach(void *buf, int *size),
```

где

— **buf** — адрес буфера памяти;
— **size** — возвращаемый размер буфера.

По практическому использованию режимов можно привести следующие рекомендации:

- стандартный режим обычно реализуется как буферизированный или синхронный, в зависимости от размера передаваемого сообщения, и зачастую является наиболее оптимизированным по производительности;
- режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т. к. обычно сложно гарантировать готовность операции приема;
- буферизованный режим также выполняется достаточно быстро, но может приводить к большим расходам ресурсов (памяти), – в целом может быть рекомендован для передачи коротких сообщений;

- синхронный режим является наиболее медленным, т.к. требует подтверждения приема, однако не нуждается в дополнительной памяти для хранения сообщения. Этот режим может быть рекомендован для передачи длинных сообщений.

В заключение отметим, что для функции приема `MPI_Recv` не существует различных режимов работы.

5.3.2. Организация неблокирующих обменов данными между процессами

Все рассмотренные ранее функции отправки и приема сообщений являются *блокирующими*, т. е. приостанавливающими выполнение процессов до момента завершения работы вызванных функций. В то же время при выполнении параллельных вычислений часть сообщений может быть отправлена и принята заранее, до момента реальной потребности в пересылаемых данных. В таких ситуациях было бы крайне желательным иметь возможность выполнения функций обмена данными без блокировки процессов для совмещения процессов передачи сообщений и вычислений. Такой неблокирующий способ выполнения обменов является, конечно, более сложным для использования, но при правильном применении может в значительной степени уменьшить потери эффективности параллельных вычислений из-за медленных (по сравнению с быстродействием процессоров) коммуникационных операций.

MPI обеспечивает возможность неблокированного выполнения операций передачи данных между двумя процессами. Наименование неблокирующих аналогов образуется из названий соответствующих функций путем добавления префикса **I** (*Immediate*). Список параметров неблокирующих функций содержит обычный набор параметров исходных функций и один дополнительный параметр *request* с типом `MPI_Request` (в функции `MPI_Irecv` отсутствует также параметр *status*):

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm, MPI_Request *request),
int MPI_Issend(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm, MPI_Request *request),
int MPI_Ibsend(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm, MPI_Request *request),
int MPI_Irsend(void *buf, int count, MPI_Datatype type, int dest,
 int tag, MPI_Comm comm, MPI_Request *request),
int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source,
 int tag, MPI_Comm comm, MPI_Request *request).
```

Вызов неблокирующей функции приводит к инициации запрошенной операции передачи, после чего выполнение функции завершается и процесс может продолжить свои действия. Перед своим завершением неблокирующая функция определяет переменную *request*, которая далее может использоваться для проверки завершения инициированной операции обмена.

Проверка состояния выполняемой неблокирующей операции передачи данных производится при помощи функции:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_status *status),
```

где

- **request** — дескриптор операции, определенный при вызове неблокирующей функции;
- **flag** — результат проверки (true, если операция завершена);
- **status** — результат выполнения операции обмена (только для завершенной операции).

Операция проверки является неблокирующей, т. е. процесс может проверить состояние неблокирующей операции обмена и продолжить далее свои вычисления, если по результатам проверки окажется, что операция все еще не завершена. Возможная схема совмещения вычислений и выполнения неблокирующей операции обмена может состоять в следующем:

```
MPI_Isend(buf, count, type, dest, tag, comm, &request);
…
do {
 …
 MPI_Test(&request, &flag, &status);
} while (!flag);
```

Если при выполнении неблокирующей операции окажется, что продолжение вычислений невозможно без получения передаваемых данных, то может быть использована блокирующая операция ожидания завершения операции:

```
int MPI_Wait(MPI_Request *request, MPI_status *status),
```

где

- **request** — дескриптор операции, определенный при вызове неблокирующей функции;
- **status** — результат выполнения операции обмена (только для завершенной операции).

Кроме рассмотренных, MPI содержит ряд дополнительных функций проверки и ожидания неблокирующих операций обмена:

— **MPI_Testall** — проверка завершения всех перечисленных операций обмена;
— **MPI_Waitall** — ожидание завершения всех операций обмена;
— **MPI_Testany** — проверка завершения хотя бы одной из перечисленных операций обмена;
— **MPI_Waitany** — ожидание завершения любой из перечисленных операций обмена;
— **MPI_Testsome** — проверка завершения каждой из перечисленных операций обмена;
— **MPI_Waitsome** — ожидание завершения хотя бы одной из перечисленных операций обмена и оценка состояния по всем операциям.

Приведение простого примера использования неблокирующих функций достаточно затруднительно. Хорошей возможностью для освоения рассмотренных функций могут служить, например, параллельные алгоритмы матричного умножения (см. лекцию 7).

5.3.3. Одновременное выполнение передачи и приема

Одной из часто выполняемых форм информационного взаимодействия в параллельных программах является обмен данными между процессами, когда для продолжения вычислений процессам необходимо отправить данные одним процессам и в то же время получить сообщения от других. Простейший вариант этой ситуации состоит, например, в обмене данными между двумя процессами. Реализация таких обменов при помощи обычных парных операций передачи данных может быть неэффективна, кроме того, такая реализация должна гарантировать отсутствие тупиковых ситуаций, которые могут возникать, например, когда два процесса начинают передавать сообщения друг другу с использованием блокирующих функций передачи данных.

Достижение эффективного и гарантированного одновременного выполнения операций передачи и приема данных может быть обеспечено при помощи функции MPI:

```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype stype,
  int dest, int stag, void *rbuf,int rcount,MPI_Datatype rtype,
  int source,int rtag, MPI_Comm comm, MPI_Status *status),
```

где

- **sbuf, scount, stype, dest, stag** — параметры передаваемого сообщения;
- **rbuf, rcount, rtype, source, rtag** — параметры принимаемого сообщения;
- **comm** — коммуникатор, в рамках которого выполняется передача данных;
- **status** — структура данных с информацией о результате выполнения операции.

Как следует из описания, функция `MPI_Sendrecv` передает сообщение, описываемое параметрами (*sbuf, scount, stype, dest, stag*), процессу с рангом *dest* и принимает сообщение в буфер, определяемый параметрами (*rbuf, rcount, rtype, source, rtag*), от процесса с рангом *source*.

В функции `MPI_Sendrecv` для передачи и приема сообщений применяются разные буферы. В случае же когда отсылаемое сообщение больше не нужно на процессе-отправителе, в MPI имеется возможность использования единого буфера:

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype type,
  int dest, int stag, int source, int rtag, MPI_Comm comm,
  MPI_Status* status),
```

где

- **buf, count, type** — параметры передаваемого сообщения;
- **dest** — ранг процесса, которому отправляется сообщение;
- **stag** — тег для идентификации отправляемого сообщения;
- **source** — ранг процесса, от которого выполняется прием сообщения;
- **rtag** — тег для идентификации принимаемого сообщения;
- **comm** — коммуникатор, в рамках которого выполняется передача данных;
- **status** — структура данных с информацией о результате выполнения операции.

Пример использования функций для одновременного выполнения операций передачи и приема приведен в лекции 7 при разработке параллельных программ матричного умножения.

5.4. Коллективные операции передачи данных

Как уже отмечалось ранее, под *коллективными операциями* в MPI понимаются операции над данными, в которых принимают участие все про-

цессы используемого коммуникатора. Выделение основных видов коллективных операций было выполнено в лекции 3. Часть коллективных операций уже была рассмотрена в п. 5.2.3 – это операции передачи от одного процесса всем процессам коммуникатора (*широковещательная рассылка*) и операции обработки данных, полученных на одном процессе от всех процессов (*редукция данных*).

Рассмотрим далее оставшиеся базовые коллективные операции передачи данных.

5.4.1. Обобщенная передача данных от одного процесса всем процессам

Обобщенная операция передачи данных от одного процесса всем процессам (*распределение данных*) отличается от широковещательной рассылки тем, что процесс передает процессам различающиеся данные (см. рис. 5.4). Выполнение данной операции может быть обеспечено при помощи функции:

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,
 void *rbuf, int rcount, MPI_Datatype rtype, int root,
 MPI_Comm comm),
```

где

— **sbuf, scount, stype** — параметры передаваемого сообщения (*scount* определяет количество элементов, передаваемых на каждый процесс);

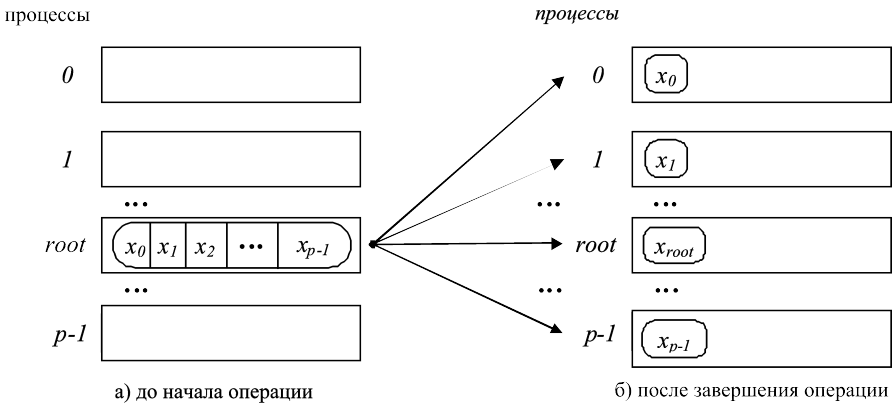


Рис. 5.4. Общая схема операции обобщенной передачи данных от одного процесса всем процессам

— **rbuf, rcount, rtype** — параметры сообщения, принимаемого в процессах;
— **root** — ранг процесса, выполняющего рассылку данных;
— **comm** — коммуникатор, в рамках которого выполняется передача данных.

При вызове этой функции процесс с рангом *root* произведет передачу данных всем другим процессам в коммуникаторе. Каждому процессу будет отправлено *scount* элементов. Процесс с рангом 0 получит блок данных из *sbuf* элементов с индексами от 0 до *scount*-1, процессу с рангом 1 будет отправлен блок из *sbuf* элементов с индексами от *scount* до 2**scount*-1 и т. д. Тем самым, общий размер отправляемого сообщения должен быть равен *scount* * *p* элементов, где *p* есть количество процессов в коммуникаторе comm.

Следует отметить, что поскольку функция `MPI_Scatter` определяет коллективную операцию, вызов этой функции при выполнении рассылки данных должен быть обеспечен на каждом процессе коммуникатора.

Отметим также, что функция `MPI_Scatter` передает всем процессам сообщения одинакового размера. Выполнение более общего варианта операции распределения данных, когда размеры сообщений для процессов могут быть различны, обеспечивается при помощи функции `MPI_Scatterv`.

Пример использования функции `MPI_Scatter` рассматривается в лекции 6 при разработке параллельных программ умножения матрицы на вектор.

5.4.2. Обобщенная передача данных от всех процессов одному процессу

Операция обобщенной передачи данных от всех процессов одному процессу (*сбор данных*) является двойственной к процедуре распределения данных (см. рис. 5.5). Для выполнения этой операции в MPI предназначена функция:

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype,
  void *rbuf, int rcount, MPI_Datatype rtype,
  int root, MPI_Comm comm),
```

где

— **sbuf, scount, stype** — параметры передаваемого сообщения;
— **rbuf, rcount, rtype** — параметры принимаемого сообщения;
— **root** — ранг процесса, выполняющего сбор данных;

— **comm** — коммуникатор, в рамках которого выполняется передача данных.

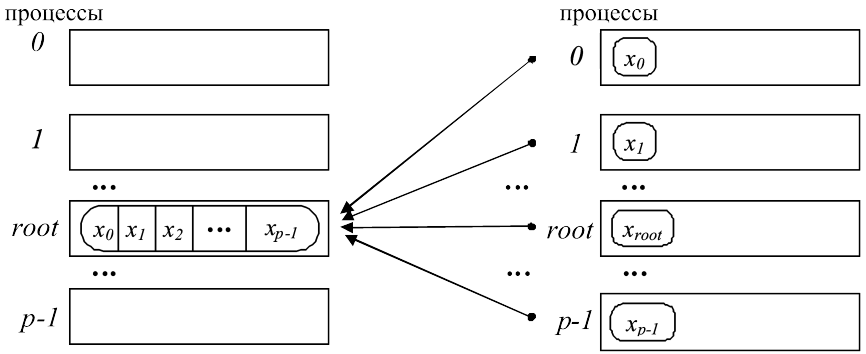


Рис. 5.5. Общая схема операции обобщенной передачи данных от всех процессов одному процессу

При выполнении функции `MPI_Gather` каждый процесс в коммуникаторе передает данные из буфера *sbuf* на процесс с рангом *root*. Процесс с рангом *root* собирает все получаемые данные в буфере *rbuf* (размещение данных в буфере осуществляется в соответствии с рангами процессов – отправителей сообщений). Для того чтобы разместить все поступающие данные, размер буфера *rbuf* должен быть равен *scount* * *p* элементов, где *p* есть количество процессов в коммуникаторе *comm*.

Функция `MPI_Gather` также определяет коллективную операцию, и ее вызов при выполнении сбора данных должен быть обеспечен в каждом процессе коммуникатора.

Следует отметить, что при использовании функции `MPI_Gather` сборка данных осуществляется только на одном процессе. Для получения всех собираемых данных на каждом из процессов коммуникатора необходимо применять функцию сбора и рассылки:

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,
 void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm),
```

где

— **sbuf, scount, stype** — параметры передаваемого сообщения;
— **rbuf, rcount, rtype** — параметры принимаемого сообщения;
— **comm** — коммуникатор, в рамках которого выполняется передача данных.

Выполнение общего варианта операции сбора данных, когда размеры передаваемых процессами сообщений могут быть различны, обеспечивается при помощи функций `MPI_Gatherv` и `MPI_Allgatherv`.

Пример использования функции `MPI_Gather` рассматривается в лекции 6 при разработке параллельных программ умножения матрицы на вектор.

5.4.3. Общая передача данных от всех процессов всем процессам

Передача данных от всех процессов всем процессам является наиболее общей операцией передачи данных (см. рис. 5.6). Выполнение данной операции может быть обеспечено при помощи функции:

```
int MPI_Alltoall(void *sbuf,int scount,MPI_Datatype stype,
 void *rbuf,int rcount,MPI_Datatype rtype,MPI_Comm comm),
```

где

- **sbuf, scount, stype** — параметры передаваемых сообщений;
- **rbuf, rcount, rtype** — параметры принимаемых сообщений;
- **comm** — коммуникатор, в рамках которого выполняется передача данных.

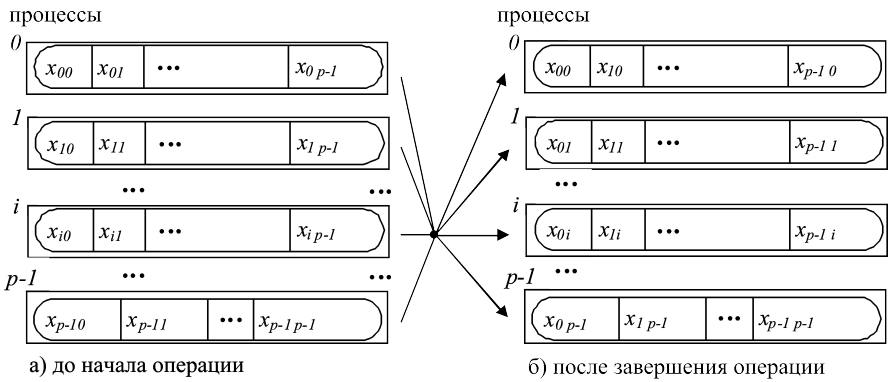


Рис. 5.6. Общая схема операции передачи данных от всех процессов всем процессам

При выполнении функции `MPI_Alltoall` каждый процесс в коммуникаторе передает данные из *scount* элементов каждому процессу (общий размер отправляемых сообщений в процессах должен быть равен *scount* * *p* элементов, где *p* есть количество процессов в коммуникаторе *comm*) и принимает сообщения от каждого процесса.

Вызов функции MPI_Alltoall при выполнении операции общего обмена данными должен быть выполнен в каждом процессе коммуникатора.

Вариант операции общего обмена данных, когда размеры передаваемых процессами сообщений могут быть различны, обеспечивается при помощи функций MPI_Alltoallv.

Пример использования функции MPI_Alltoall рассматривается в лекции 6 при разработке параллельных программ умножения матрицы на вектор как задание для самостоятельного выполнения.

5.4.4. Дополнительные операции редукции данных

Рассмотренная в п. 5.2.3.2 функция MPI_Reduce обеспечивает получение результатов редукции данных только на одном процессе. Для получения результатов редукции данных на каждом из процессов коммуникатора необходимо использовать функцию редукции и рассылки:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, MPI_Comm comm),
```

где

— **sendbuf** — буфер памяти с отправляемым сообщением;
— **recvbuf** — буфер памяти для результирующего сообщения;
— **count** — количество элементов в сообщениях;
— **type** — тип элементов сообщений;
— **op** — операция, которая должна быть выполнена над данными;
— **comm** — коммуникатор, в рамках которого выполняется операция.

Функция MPI_Allreduce выполняет рассылку между процессами всех результатов операции редукции. Возможность управления распределением этих данных между процессами предоставляется функций MPI_Reduce_scatter.

И еще один вариант операции сбора и обработки данных, при котором обеспечивается получение всех частичных результатов редуцирования, может быть реализован при помощи функции:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, MPI_Comm comm),
```

где

— **sendbuf** — буфер памяти с отправляемым сообщением;
— **recvbuf** — буфер памяти для результирующего сообщения;
— **count** — количество элементов в сообщениях;

— **type** — тип элементов сообщений;
— **op** — операция, которая должна быть выполнена над данными;
— **comm** — коммуникатор, в рамках которого выполняется операция.

Общая схема выполнения функции `MPI_Scan` показана на рис. 5.7. Элементы получаемых сообщений представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, при этом для получения результатов на процессе с рангом i, $0 \le i < n$, используются данные от процессов, ранг которых меньше или равен i, т. е.

$$y_{ij} = \overset{i}{\underset{k=0}{\otimes}} x_{kj},\ 0 \le i, j < n,$$

где $\otimes$ есть операция, задаваемая при вызове функции `MPI_Scan`.

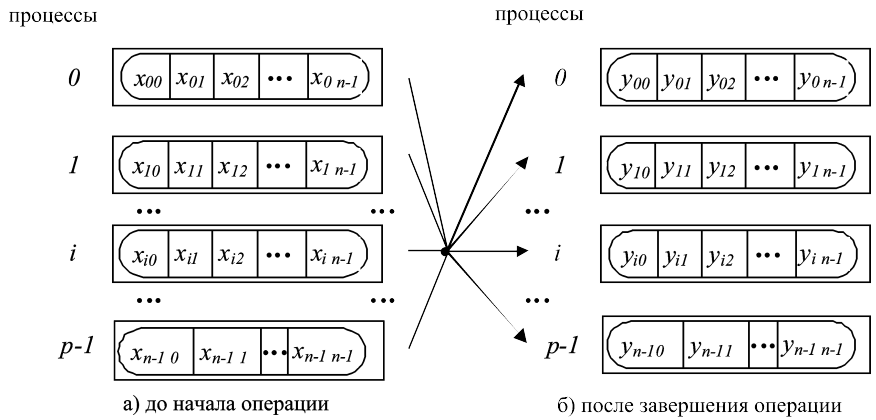


Рис. 5.7. Общая схема операции редукции с получением частичных результатов обработки данных

5.4.5. Сводный перечень коллективных операций данных

Для удобства использования сводный перечень всего рассмотренного учебного материала о коллективных операциях передачи данных представлен в виде табл. 5.4.

5.5. Производные типы данных в MPI

Во всех ранее рассмотренных примерах использования функций передачи данных предполагалось, что сообщения представляют собой некоторый непрерывный вектор элементов предусмотренного в MPI типа

Таблица 5.4. Сводный перечень учебного материала о коллективных операциях передачи данных

Вид коллективной операции	**Общее описание и оценка сложности**	**Функция MPI**	**Примеры использования**
Передача от одного процесса всем процессам (широковещательная рассылка)	п. 3.2.2	MPI_Bcast п. 5.2.3.1	п. 5.2.3.1
Сбор и обработка данных на одном процессе от всех процессов (редукция данных)	пп. 3.2.2, 3.2.3	MPI_Reduce п. 5.2.3.2	п. 5.2.3.2
– то же с рассылкой результатов всем процессам	пп. 3.2.2, 3.2.3	MPI_Allreduce MPI_Reduce_ scatter п. 5.4.4	
– то же с получением частичных результатов обработки	пп. 3.2.2, 3.2.3	MPI_Scan п. 5.4.4	
Обобщенная передача от одного процесса всем процессам (распределение данных)	п. 3.2.4	MPI_Scatter MPI_Scatterv п. 5.4.1	Лекция 6
Обобщенная передача от всех процессов одному процессу (сбор данных)	п. 3.2.4	MPI_Gather MPI_Gatherv п. 5.4.2	Лекция 6
– то же с рассылкой результатов всем процессам	п. 3.2.4	MPI_Allgather MPI_Allgatherv п. 5.4.2	
Обобщенная передача данных от всех процессов всем процессам	п. 3.2.5	MPI_Alltoall MPI_Alltoallv п. 5.4.3	Лекция 6

(список имеющихся в MPI типов представлен в табл. 5.1). Понятно, что в общем случае необходимые к пересылке данные могут рядом не располагаться и состоять из значений разных типов. Конечно, и в этих ситуациях разрозненные данные могут быть переданы с использованием нескольких сообщений, но такой способ решения неэффективен в силу накопления латентностей множества выполняемых операций передачи данных. Другой возможный подход состоит в предварительной упаковке передаваемых данных в формат того или иного непрерывного вектора, однако и здесь появляются лишние операции копирования данных, да и понятность таких операций передачи далека от желаемой.

Для обеспечения больших возможностей при определении состава передаваемых сообщений в MPI предусмотрен механизм так называемых производных типов данных. Далее будут даны основные понятия используемого подхода, приведены возможные способы конструирования производных типов данных и рассмотрены функции упаковки и распаковки данных.

5.5.1. Понятие производного типа данных

В самом общем виде под *производным типом данных* в MPI можно понимать описание набора значений предусмотренного в MPI типа, причем в общем случае описываемые значения не обязательно непрерывно располагаются в памяти. Задание типа в MPI принято осуществлять при помощи *карты типа* (*type map*) в виде последовательности описаний входящих в тип значений; каждое отдельное значение описывается указанием типа и смещения адреса месторасположения от некоторого базового адреса, т. е.

$$\textbf{\textit{TypeMap}} = \{(type_0, disp_0),...,(type_{n-1}, disp_{n-1})\}.$$

Часть карты типа с указанием только типов значений именуется в MPI *сигнатурой типа*:

$$\textbf{\textit{TypeSignature}} = \{type_{0,...,}type_{n-1}\}.$$

Сигнатура типа описывает, какие базовые типы данных образуют некоторый производный тип данных MPI, и, тем самым, управляет интерпретацией элементов данных при передаче или получении сообщений. Смещения карты типа определяют, где находятся значения данных.

Поясним рассмотренные понятия на следующем примере. Пусть в сообщение должны входить значения переменных:

```
double a; /* адрес 24 */
double b; /* адрес 40 */
int    n; /* адрес 48 */
```

Тогда производный тип для описания таких данных должен иметь карту типа следующего вида:

```
{(MPI_DOUBLE, 0),
(MPI_DOUBLE, 16),
(MPI_INT,    24)}
```

Дополнительно для производных типов данных в MPI используется следующий ряд новых понятий:

- нижняя граница типа
  $lb(\textbf{\textit{TypeMap}}) = \min_j(disp_j)$;
- верхняя граница типа
  $ub(\textbf{\textit{TypeMap}}) = \max_j(disp_j + sizeof(type_j)) + \Delta$;
- протяженность типа
  $extent(\textbf{\textit{TypeMap}}) = ub(\textbf{\textit{TypeMap}}) - lb(\textbf{\textit{TypeMap}})$.

Согласно определению, *нижняя граница* есть смещение для первого байта значений рассматриваемого типа данных. Соответственно, *верхняя граница* представляет собой смещение для байта, располагающегося вслед за последним элементом рассматриваемого типа данных. При этом величина смещения для верхней границы может быть округлена вверх с учетом требований выравнивания адресов. Так, одно из требований, которые налагают некоторые реализации языков C и Fortran, состоит в том, чтобы адрес элемента был кратен длине этого элемента в байтах. Например, если тип *int* занимает четыре байта, то адрес элемента типа *int* должен нацело делиться на четыре. Именно это требование и отражается в определении верхней границы типа данных MPI. Поясним данный момент на ранее рассмотренном примере набора переменных *a*, *b* и *n*, для которого нижняя граница равна 0, а верхняя принимает значение 32 (величина округления 6 или 4 в зависимости от размера типа *int*). Здесь следует отметить, что требуемое выравнивание определяется по типу первого элемента данных в карте типа.

Следует также указать на различие понятий «протяженность» и «размер типа». *Протяженность* – это размер памяти в байтах, который нужно отводить для одного элемента производного типа. *Размер типа данных* – это число байтов, которые занимают данные (разность между адресами последнего и первого байтов данных). Различие в значениях протяженности и размера опять же в величине округления для выравнивания адресов. Так, в рассматриваемом примере размер типа равен 28, а протяженность – 32 (предполагается, что тип *int* занимает четыре байта).

Для получения значения протяженности типа в MPI предусмотрена функция:

```
int MPI_Type_extent(MPI_Datatype type, MPI_Aint *extent),
```

где

— **type** — тип данных, протяженность которого отыскивается;
— **extent** — протяженность типа.

Размер типа можно найти, используя функцию:

```
int MPI_Type_size(MPI_Datatype type, MPI_Aint *size),
```

где
— **type** — тип данных, размер которого отыскивается;
— **size** — размер типа.

Определение нижней и верхней границ типа может быть выполнено при помощи функций:

```
int MPI_Type_lb(MPI_Datatype type, MPI_Aint *disp) и
int MPI_Type_ub(MPI_Datatype type, MPI_Aint *disp),
```

где
— **type** — тип данных, нижняя граница которого отыскивается;
— **disp** — нижняя/верхняя граница типа.

Важной и необходимой при конструировании производных типов является функция получения адреса переменной:

```
int MPI_Address(void *location, MPI_Aint *address),
```

где
— **location** — адрес памяти;
— **address** — адрес памяти в переносимом MPI-формате
(следует отметить, что данная функция является переносимым вариантом средств получения адресов в алгоритмических языках C и Fortran).

5.5.2. Способы конструирования производных типов данных

Для снижения сложности в MPI предусмотрено несколько различных способов конструирования производных типов:

- *непрерывный* способ позволяет определить непрерывный набор элементов существующего типа как новый производный тип;
- *векторный* способ обеспечивает создание нового производного типа как набора элементов существующего типа, между элементами которого имеются регулярные промежутки по памяти. При этом размер промежутков задается в числе элементов исходного типа, в

то время как в варианте *H-векторного* способа этот размер указывается в байтах;
- *индексный* способ отличается от векторного метода тем, что промежутки между элементами исходного типа могут иметь нерегулярный характер (имеется и *H-индексный* способ, отличающийся способом задания промежутков);
- *структурный* способ обеспечивает самое общее описание производного типа через явное указание карты создаваемого типа данных.

Далее перечисленные способы конструирования производных типов данных будут рассмотрены более подробно.

5.5.2.1. Непрерывный способ конструирования

При *непрерывном способе* конструирования производного типа данных в MPI используется функция:

```
int MPI_Type_contiguous(int count, MPI_Data_type oldtype,
  MPI_Datatype *newtype),
```

где

— **count** — количество элементов исходного типа;
— **oldtype** — исходный тип данных;
— **newtype** — новый определяемый тип данных.

Как следует из описания, новый тип *newtype* создается как *count* элементов исходного типа *oldtype*. Например, если исходный тип данных имеет карту типа

```
{(MPI_INT, 0), (MPI_DOUBLE, 8)},
```

то вызов функции `MPI_Type_contiguous` с параметрами

```
MPI_Type_contiguous(2, oldtype, &newtype);
```

приведет к созданию типа данных с картой типа

```
{(MPI_INT, 0), (MPI_DOUBLE, 8), (MPI_INT, 16), (MPI_DOUBLE, 24)}.
```

В определенном плане наличие непрерывного способа конструирования является избыточным, поскольку использование аргумента *count* в процедурах MPI равносильно использованию непрерывного типа данных такого же размера.

5.5.2.2. Векторный способ конструирования

При *векторном способе* конструирования производного типа данных в MPI применяются функции

```
int MPI_Type_vector(int count, int blocklen, int stride,
 MPI_Data_type oldtype, MPI_Datatype *newtype) и
int MPI_Type_hvector(int count, int blocklen, MPI_Aint stride,
 MPI_Data_type oldtype, MPI_Datatype *newtype),
```

где

— **count** — количество блоков;
— **blocklen** — размер каждого блока;
— **stride** — количество элементов, расположенных между двумя соседними блоками;
— **oldtype** — исходный тип данных;
— **newtype** — новый определяемый тип данных.

Отличие способа конструирования, определяемого функцией `MPI_Type_hvector`, состоит лишь в том, что параметр *stride* для определения интервала между блоками задается в байтах, а не в элементах исходного типа данных.

Как следует из описания, при векторном способе новый производный тип создается как набор блоков из элементов исходного типа, при этом между блоками могут иметься регулярные промежутки по памяти. Приведем несколько примеров использования данного способа конструирования типов:

- конструирование типа для выделения половины (только четных или только нечетных) строк матрицы размером $n \times n$:

```
MPI_Type_vector(n / 2, n, 2 * n, &StripRowType, &ElemType),
```

- конструирование типа для выделения столбца матрицы размером $n \times n$:

```
MPI_Type_vector(n, 1, n, &ColumnType, &ElemType),
```

- конструирование типа для выделения главной диагонали матрицы размером $n \times n$:

```
MPI_Type_vector(n, 1, n + 1, &DiagonalType, &ElemType).
```

С учетом характера приводимых примеров можно упомянуть имеющуюся в MPI возможность создания производных типов для описания подмассивов многомерных массивов при помощи функции (данная функция предусматривается стандартом MPI-2):

```
int MPI_Type_create_subarray(int ndims, int *sizes, int *subsizes,
 int *starts, int order, MPI_Data_type oldtype, MPI_Datatype *newtype),
```

где

— **ndims** — размерность массива;
— **sizes** — количество элементов в каждой размерности исходного массива;
— **subsizes** — количество элементов в каждой размерности определяемого подмассива;
— **starts** — индексы начальных элементов в каждой размерности определяемого подмассива;
— **order** — параметр для указания необходимости переупорядочения;
— **oldtype** — тип данных элементов исходного массива;
— **newtype** — новый тип данных для описания подмассива.

5.5.2.3. Индексный способ конструирования

При *индексном способе* конструирования производного типа данных в MPI используются функции:

```
int MPI_Type_indexed(int count, int blocklens[], int indices[],
 MPI_Data_type oldtype, MPI_Datatype *newtype) и
int MPI_Type_hindexed(int count, int blocklens[], MPI_Aint indices[],
 MPI_Data_type oldtype, MPI_Datatype *newtype),
```

где

— **count** — количество блоков;
— **blocklens** — количество элементов в каждом блоке;
— **indices** — смещение каждого блока от начала типа;
— **oldtype** — исходный тип данных;
— **newtype** — новый определяемый тип данных.

Как следует из описания, при индексном способе новый производный тип создается как набор блоков разного размера из элементов исходного типа, при этом между блоками могут иметься разные промежутки по памяти. Для пояснения данного способа можно привести пример конструирования типа для описания верхней треугольной матрицы размером $n \times n$:

```
// Конструирование типа для описания верхней треугольной матрицы
for ( i = 0, i < n; i++ ) {
 blocklens[i] = n - i;
 indices  [i] = i * n + i;
}
MPI_Type_indexed(n, blocklens, indices, &UTMatrixType, &ElemType).
```

Как и ранее, способ конструирования, определяемый функцией `MPI_Type_hindexed`, отличается тем, что элементы *indices* для определения интервалов между блоками задаются в байтах, а не в элементах исходного типа данных.

Следует отметить, что существует еще одна дополнительная функция `MPI_Type_create_indexed_block` индексного способа конструирования для определения типов с блоками одинакового размера (данная функция предусматривается стандартом MPI-2).

5.5.2.4. Структурный способ конструирования

Как отмечалось ранее, *структурный способ* является самым общим методом конструирования производного типа данных при явном задании соответствующей карты типа. Использование такого способа производится при помощи функции:

```
int MPI_Type_struct(int count, int blocklens[], MPI_Aint indices[],
 MPI_Data_type oldtypes[], MPI_Datatype *newtype),
```

где

— **count** — количество блоков;
— **blocklens** — количество элементов в каждом блоке;
— **indices** — смещение каждого блока от начала типа (в байтах);
— **oldtypes** — исходные типы данных в каждом блоке в отдельности;
— **newtype** — новый определяемый тип данных.

Как следует из описания, структурный способ дополнительно к индексному методу позволяет указывать типы элементов для каждого блока в отдельности.

5.5.3. Объявление производных типов и их удаление

Рассмотренные в предыдущем пункте функции конструирования позволяют определить производный тип данных. Дополнительно перед использованием созданный тип *должен быть объявлен* при помощи функции:

```
int MPI_Type_commit(MPI_Datatype *type),
```

где

— **type** – объявляемый тип данных.

При завершении использования производный тип *должен быть аннулирован* при помощи функции:

```
int MPI_Type_free(MPI_Datatype *type),
```

где

— **type** – аннулируемый тип данных.

5.5.4. Формирование сообщений при помощи упаковки и распаковки данных

Наряду с рассмотренными в п. 5.5.2 методами конструирования производных типов в MPI предусмотрен и явный способ сборки и разборки сообщений, содержащих значения разных типов и располагаемых в разных областях памяти.

Для использования данного подхода должен быть определен буфер памяти достаточного размера для сборки сообщения. Входящие в состав сообщения данные должны быть *упакованы* в буфер при помощи функции:

```
int MPI_Pack(void *data, int count, MPI_Datatype type, void *buf,
  int bufsize, int *bufpos, MPI_Comm comm),
```

где

— **data** — буфер памяти с элементами для упаковки;
— **count** — количество элементов в буфере;
— **type** — тип данных для упаковываемых элементов;
— **buf** — буфер памяти для упаковки;
— **bufsize** — размер буфера в байтах;
— **bufpos** — позиция для начала записи в буфер (в байтах от начала буфера);
— **comm** — коммуникатор для упакованного сообщения.

Функция `MPI_Pack` упаковывает *count* элементов из буфера *data* в буфер упаковки *buf*, начиная с позиции *bufpos*. Общая схема процедуры упаковки показана на рис. 5.8а.

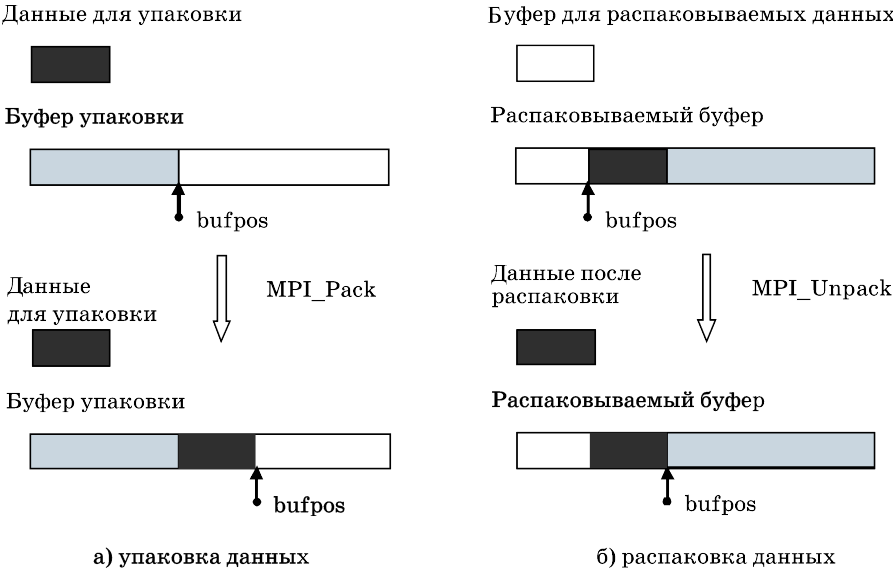


Рис. 5.8. Общая схема упаковки и распаковки данных

Начальное значение переменной *bufpos* должно быть сформировано до начала упаковки и далее устанавливается функцией `MPI_Pack`. Вызов функции `MPI_Pack` осуществляется последовательно для упаковки всех необходимых данных. Так, в ранее рассмотренном примере набора переменных *a*, *b* и *n* для их упаковки необходимо выполнить:

```
bufpos = 0;
MPI_Pack(&a, 1, MPI_DOUBLE, buf, buflen, &bufpos, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, buf, buflen, &bufpos, comm);
MPI_Pack(&n, 1, MPI_INT,    buf, buflen, &bufpos, comm);
```

Для определения необходимого размера буфера для упаковки может быть применена функция:

```
int MPI_Pack_size(int count, MPI_Datatype type, MPI_Comm comm,
  int *size),
```

где

— **count** — количество элементов в буфере;
— **type** — тип данных для упаковываемых элементов;
— **comm** — коммуникатор для упакованного сообщения;

— **size** — рассчитаный размер буфера.

После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа MPI_PACKED.

После получения сообщения с типом MPI_PACKED данные могут быть распакованы при помощи функции:

```
int MPI_Unpack(void *buf, int bufsize, int *bufpos, void *data,
  int count, MPI_Datatype type, MPI_Comm comm),
```

где

— **buf** — буфер памяти с упакованными данными;
— **bufsize** — размер буфера в байтах;
— **bufpos** — позиция начала данных в буфере (в байтах от начала буфера);
— **data** — буфер памяти для распаковываемых данных;
— **count** — количество элементов в буфере;
— **type** — тип распаковываемых данных;
— **comm** — коммуникатор для упакованного сообщения.

Функция MPI_Unpack распаковывает, начиная с позиции *bufpos*, очередную порцию данных из буфера *buf* и помещает распакованные данные в буфер *data*. Общая схема процедуры распаковки показана на рис. 5.8б.

Начальное значение переменной *bufpos* должно быть сформировано до начала распаковки и далее устанавливается функцией MPI_Unpack. Вызов функции MPI_Unpack осуществляется последовательно для распаковки всех упакованных данных, при этом порядок распаковки должен соответствовать порядку упаковки. Так, в ранее рассмотренном примере упаковки для распаковки упакованных данных необходимо выполнить:

```
bufpos = 0;
MPI_Unpack(buf, buflen, &bufpos, &a, 1, MPI_DOUBLE, comm);
MPI_Unpack(buf, buflen, &bufpos, &b, 1, MPI_DOUBLE, comm);
MPI_Unpack(buf, buflen, &bufpos, &n, 1, MPI_INT,    comm);
```

В заключение выскажем ряд рекомендаций по применению упаковки для формирования сообщений. Поскольку такой подход приводит к появлению дополнительных действий по упаковке и распаковке данных, то данный способ может быть оправдан при сравнительно небольших размерах сообщений и при малом количестве повторений. Упаковка и

распаковка может оказаться полезной при явном использовании буферов для буферизованного способа передачи данных.

5.6. Управление группами процессов и коммуникаторами

Рассмотрим теперь возможности MPI по управлению группами процессов и коммуникаторами.

Для изложения последующего материала напомним ряд понятий и определений, приведенных в начале данной лекции.

Процессы параллельной программы объединяются в *группы*. В группу могут входить все процессы параллельной программы; с другой стороны, в группе может находиться только часть имеющихся процессов. Один и тот же процесс может принадлежать нескольким группам. Управление группами процессов предпринимается для создания на их основе коммуникаторов.

Под коммуникатором в MPI понимается специально создаваемый служебный объект, который объединяет в своем составе группу процессов и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных. Парные операции передачи данных выполняются только для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов коммуникатора. Создание коммуникаторов предпринимается для уменьшения области действия коллективных операций и для устранения взаимовлияния разных выполняемых частей параллельной программы. Важно еще раз подчеркнуть – коммуникационные операции, выполняемые с использованием разных коммуникаторов, являются независимыми и не влияют друг на друга.

Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором `MPI_COMM_WORLD`.

При необходимости передачи данных между процессами из разных групп необходимо создавать определенные в стандарте MPI-2 глобальные коммуникаторы (*intercommunicator*). Взаимодействие между процессами разных групп оказывается необходимым в достаточно редких ситуациях, в данном учебном материале не рассматривается и может служить темой для самостоятельного изучения – см., например, [4, 40 – 42, 57].

5.6.1. Управление группами

Группы процессов могут быть созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, свя-

занная с предопределенным коммуникатором MPI_COMM_WORLD. Также иногда может быть полезным коммуникатор MPI_COMM_SELF, определенный для каждого процесса параллельной программы и включающий только этот процесс.

Для получения группы, связанной с существующим коммуникатором, применяется функция:

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group),
```

где

— **comm** — коммуникатор;
— **group** — группа, связанная с коммуникатором.

Далее, на основе существующих групп, могут быть созданы новые группы:

- создание новой группы *newgroup* из существующей группы *oldgroup*, которая будет включать в себя *n* процессов — их ранги перечисляются в массиве *ranks*:

```
int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks,
  MPI_Group *newgroup),
```

где

— **oldgroup** — существующая группа;
— **n** — число элементов в массиве *ranks*;
— **ranks** — массив рангов процессов, которые будут включены в новую группу;
— **newgroup** — созданная группа;

- создание новой группы *newgroup* из группы *oldgroup*, которая будет включать в себя *n* процессов, чьи ранги не совпадают с рангами, перечисленными в массиве *ranks*:

```
int MPI_Group_excl(MPI_Group oldgroup, int n, int *ranks,
  MPI_Group *newgroup),
```

где

— **oldgroup** — существующая группа;
— **n** — число элементов в массиве *ranks*;
— **ranks** — массив рангов процессов, которые будут исключены из новой группы;
— **newgroup** — созданная группа.

Для получения новых групп над имеющимися группами процессов могут быть выполнены операции объединения, пересечения и разности:

• создание новой группы *newgroup* как объединения групп *group*1 и *group*2:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup),
```

где

— **group1** — первая группа;
— **group2** — вторая группа;
— **newgroup** — объединение групп;

• создание новой группы *newgroup* как пересечения групп *group*1 и *group*2:

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup),
```

где

— **group1** — первая группа;
— **group2** — вторая группа;
— **newgroup** — пересечение групп;

• создание новой группы *newgroup* как разности групп *group*1 и *group*2:

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup),
```

где

— **group1** — первая группа;
— **group2** — вторая группа;
— **newgroup** — разность групп.

При конструировании групп может оказаться полезной специальная пустая группа MPI_COMM_EMPTY.

Ряд функций MPI обеспечивает получение информации о группе процессов:

• получение количества процессов в группе:

```
int MPI_Group_size(MPI_Group group, int *size),
```

где

— **group** — группа;
— **size** — число процессов в группе;

• получение ранга текущего процесса в группе:

```
int MPI_Group_rank(MPI_Group group, int *rank),
```

где

— **group** — группа;
— **rank** — ранг процесса в группе.

После завершения использования группа должна быть удалена:

```
int MPI_Group_free(MPI_Group *group),
```

где

— **group** — группа, подлежащая удалению
(выполнение данной операции не затрагивает коммуникаторы, в которых используется удаляемая группа).

5.6.2. Управление коммуникаторами

Отметим прежде всего, что в данном пункте рассматривается управление *интракоммуникаторами*, используемыми для операций передачи данных внутри одной группы процессов. Как отмечалось ранее, применение *интеркоммуникаторов* для обменов между группами процессов выходит за пределы данного учебного материала.

Для создания новых коммуникаторов существуют два основных способа:

• дублирование уже существующего коммуникатора:

```
int MPI_Comm_dup(MPI_Comm oldcom, MPI_comm *newcom),
```

где

— **oldcom** — существующий коммуникатор, копия которого создается;
— **newcom** — новый коммуникатор;

• создание нового коммуникатора из подмножества процессов существующего коммуникатора:

```
int MPI_Comm_create(MPI_Comm oldcom, MPI_Group group,
  MPI_Comm *newcom),
```

где

— **oldcom** — существующий коммуникатор;
— **group** — подмножество процессов коммуникатора *oldcom*;
— **newcom** — новый коммуникатор.

Дублирование коммуникатора может применяться, например, для устранения возможности пересечения по тегам сообщений в разных частях параллельной программы (в том числе и при использовании функций разных программных библиотек).

Следует отметить также, что операция создания коммуникаторов является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Для пояснения рассмотренных функций можно привести пример создания коммуникатора, в котором содержатся все процессы, кроме процесса, имеющего ранг 0 в коммуникаторе `MPI_COMM_WORLD` (такой коммуникатор может быть полезен для поддержки схемы организации параллельных вычислений «менеджер – исполнители» – см. лекцию 4):

```
MPI_Group WorldGroup, WorkerGroup;
MPI_Comm Workers;
int ranks[1];
ranks[0] = 0;
// Получение группы процессов в MPI_COMM_WORLD
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
// Создание группы без процесса с рангом 0
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);
// Создание коммуникатора по группе
MPI_Comm_create(MPI_COMM_WORLD, WorkerGroup, &Workers);
...
MPI_Group_free(&WorkerGroup);
MPI_Comm_free(&Workers);
```

Быстрый и полезный способ одновременного создания нескольких коммуникаторов обеспечивает функция:

```
int MPI_Comm_split(MPI_Comm oldcomm, int split, int key,
 MPI_Comm *newcomm),
```

где

— **oldcomm** — исходный коммуникатор;
— **split** — номер коммуникатора, которому должен принадлежать процесс;

— **key** — порядок ранга процесса в создаваемом коммуникаторе;
— **newcomm** — создаваемый коммуникатор.

Создание коммуникаторов относится к коллективным операциям, и поэтому вызов функции `MPI_Comm_split` должен быть выполнен в каждом процессе коммуникатора *oldcomm*. В результате выполнения функции процессы разделяются на непересекающиеся группы с одинаковыми значениями параметра *split*. На основе сформированных групп создается набор коммуникаторов. Для того чтобы указать, что процесс не должен входить ни в один из создаваемых коммуникаторов, необходимо воспользоваться константой `MPI_UNDEFINED` в качестве значения параметра *split*. При создании коммуникаторов для рангов процессов в новом коммуникаторе выбирается такой порядок нумерации, чтобы он соответствовал порядку значений параметров *key* (процесс с большим значением параметра *key* получает больший ранг, процессы с одинаковым значением параметра *key* сохраняют свою относительную нумерацию).

В качестве примера можно рассмотреть задачу представления набора процессов в виде двумерной решетки. Пусть $p=q*q$ есть общее количество процессов; следующий далее фрагмент программы обеспечивает получение коммуникаторов для каждой строки создаваемой топологии:

```
MPI_Comm comm;
int rank, row;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
row = rank / q;
MPI_Comm_split(MPI_COMM_WORLD, row, rank, &comm);
```

При выполнении данного примера, например, при $p=9$, процессы с рангами (0, 1, 2) образуют первый коммуникатор, процессы с рангами (3, 4, 5) – второй и т. д.

После завершения использования коммуникатор должен быть удален, для чего используется функция:

```
int MPI_Comm_free(MPI_Comm *comm),
```

где

— **comm** — коммуникатор, подлежащий удалению.

5.7. Виртуальные топологии

Под *топологией* вычислительной системы обычно понимается структура узлов сети и линий связи между этими узлами. Топология может быть

представлена в виде графа, в котором вершины есть процессоры (процессы) системы, а дуги соответствуют имеющимся линиям (каналам) связи.

Как уже отмечалось ранее, парные операции передачи данных могут быть выполнены между любыми процессами одного и того же коммуникатора, а в коллективной операции принимают участие все процессы коммуникатора. В этом плане, логическая топология линий связи между процессами в параллельной программе имеет структуру полного графа (независимо от наличия реальных физических каналов связи между процессорами).

Понятно, что физическая топология системы является аппаратно реализуемой и изменению не подлежит (хотя существуют и программируемые средства построения сетей). Но, оставляя неизменной физическую основу, мы можем организовать логическое представление любой необходимой *виртуальной топологии*. Для этого достаточно, например, сформировать тот или иной механизм дополнительной адресации процессов.

Использование виртуальных процессов может оказаться полезным в силу ряда разных причин. Виртуальная топология, например, может больше соответствовать имеющейся структуре линий передачи данных. Применение виртуальных топологий может заметно упростить в ряде случаев представление и реализацию параллельных алгоритмов.

В MPI поддерживаются два вида топологий – *прямоугольная решетка* произвольной размерности (*декартова топология*) и топология *графа* произвольного вида. Следует отметить, что имеющиеся в MPI функции обеспечивают лишь получение новых логических систем адресации процессов, соответствующих формируемым виртуальным топологиям. Выполнение же всех коммуникационных операций должно осуществляться, как и ранее, при помощи обычных функций передачи данных с использованием исходных рангов процессов.

5.7.1. Декартовы топологии (решетки)

Декартовы топологии, в которых множество процессов представляется в виде прямоугольной решетки (см. п. 1.4.1 и рис. 1.7), а для указания процессов используется декартова система координат, широко применяются во многих задачах для описания структуры имеющихся информационных зависимостей. В числе примеров таких задач – матричные алгоритмы (см. лекции 6 и 7) и сеточные методы решения дифференциальных уравнений в частных производных (см. лекцию 11).

Для создания декартовой топологии (решетки) в MPI предназначена функция:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
  int *periods, int reorder, MPI_Comm *cartcomm),
```

где

— **oldcomm** — исходный коммуникатор;
— **ndims** — размерность декартовой решетки;
— **dims** — массив длины *ndims*, задает количество процессов в каждом измерении решетки;
— **periods** — массив длины *ndims*, определяет, является ли решетка периодической вдоль каждого измерения;
— **reorder** — параметр допустимости изменения нумерации процессов;
— **cartcomm** — создаваемый коммуникатор с декартовой топологией процессов.

Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Для пояснения назначения параметров функции `MPI_Cart_create` рассмотрим пример создания двумерной решетки 4x4, в которой строки и столбцы имеют кольцевую структуру (за последним процессом следует первый процесс):

```
// Создание двумерной решетки 4x4
MPI_Comm GridComm;
int dims[2], periods[2], reorder = 1;
dims[0]    = dims[1]    = 4;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
  &GridComm);
```

Следует отметить, что в силу кольцевой структуры измерений сформированная в рамках примера топология является тором.

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
  int *coords),
```

где

— **comm** — коммуникатор с топологией решетки;
— **rank** — ранг процесса, для которого определяются декартовы координаты;
— **ndims** — размерность решетки;

— **coords** — возвращаемые функцией декартовы координаты процесса.

Обратное действие – определение ранга процесса по его декартовым координатам – обеспечивается при помощи функции:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
```

где

— **comm** — коммуникатор с топологией решетки;
— **coords** — декартовы координаты процесса;
— **rank** — возвращаемый функцией ранг процесса.

Полезная во многих приложениях процедура разбиения решетки на подрешетки меньшей размерности обеспечивается при помощи функции:

```
int MPI_Cart_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
```

где

— **comm** — исходный коммуникатор с топологией решетки;
— **subdims** — массив для указания, какие измерения должны остаться в создаваемой подрешетке;
— **newcomm** — создаваемый коммуникатор с подрешеткой.

Операция создания подрешеток также является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора. В ходе своего выполнения функция `MPI_Cart_sub` определяет коммуникаторы для каждого сочетания координат фиксированных измерений исходной решетки.

Для пояснения функции `MPI_Cart_sub` дополним ранее рассмотренный пример создания двумерной решетки и определим коммуникаторы с декартовой топологией для каждой строки и столбца решетки в отдельности:

```
// Создание коммуникаторов для каждой строки и столбца решетки
MPI_Comm RowComm, ColComm;
int subdims[2];
// Создание коммуникаторов для строк
subdims[0] = 0;  // фиксации измерения
subdims[1] = 1;  // наличие данного измерения в подрешетке
MPI_Cart_sub(GridComm, subdims, &RowComm);
// Создание коммуникаторов для столбцов
subdims[0] = 1;
subdims[1] = 0;
MPI_Cart_sub(GridComm, subdims, &ColComm);
```

В приведенном примере для решетки размером 4x4 создаются 8 коммуникаторов, по одному для каждой строки и столбца решетки. Для каждого процесса определяемые коммуникаторы *RowComm* и *ColComm* соответствуют строке и столбцу процессов, к которым данный процесс принадлежит.

Дополнительная функция `MPI_Cart_shift` обеспечивает поддержку процедуры последовательной передачи данных по одному из измерений решетки (*операция сдвига данных* – см. лекцию 3). В зависимости от периодичности измерения решетки, по которому выполняется сдвиг, различаются два типа данной операции:

- *циклический сдвиг* на k элементов вдоль измерения решетки – в этой операции данные от процесса i пересылаются процессу $(i+k) \bmod dim$, где dim есть размер измерения, вдоль которого производится сдвиг;
- *линейный сдвиг* на k позиций вдоль измерения решетки – в этом варианте операции данные от процесса i пересылаются процессу $i+k$ (если таковой существует).

Функция `MPI_Cart_shift` обеспечивает получение рангов процессов, с которыми текущий процесс (процесс, вызвавший функцию `MPI_Cart_shift`) должен выполнить обмен данными:

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp, int *source,
  int *dst),
```

где

— **comm** — коммуникатор с топологией решетки;
— **dir** — номер измерения, по которому выполняется сдвиг;
— **disp** — величина сдвига (при отрицательных значениях сдвиг производится к началу измерения);
— **source** — ранг процесса, от которого должны быть получены данные;
— **dst** — ранг процесса, которому должны быть отправлены данные.

Следует отметить, что функция `MPI_Cart_shift` только определяет ранги процессов, между которыми должен быть выполнен обмен данными в ходе операции сдвига. Непосредственная передача данных может быть выполнена, например, при помощи функции `MPI_Sendrecv`.

5.7.2. Топологии графа

Сведения по функциям MPI для работы с виртуальными топологиями типа граф будут рассмотрены более кратко – дополнительная информация может быть получена, например, в [4, 40 – 42, 57].

Для создания коммуникатора с топологией типа граф в MPI предназначена функция:

```
int MPI_Graph_create(MPI_Comm oldcom, int nnodes, int *index,
  int *edges, int reorder, MPI_Comm *graphcom),
```

где

- **oldcom** — исходный коммуникатор;
- **nnodes** — количество вершин графа;
- **index** — количество исходящих дуг для каждой вершины;
- **edges** — последовательный список дуг графа;
- **reorder** — параметр допустимости изменения нумерации процессов;
- **graphcom** – создаваемый коммуникатор с топологией типа граф.

Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

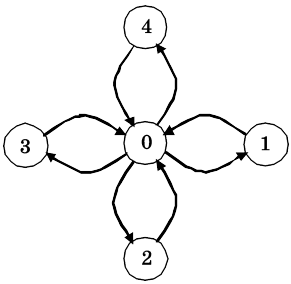


Рис. 5.9. Пример графа для топологии типа звезда

Для примера создадим топологию графа со структурой, представленной на рис. 5.9. В этом случае количество процессов равно 5, порядки вершин (количества исходящих дуг) принимают значения (4, 1, 1, 1, 1), а матрица инцидентности (номера вершин, для которых дуги являются входящими) имеет вид:

Процессы	**Линии связи**
0	1, 2, 3, 4
1	0
2	0
3	0
4	0

Для создания топологии с графом данного вида необходимо выполнить следующий программный код:

```
/* Создание топологии типа звезда */
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Приведем еще две полезные функции для работы с топологиями графа. Количество соседних процессов, в которых от проверяемого процесса есть выходящие дуги, может быть получено при помощи функции:

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank,
  int *nneighbors),
```

где

— **comm** — коммуникатор с топологией типа граф;
— **rank** — ранг процесса в коммуникаторе;
— **nneighbors** — количество соседних процессов.

Получение рангов соседних вершин обеспечивается функцией:

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int mneighbors,
  int *neighbors),
```

где

— **comm** — коммуникатор с топологией типа граф;
— **rank** — ранг процесса в коммуникаторе;
— **mneighbors** — размер массива *neighbors*;
— **neighbors** — ранги соседних в графе процессов.

5.8. Дополнительные сведения о MPI

5.8.1. Разработка параллельных программ с использованием MPI на алгоритмическом языке Fortran

При разработке параллельных программ с использованием MPI на алгоритмическом языке Fortran существует не так много особенностей по сравнению с применением алгоритмического языка C:

- все константы, переменные и функции объявляются в подключаемом файле mpif.h;

- подпрограммы библиотеки MPI являются процедурами и, тем самым, вызываются при помощи оператора вызова процедур CALL;
- коды завершения передаются через дополнительный параметр целого типа, располагаемый на последнем месте в списке параметров процедур (кроме MPI_Wtime и MPI_Wtick);
- все структуры (такие, например, как переменная status) являются массивами целого типа, размеры и номера ячеек которых описаны символическими константами (такими, как MPI_STATUS_SIZE для статуса операций);
- типы MPI_Comm и MPI_Datatype представлены целых типом INTEGER.

В качестве принятых соглашений при разработке программ на языке Fortran рекомендуется записывать имена подпрограмм с применением прописных символов.

В качестве примера приведем варианты программы из п. 5.2.1.5 на алгоритмических языках Fortran 77 и Fortran 90.

Программа 5.3. Параллельная программа на языке Fortran 77

```
! Пример программы, использующей MPI на Fortran 77
      program main
      include 'mpif.h'
      integer ProcNum, ProcRank, RecvRank, ierr
      integer i
      integer st(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, ProcNum,  ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, ProcRank, ierr)

      if (ProcRank .gt. 0) goto 20
c Действия, выполняемые только процессом с рангом 0
      print *, "Hello from process ", ProcRank
      do 10 i = 1, ProcNum - 1
        call MPI_RECV(RecvRank, 1, MPI_INTEGER, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, st, ierr)
        print *, "Hello from process ", RecvRank
10    continue
      goto 30
c Сообщение, отправляемое всеми процессами, кроме процесса
c рангом 0
 20    call MPI_SEND(ProcRank, 1, MPI_INTEGER, 0, 0,
         MPI_COMM_WORLD, ierr)
```

```
30    call MPI_FINALIZE(ierr)
      stop
      end
```

Программа 5.4. Параллельная программа на языке Fortran 90

```
! Пример программы, использующей MPI на Fortran 90
program main
use mpi

integer ProcNum, ProcRank, RecvRank, ierr
integer status(MPI_STATUS_SIZE)
integer i
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, ProcNum,  ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, ProcRank, ierr)
if (ProcRank .EQ. 0) then
! Действия, выполняемые только процессом с рангом 0
  print *, "Hello from process ", ProcRank
  do i = 1, procnum - 1
    call MPI_RECV(RecvRank, 1, MPI_INTEGER, MPI_ANY_SOURCE,
      MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
    print *, "Hello from process ", RecvRank
  enddo
else
c Сообщение, отправляемое всеми процессами, кроме процесса
c с рангом 0
  call MPI_SEND(ProcRank, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD,
    ierr)
endif
call MPI_FINALIZE(ierr)
stop
end
```

5.8.2. Общая характеристика среды выполнения MPI-программ

Для проведения параллельных вычислений в вычислительной системе должна быть установлена среда выполнения MPI-программ, которая бы обеспечивала разработку, компиляцию, компоновку и выполнение параллельных программ. Для выполнения первой части перечисленных действий – разработки, компиляции, компоновки – как правило, достаточно обычных средств разработки программ (таких, например, как Microsoft

Visual Studio), необходимо лишь наличие той или иной библиотеки MPI. Для выполнения же параллельных программ от среды выполнения требуется ряд дополнительных возможностей, среди которых наличие средств указания используемых процессоров, операций удаленного запуска программ и т. п. Крайне желательно также наличие в среде выполнения средств профилирования, трассировки и отладки параллельных программ.

Здесь, к сожалению, стандартизация заканчивается[1]. Существует несколько различных сред выполнения MPI-программ. В большинстве случаев эти среды создаются совместно с разработкой тех или иных вариантов MPI-библиотек. Обычно выбор реализации MPI-библиотеки, установка среды выполнения и подготовка инструкций по использованию осуществляются администратором вычислительной системы. Как правило, информационные ресурсы сети Интернет, на которых располагаются свободно используемые реализации MPI, и промышленные версии MPI содержат исчерпывающую информацию о процедурах установки MPI, и выполнение всех необходимых действий не составляет большого труда.

Запуск MPI-программы также зависит от среды выполнения, но в большинстве случаев данное действие выполняется при помощи команды **mpirun**[2]. В числе возможных параметров этой команды:

- *режим выполнения* – локальный или многопроцессорный; локальный режим обычно указывается при помощи ключа -localonly, при выполнении параллельной программы в локальном режиме все процессы программы располагаются на компьютере, с которого был произведен запуск программы. Такой способ выполнения чрезвычайно полезен для начальной проверки работоспособности и отладки параллельной программы, часть такой работы может быть выполнена даже на отдельном компьютере вне рамок многопроцессорной вычислительной системы;
- *количество процессов*, которые необходимо создать при запуске параллельной программы;
- *состав используемых процессоров*, определяемый тем или иным конфигурационным файлом;
- *исполняемый файл* параллельной программы;
- *командная строка* с параметрами для выполняемой программы.

Существует и значительное количество других параметров, но они обычно используются при разработке достаточно сложных параллельных программ – их описание может быть получено в справочной информации по соответствующей среде выполнения MPI-программ.

[1] Стандарт MPI-2 дает некоторые рекомендации о том, как должна быть организована среда выполнения MPI-программ, однако они не являются обязательными.

[2] Стандарт MPI-2 рекомендует использовать команду **mpiexec**.

При запуске программы на нескольких компьютерах исполняемый файл программы должен быть скопирован на все эти компьютеры или же должен находиться на общем доступном для всех компьютеров ресурсе.

Дополнительная информация по средам выполнения параллельных программ на кластерных системах может быть получена, например, в [70, 71].

5.8.3. Дополнительные возможности стандарта MPI-2

Как уже отмечалось, стандарт MPI-2 был принят в 1997 г. Несмотря на достаточно большой период времени, прошедший с тех пор, использование данного варианта стандарта все еще ограничено. Среди основных причин такой ситуации можно назвать обычный консерватизм разработчиков программного обеспечения, сложность реализации нового стандарта и т.п. Важный момент состоит также в том, что возможностей MPI-1 достаточно для реализации многих параллельных алгоритмов, а сфера применимости дополнительных возможностей MPI-2 оказывается не столь широкой.

Для знакомства со стандартом MPI-2 может быть рекомендован информационный ресурс http://www.mpiforum.org, а также работа [42]. Здесь же дадим краткую характеристику дополнительных возможностей стандарта MPI-2:

- *динамическое порождение процессов*, при котором процессы параллельной программы могут создаваться и уничтожаться в ходе выполнения;
- *одностороннее взаимодействие процессов*, что позволяет быть инициатором операции передачи и приема данных только одному процессу;
- *параллельный ввод/вывод*, обеспечивающий специальный интерфейс для работы процессов с файловой системой;
- *расширение возможностей коллективных операций*, в числе которых, например, взаимодействие через глобальные коммуникаторы (intercommunicator);
- *интерфейс для алгоритмических языков C++ и Fortran 90.*

5.9. Краткий обзор лекции

Данная лекция посвящена рассмотрению методов параллельного программирования для вычислительных систем с распределенной памятью с использованием MPI.

В самом начале лекции отмечается, что MPI – *интерфейс передачи сообщений* (*message passing interface*) – является в настоящий момент одним из основных подходов к разработке параллельных программ для вычислительных систем с распределенной памятью. Использование MPI позволя-

ет распределить вычислительную нагрузку и организовать информационное взаимодействие (*передачу данных*) между процессорами. Сам термин MPI означает, с одной стороны, стандарт, которому должны удовлетворять средства организации передачи сообщений, а с другой стороны, обозначает программные библиотеки, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI.

В подразделе 5.1 рассматривается ряд понятий и определений, являющихся основополагающими для стандарта MPI. Так, дается представление о *параллельной программе* как множестве одновременно выполняемых *процессов*. При этом процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). Далее приводится краткая характеристика понятий для операций передачи сообщений, типов данных, коммуникаторов и виртуальных топологий.

В подразделе 5.2 проводится быстрое и простое введение в разработку параллельных программ с использованием MPI. Излагаемого в подразделе материала достаточно для начала разработки параллельных программ разного уровня сложности.

В подразделе 5.3 излагается материал, связанный с *операциями передачи данных между двумя процессами*. Здесь подробно характеризуются имеющиеся в MPI режимы выполнения операций – стандартный, синхронный, буферизованный, по готовности. Для всех рассмотренных операций обсуждается возможность организации неблокирующих обменов данными между процессами.

В подразделе 5.4 рассматриваются *коллективные операции передачи данных*. Изложение материала соответствует последовательности изучения коммуникационных операций, использованной в лекции 3. Основной вывод данного подраздела состоит в том, что MPI обеспечивает поддержку практически всех основных операций информационных обменов между процессами.

В подразделе 5.5 излагается материал, связанный с использованием в MPI *производных типов данных*. В подразделе представлены все основные способы конструирования производных типов – непрерывный, векторный, индексный и структурный. Обсуждается также возможность явного формирования сложных сообщений при помощи упаковки и распаковки данных.

В подразделе 5.6 обсуждаются вопросы *управления группами процессов и коммуникаторами*. Рассматриваемые в подразделе возможности MPI позволяют управлять областями действия коллективных операций и исключить взаимовлияние разных выполняемых частей параллельной программы.

В подразделе 5.7 рассматриваются возможности MPI по использованию *виртуальных топологий*. В подразделе представлены топологии, под-

держиваемые MPI, – *прямоугольная решетка* произвольной размерности (*декартова топология*) и топология *графа* любого необходимого вида.

В подразделе 5.8 приводятся дополнительные сведения о MPI. В их числе обсуждаются вопросы разработки параллельных программ с использованием MPI на алгоритмическом языке Fortran, дается краткая характеристика сред выполнения MPI-программ и приводится обзор дополнительных возможностей стандарта MPI-2.

5.10. Обзор литературы

Имеется ряд источников, в которых может быть получена информация о MPI. Прежде всего, это информационный ресурс Интернет с описанием стандарта MPI: http://www.mpiforum.org. Одна из наиболее распространенных реализаций MPI – библиотека MPICH – представлена на http://www-unix.mcs.anl.gov/mpi/mpich (библиотека MPICH2 с реализацией стандарта MPI-2 содержится на http://www-unix.mcs.anl.gov/mpi/mpich2). Русскоязычные материалы о MPI имеются на сайте http://www.parallel.ru.

Среди опубликованных изданий могут быть рекомендованы работы [4, 40 – 42, 57]. Описание стандарта MPI-2 может быть получено в [42]. Среди русскоязычных изданий могут быть рекомендованы работы [2, 4, 12].

Следует отметить также работу [63], в которой изучение MPI проводится на примере ряда типовых задач параллельного программирования – матричных вычислений, сортировки, обработки графов и др.

5.11. Контрольные вопросы

1. Какой минимальный набор средств является достаточным для организации параллельных вычислений в системах с распределенной памятью?
2. В чем состоит важность стандартизации средств передачи сообщений?
3. Что следует понимать под параллельной программой?
4. В чем различие понятий процесса и процессора?
5. Какой минимальный набор функций MPI позволяет начать разработку параллельных программ?
6. Как описываются передаваемые сообщения?
7. Как можно организовать прием сообщений от конкретных процессов?
8. Как определить время выполнения MPI-программы?
9. В чем различие парных и коллективных операций передачи данных?
10. Какая функция MPI обеспечивает передачу данных от одного процесса всем процессам?

11. Что понимается под операцией редукции?
12. В каких ситуациях следует применять барьерную синхронизацию?
13. Какие режимы передачи данных поддерживаются в MPI?
14. Как организуется неблокирующий обмен данными в MPI?
15. В чем состоит понятие тупика? Когда функция одновременного выполнения передачи и приема гарантирует отсутствие тупиковых ситуаций?
16. Какие коллективные операции передачи данных предусмотрены в MPI?
17. Что понимается под производным типом данных в MPI?
18. Какие способы конструирования типов имеются в MPI?
19. В каких ситуациях может быть полезна упаковка и распаковка данных?
20. Что понимается в MPI под коммуникатором?
21. Для чего может потребоваться создание новых коммуникаторов?
22. Что понимается в MPI под виртуальной топологией?
23. Какие виды топологий предусмотрены в MPI?
24. Для чего может оказаться полезным использование виртуальных топологий?
25. В чем состоят особенности разработки параллельных программ с использованием MPI на алгоритмическом языке Fortran?
26. Какие основные дополнительные возможности предусмотрены в стандарте MPI-2?

5.12. Задачи и упражнения

Подраздел 5.2.

1. Разработайте программу для нахождения минимального (максимального) значения среди элементов вектора.
2. Разработайте программу для вычисления скалярного произведения двух векторов.
3. Разработайте программу, в которой два процесса многократно обмениваются сообщениями длиной n байт. Выполните эксперименты и оцените зависимость времени выполнения операции передачи данных от длины сообщения. Сравните с теоретическими оценками, построенными по модели Хокни.

Подраздел 5.3.

4. Подготовьте варианты ранее разработанных программ с разными режимами выполнения операций передачи данных. Сравните время выполнения операций передачи данных при разных режимах работы.
5. Подготовьте варианты ранее разработанных программ с использованием неблокирующего способа выполнения операций передачи

данных. Оцените количество вычислительных операций, необходимое для того, чтобы полностью совместить передачу данных и вычисления. Разработайте программу, в которой бы полностью отсутствовали задержки вычислений из-за ожидания передаваемых данных.

6. Выполните задание 3 с использованием операции одновременного выполнения передачи и приема данных. Сравните результаты вычислительных экспериментов.

Подраздел 5.4.

7. Разработайте программу-пример для каждой имеющейся в MPI коллективной операции.
8. Разработайте реализации коллективных операций при помощи парных обменов между процессами. Выполните вычислительные эксперименты и сравните время выполнения разработанных программ и функций MPI для коллективных операций.
9. Разработайте программу, выполните эксперименты и сравните результаты для разных алгоритмов реализации операции сбора, обработки и рассылки данных всех процессам (функция MPI_Allreduce).

Подраздел 5.5.

10. Разработайте программу-пример для каждого имеющегося в MPI способа конструирования производных типов данных.
11. Разработайте программу-пример с использованием функций упаковки и распаковки данных. Выполните эксперименты и сравните с результатами при использовании производных типов данных.
12. Разработайте производные типы данных для строк, столбцов, диагоналей матриц.
13. Разработайте программу-пример для каждой из рассмотренных функций для управления процессами и коммуникаторами.
14. Разработайте программу для представления множества процессов в виде прямоугольной решетки. Создайте коммуникаторы для каждой строки и столбца процессов. Выполните коллективную операцию для всех процессов и для одного из созданных коммуникаторов. Сравните время выполнения операции.
15. Изучите самостоятельно и разработайте программы-примеры для передачи данных между процессами разных коммуникаторов.

Подраздел 5.7.

16. Разработайте программу-пример для декартовой топологии.
17. Разработайте программу-пример для топологии графа.
18. Разработайте подпрограммы для создания некоторого набора дополнительных виртуальных топологий (звезда, дерево и др.).

Лекция 6. Параллельные методы умножения матрицы на вектор

В лекции рассматривается задача умножения матрицы на вектор. Приводится постановка задачи и последовательный алгоритм ее решения. Описываются методы разделения матрицы между процессорами вычислительной системы, которые необходимы для параллельной реализации матричных операций. Далее излагаются три возможных подхода к параллельной реализации алгоритма умножения матрицы на вектор.

Ключевые слова: параллельные алгоритмы матричных вычислений, принципы распараллеливания, параллелизм по данным, ленточное и блочное разбиения матрицы, ускорение, эффективность, вычислительная и коммуникационная сложность, теоретическая оценка времени выполнения алгоритма, программная реализация, операции передачи данных, MPI, вычислительный эксперимент.

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

С учетом значимости эффективного выполнения матричных расчетов многие стандартные библиотеки программ содержат процедуры для различных матричных операций. Объем программного обеспечения для обработки матриц постоянно увеличивается – разрабатываются новые экономные структуры хранения для матриц специального типа (треугольных, ленточных, разреженных и т.п.), создаются различные высокоэффективные машинно-зависимые реализации алгоритмов, проводятся теоретические исследования для поиска более быстрых методов матричных вычислений.

Являясь вычислительно трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений. С одной стороны, использование высокопроизводительных многопроцессорных систем позволяет существенно повысить сложность решаемых задач. С другой стороны, в силу своей достаточно простой формулировки матричные операции предоставляют прекрасную возможность для демонстрации многих приемов и методов параллельного программирования.

В данной лекции обсуждаются методы параллельных вычислений для операции матрично-векторного умножения, в следующей лекции

(лекция 7) излагается более общий случай – задача перемножения матриц. Важный вид матричных вычислений – решение систем линейных уравнений – представлен в лекции 8. Общий для всех перечисленных задач вопрос разделения обрабатываемых матриц между параллельно работающими процессорами рассматривается в первом подразделе лекции 6.

При изложении следующего материала будем полагать, что рассматриваемые матрицы являются *плотными* (*dense*), то есть число нулевых элементов в них незначительно по сравнению с общим количеством элементов матриц.

6.1. Принципы распараллеливания

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данное свойство свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов, и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

1. Ленточное разбиение матрицы. При *ленточном* (*block-striped*) разбиении каждому процессору выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (рис. 6.1). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, например, матрица A представляется в виде (см. рис. 6.1)

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = ik + j,\ 0 \le j < k,\ k = m/p, \qquad (6.1)$$

где $a_i = (a_{i1}, a_{i2}, ... a_{in})$, $0 \le i < m$, есть i-я строка матрицы A (предполагается, что количество строк m кратно числу процессоров p, т.е. $m = k \cdot p$). Во всех алгоритмах матричного умножения и умножения матрицы на вектор, которые будут рассмотрены в этой и следующей лекциях, применяется разделение данных на непрерывной основе.

Другой возможный подход к формированию полос состоит в применении той или иной схемы *чередования* (*цикличности*) строк или столбцов. Как правило, для чередования используется число процессоров p – в этом случае при горизонтальном разбиении матрица A принимает вид

$$A = (A_0, A_1, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = i + jp,\ 0 \le j < k,\ k = m/p\ . \qquad (6.2)$$

Циклическая схема формирования полос может оказаться полезной для лучшей балансировки вычислительной нагрузки процессоров (например, при решении системы линейных уравнений с использованием метода Гаусса – см. лекцию 8).

2. Блочное разбиение матрицы. При *блочном* (*chessboard block*) разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров составляет $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$. Представим исходную матрицу A в виде набора прямоугольных блоков следующим образом:

$$A = \begin{pmatrix} A_{00} & A_{02} & ...A_{0q-1} \\ & ... & \\ A_{s-11} & A_{s-12} & ...A_{s-1q-1} \end{pmatrix},$$

где A_{ij} — блок матрицы, состоящий из элементов:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & ...a_{i_0 j_{l-1}} \\ & ... & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix},\ \begin{matrix} i_v = ik + v,\ 0 \le v < k,\ k = m/s, \\ j_u = jl + u,\ 0 \le u \le l,\ l = n/q. \end{matrix} \qquad (6.3)$$

При таком подходе целесообразно, чтобы вычислительная система имела физическую или, по крайней мере, логическую топологию процессорной решетки из s строк и q столбцов. В этом случае при разделении данных на непрерывной основе процессоры, соседние в структуре решетки, обрабатывают смежные блоки исходной матрицы. Следует отметить, однако, что и для блочной схемы может быть применено циклическое чередование строк и столбцов.

В данной лекции рассматриваются три параллельных алгоритма для умножения квадратной матрицы на вектор. Каждый подход основан на разном типе распределения исходных данных (элементов матрицы и вектора) между процессорами. Разделение данных меняет схему взаимодействия процессоров, поэтому каждый из представленных методов существенным образом отличается от двух остальных.

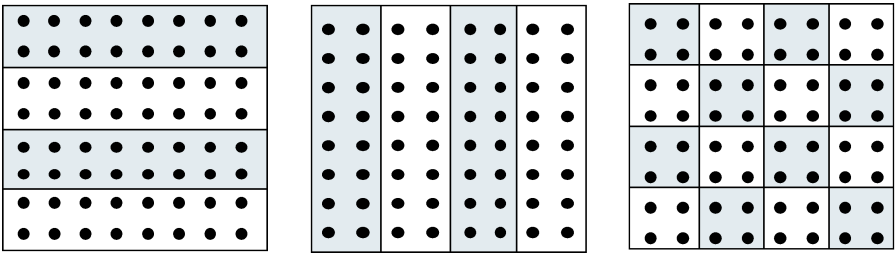


Рис. 6.1. Способы распределения элементов матрицы между процессорами вычислительной системы

6.2. Постановка задачи

В результате умножения матрицы A размерности $m \times n$ и вектора b, состоящего из n элементов, получается вектор c размера m, каждый i-й элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора b:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j, 1 \le i \le m. \quad (6.4)$$

Тем самым получение результирующего вектора c предполагает повторение m однотипных операций по умножению строк матрицы A и вектора b. Каждая такая операция включает умножение элементов строки матрицы и вектора b (n операций) и последующее суммирование полученных произведений (n-1 операций). Общее количество необходимых скалярных операций есть величина

$$T_1 = m \cdot (2n - 1)$$

6.3. Последовательный алгоритм

Последовательный алгоритм умножения матрицы на вектор может быть представлен следующим образом.

Алгоритм 6.1. Последовательный алгоритм умножения матрицы на вектор

```
// Алгоритм 6.1
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < m; i++){
```

```
    c[i] = 0;
    for (j = 0; j < n; j++){
      c[i] += A[i][j]*b[j]
    }
  }
```

Матрично-векторное умножение – это последовательность вычисления скалярных произведений. Поскольку каждое вычисление скалярного произведения векторов длины n требует выполнения n операций умножения и n-1 операций сложения, его трудоемкость порядка $O(n)$. Для выполнения матрично-векторного умножения необходимо осуществить m операций вычисления скалярного произведения, таким образом, алгоритм имеет трудоемкость порядка $O(mn)$.

6.4. Разделение данных

При выполнении параллельных алгоритмов умножения матрицы на вектор, кроме матрицы A, необходимо разделить еще вектор b и вектор результата c. Элементы векторов можно *продублировать*, то есть скопировать все элементы вектора на все процессоры, составляющие многопроцессорную вычислительную систему, или разделить между процессорами. При *блочном* разбиении вектора из n элементов каждый процессор обрабатывает непрерывную последовательность из k элементов вектора (мы предполагаем, что размерность вектора n нацело делится на число процессоров, т.е. $n = k \cdot p$).

Поясним, почему дублирование векторов b и c между процессорами является допустимым решением (далее для простоты изложения будем полагать, что $m=n$). Векторы b и c состоят из n элементов, т. е. содержат столько же данных, сколько и одна строка или один столбец матрицы. Если процессор хранит строку или столбец матрицы и одиночные элементы векторов b и c, то общее число сохраняемых элементов имеет порядок $O(n)$. Если процессор хранит строку (столбец) матрицы и все элементы векторов b и c, то общее число сохраняемых элементов также порядка $O(n)$. Таким образом, при дублировании и при разделении векторов требования к объему памяти из одного класса сложности.

6.5. Умножение матрицы на вектор при разделении данных по строкам

Рассмотрим в качестве первого примера организации параллельных матричных вычислений алгоритм умножения матрицы на вектор, основанный на представлении матрицы непрерывными наборами (горизон-

тальными полосами) строк. При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция скалярного умножения одной строки матрицы на вектор.

6.5.1. Выделение информационных зависимостей

Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы A и копию вектора b. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата c.

Для объединения результатов расчета и получения полного вектора c на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных (см. лекцию 4), в которой каждый процессор передает свой вычисленный элемент вектора c всем остальным процессорам. Этот шаг можно выполнить, например, с использованием функции `MPI_Allgather` из библиотеки MPI.

В общем виде схема информационного взаимодействия подзадач в ходе выполняемых вычислений показана на рис. 6.2.

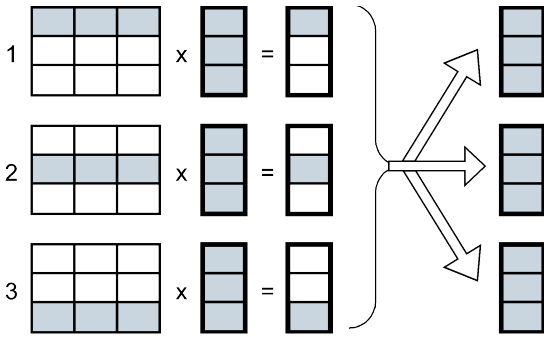


Рис. 6.2. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор, основанного на разделении матрицы по строкам

6.5.2. Масштабирование и распределение подзадач по процессорам

В процессе умножения плотной матрицы на вектор количество вычислительных операций для получения скалярного произведения одинаково для всех базовых подзадач. Поэтому в случае когда число процессоров p меньше числа базовых подзадач m, мы можем объединить базовые подзадачи таким образом, чтобы каждый процессор выполнял несколько

таких задач, соответствующих непрерывной последовательности строк матрицы A. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора c.

Распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

6.5.3. Анализ эффективности

Для анализа эффективности параллельных вычислений здесь и далее будут строиться два типа оценок. В первой из них трудоемкость алгоритмов оценивается в количестве вычислительных операций, необходимых для решения поставленной задачи, без учета затрат времени на передачу данных между процессорами, а длительность всех вычислительных операций считается одинаковой. Кроме того, константы в получаемых соотношениях, как правило, не указываются — для первого типа оценок важен прежде всего порядок сложности алгоритма, а не точное выражение времени выполнения вычислений. Как результат, в большинстве случаев подобные оценки получаются достаточно простыми и могут быть использованы для начального анализа эффективности разрабатываемых алгоритмов и методов.

Второй тип оценок направлен на формирование как можно более точных соотношений для предсказания времени выполнения алгоритмов. Получение таких оценок проводится, как правило, при помощи уточнения выражений, полученных на первом этапе. Для этого в имеющиеся соотношения вводятся параметры, задающие длительность выполнения операций, строятся оценки трудоемкости коммуникационных операций, указываются все необходимые константы. Точность получаемых выражений проверяется при помощи вычислительных экспериментов, по результатам которых время выполненных расчетов сравнивается с теоретически предсказанными оценками длительностей вычислений. Как результат, оценки подобного типа имеют, как правило, более сложный вид, но позволяют более точно оценивать эффективность разрабатываемых методов параллельных вычислений.

Рассмотрим трудоемкость алгоритма умножения матрицы на вектор. В случае если матрица A квадратная ($m=n$), последовательный алгоритм умножения матрицы на вектор имеет сложность $T_1=n^2$. В случае параллельных вычислений каждый процессор производит умножение только части (полосы) матрицы A на вектор b, размер этих полос равен n/p строк. При вычислении скалярного произведения одной строки матрицы и вектора необходимо произвести n операций умножения и $(n\text{-}1)$ операций сложения. Следовательно, вычислительная трудоемкость параллельного алгоритма определяется выражением:

$$T_p = n^2/p. \qquad (6.5)$$

С учетом этой оценки показатели ускорения и эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n^2}{n^2/p} = p\ ,\quad E_p = \frac{n^2}{p\cdot(n^2/p)} = 1. \qquad (6.6)$$

Построенные выше оценки времени вычислений выражены в количестве операций и, кроме того, определены без учета затрат на выполнение операций передачи данных. Используем ранее высказанные предположения о том, что выполняемые операции умножения и сложения имеют одинаковую длительность τ. Кроме того, будем предполагать также, что вычислительная система является однородной, т.е. все процессоры, составляющие эту систему, обладают одинаковой производительностью. С учетом введенных предположений время выполнения параллельного алгоритма, связанное непосредственно с вычислениями, составляет

$$T_p(calc) = \lceil n/p \rceil \cdot (2n-1)\cdot\tau$$

(здесь и далее операция $\lceil\ \rceil$ есть округление до целого в большую сторону).

Оценка трудоемкости операции обобщенного сбора данных уже выполнялась в лекции 4 (см. п. 4.3.4). Как уже отмечалась ранее, данная операция может быть выполнена за $\lceil \log_2 p \rceil$ итераций[1]. На первой итерации взаимодействующие пары процессоров обмениваются сообщениями объемом $w\lceil n/p \rceil$ (w есть размер одного элемента вектора c в байтах), на второй итерации этот объем увеличивается вдвое и оказывается равным $2w\lceil n/p \rceil$ и т.д. Как результат, длительность выполнения операции сбора данных при использовании модели Хокни может быть определена при помощи следующего выражения

$$T_p(comm) = \sum_{i=1}^{\lceil \log_2 p \rceil} (\alpha + 2^{i-1} w \lceil n/p \rceil / \beta) = \alpha \lceil \log_2 p \rceil + w \lceil n/p \rceil (2^{\lceil \log_2 p \rceil} - 1)/\beta, \qquad (6.7)$$

где α – латентность сети передачи данных, β – пропускная способность сети. Таким образом, общее время выполнения параллельного алгоритма составляет

$$T_p = (n/p)\cdot(2n-1)\cdot\tau + \alpha\cdot\log_2 p + w(n/p)(p-1)/\beta \qquad (6.8)$$

[1] Будем предполагать, что топология вычислительной системы допускает возможность такого эффективного способа выполнения операции обобщенного сбора данных (это возможно, в частности, если структура сети передачи данных имеет вид гиперкуба или полного графа).

(для упрощения выражения в (6.8) предполагалось, что значения n/p и $\log_2 p$ являются целыми).

6.5.4. Программная реализация

Представим возможный вариант параллельной программы умножения матрицы на вектор с использованием алгоритма разбиения матрицы по строкам. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияние на понимании общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 6.1
// Умножение матрицы на вектор – ленточное горизонтальное разбиение
// (исходный и результирующий векторы дублируются между процессами)
void main(int argc, char* argv[]) {
  double* pMatrix;  // Первый аргумент – исходная матрица
  double* pVector;  // Второй аргумент – исходный вектор
  double* pResult;  // Результат умножения матрицы на вектор
  int Size;         // Размеры исходных матрицы и вектора
  double* pProcRows;
  double* pProcResult;
  int RowNum;
  double Start, Finish, Duration;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  // Выделение памяти и инициализация исходных данных
  ProcessInitialization(pMatrix, pVector, pResult, pProcRows,
    pProcResult, Size, RowNum);

  // Распределение исходных данных между процессами
  DataDistribution(pMatrix, pProcRows, pVector, Size, RowNum);

  // Параллельное выполнение умножения матрицы на вектор
  ParallelResultCalculation(pProcRows, pVector, pProcResult, Size,
    RowNum);

  // Сбор результирующего вектора на всех процессах
```

```
  ResultReplication(pProcResult, pResult, Size, RowNum);

  // Завершение процесса вычислений
  ProcessTermination(pMatrix, pVector, pResult, pProcRows,
    pProcResult);

  MPI_Finalize();
}
```

2. Функция ProcessInitialization. Эта функция задает размер и элементы для матрицы A и вектора b. Значения для матрицы A и вектора b определяются в функции `RandomDataInitialization`.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pMatrix, double* &pVector,
  double* &pResult, double* &pProcRows, double* &pProcResult,
  int &Size, int &RowNum) {
  int RestRows;   // Количество строк матрицы, которые еще
                  // не распределены
  int i;

  if (ProcRank == 0) {
    do {
      printf("\nВведите размер матрицы: ");
      scanf("%d", &Size);
      if (Size < ProcNum) {
        printf("Размер матрицы должен превышать количество
        процессов! \n ");
      }
    }
    while (Size < ProcNum);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  RestRows = Size;
  for (i=0; i<ProcRank; i++)
    RestRows = RestRows-RestRows/(ProcNum-i);
  RowNum = RestRows/(ProcNum-ProcRank);

  pVector = new double [Size];
  pResult = new double [Size];
  pProcRows = new double [RowNum*Size];
```

```
  pProcResult = new double [RowNum];

  if (ProcRank == 0) {
    pMatrix = new double [Size*Size];
    RandomDataInitialization(pMatrix, pVector, Size);
  }
}
```

3. Функция DataDistribution. Осуществляет рассылку вектора b и распределение строк исходной матрицы A по процессам вычислительной системы. Следует отметить, что когда количество строк матрицы n не является кратным числу процессоров p, объем пересылаемых данных для процессов может оказаться разным и для передачи сообщений необходимо использовать функцию `MPI_Scatterv` библиотеки MPI.

```
// Функция для распределения исходных данных между процессами
void DataDistribution(double* pMatrix, double* pProcRows,
  double* pVector, int Size, int RowNum) {
  int *pSendNum;     // Количество элементов, посылаемых процессу
  int *pSendInd;     // Индекс первого элемента данных,
                     // посылаемого процессу
  int RestRows=Size; // Количество строк матрицы, которые еще
                     // не распределены

  MPI_Bcast(pVector, Size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Выделение памяти для хранения временных объектов
  pSendInd = new int [ProcNum];
  pSendNum = new int [ProcNum];

  // Определение положения строк матрицы, предназначенных
  // каждому процессу
  RowNum = (Size/ProcNum);
  pSendNum[0] = RowNum*Size;
  pSendInd[0] = 0;
  for (int i=1; i<ProcNum; i++) {
    RestRows -= RowNum;
    RowNum = RestRows/(ProcNum-i);
    pSendNum[i] = RowNum*Size;
    pSendInd[i] = pSendInd[i-1]+pSendNum[i-1];
  }
```

```
  // Рассылка строк матрицы
  MPI_Scatterv(pMatrix, pSendNum, pSendInd, MPI_DOUBLE, pProcRows,
    pSendNum[ProcRank], MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // Освобождение памяти
  delete [] pSendNum;
  delete [] pSendInd;
}
```

Следует отметить, что такое разделение действий генерации исходных данных и их рассылки между процессами может быть неоправданным в реальных параллельных вычислениях при большом объеме данных. Широко используемый подход в таких случаях состоит в организации передачи сообщений процессам сразу же после того, как данные процессов будут сгенерированы. Снижение затрат памяти для хранения данных может быть достигнуто также и за счет организации генерации данных в последнем процессе (при таком подходе память для пересылаемых данных и для данных процесса может быть одной и той же).

4. Функция ParallelResultCaculation. Данная функция производит умножение на вектор тех строк матрицы, которые распределены на данный процесс, и таким образом получается блок результирующего вектора *c*.

```
// Функция для вычисления части результирующего вектора
void ParallelResultCalculation(double* pProcRows, double* pVector,
  double* pProcResult, int Size, int RowNum) {
  int i, j;
  for (i=0; i<RowNum; i++) {
    pProcResult[i] = 0;
    for (j=0; j<Size; j++)
      pProcResult[i] += pProcRows[i*Size+j]*pVector[j];
  }
}
```

5. Функция ResultReplication. Объединяет блоки результирующего вектора *c*, полученные на разных процессах, и копирует вектор результата на все процессы вычислительной системы.

```
// Функция для сбора результирующего вектора на всех процессах
void ResultReplication(double* pProcResult, double* pResult,
  int Size, int RowNum) {
  int *pReceiveNum;  // Количество элементов, посылаемых процессом
```

```
  int *pReceiveInd;  // Индекс элемента данных в результирующем
                     // векторе
  int RestRows=Size; // Количество строк матрицы, которые еще не
                     // распределены
  int i;

  // Выделение памяти для временных объектов
  pReceiveNum = new int [ProcNum];
  pReceiveInd = new int [ProcNum];

  // Определение положения блоков результирующего вектора
  pReceiveInd[0] = 0;
  pReceiveNum[0] = Size/ProcNum;
  for (i=1; i<ProcNum; i++) {
    RestRows -= pReceiveNum[i-1];
    pReceiveNum[i] = RestRows/(ProcNum-i);
    pReceiveInd[i] = pReceiveInd[i-1]+pReceiveNum[i-1];
  }
  // Сбор всего результирующего вектора на всех процессах
  MPI_Allgatherv(pProcResult, pReceiveNum[ProcRank],
    MPI_DOUBLE, pResult, pReceiveNum, pReceiveInd,
    MPI_DOUBLE, MPI_COMM_WORLD);

  // Освобождение памяти
  delete [] pReceiveNum;
  delete [] pReceiveInd;
}
```

6.5.5. Результаты вычислительных экспериментов

Рассмотрим результаты вычислительных экспериментов, выполненных для оценки эффективности приведенного выше параллельного алгоритма умножения матрицы на вектор. Кроме того, используем полученные результаты для сравнения теоретических оценок и экспериментальных показателей времени вычислений и проверим тем самым точность полученных аналитических соотношений. Эксперименты проводились на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server (см. п. 1.2.3).

Определение параметров теоретических зависимостей (величин τ, w, α, β) осуществлялось следующим образом. Для оценки длительности τ ба-

зовой скалярной операции проводилось решение задачи умножения матрицы на вектор при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 1,93 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 47 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина w равна 8 байт.

Результаты вычислительных экспериментов приведены в таблице 6.1. Эксперименты проводились с использованием двух, четырех и восьми процессоров. Времена выполнения алгоритмов указаны в секундах.

Таблица 6.1. Результаты вычислительных экспериментов для параллельного алгоритма умножения матрицы на вектор при ленточной схеме разделения данных по строкам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,0041	0,0021	1,8798	0,0017	2,4089	0,0175	0,2333
2000	0,016	0,0084	1,8843	0,0047	3,3388	0,0032	4,9443
3000	0,031	0,0185	1,6700	0,0097	3,1778	0,0059	5,1952
4000	0,062	0,0381	1,6263	0,0188	3,2838	0,0244	2,5329
5000	0,11	0,0574	1,9156	0,0314	3,4993	0,0150	7,3216

Сравнение экспериментального времени T_p' выполнения параллельного алгоритма и теоретического времени T_p, вычисленного в соответствии с выражением (6.8), представлено в таблице 6.2 и в графическом виде на рис. 6.3 и 6.4.

Таблица 6.2. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по строкам

Размер объектов	2 процессора		4 процессора		8 процессоров	
	T_p	T_p'	T_p	T_p'	T_p	T_p'
1000	0,0069	0,0021	0,0108	0,0017	0,0152	0,0175
2000	0,0132	0,0084	0,0140	0,0047	0,0169	0,0032
3000	0,0235	0,0185	0,0193	0,0097	0,0196	0,0059
4000	0,0379	0,0381	0,0265	0,0188	0,0233	0,0244
5000	0,0565	0,0574	0,0359	0,0314	0,0280	0,0150

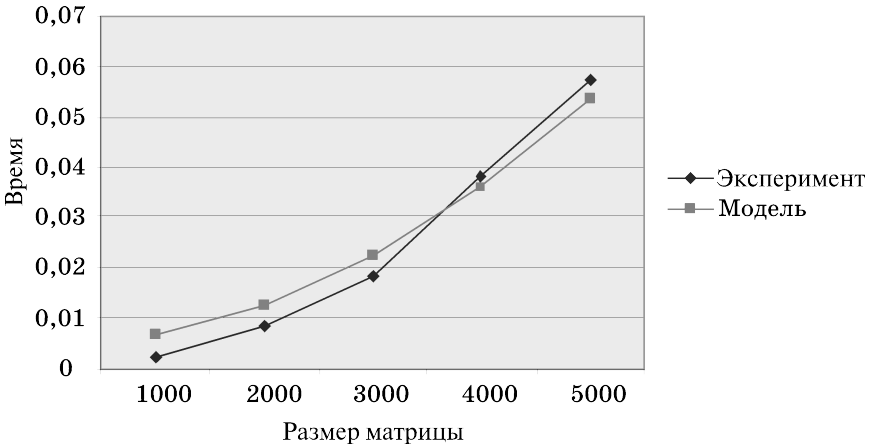


Рис. 6.3. График зависимости экспериментального T_p' и теоретического T_p времени выполнения параллельного алгоритма на двух процессорах от объема исходных данных (ленточное разбиение матрицы по строкам)

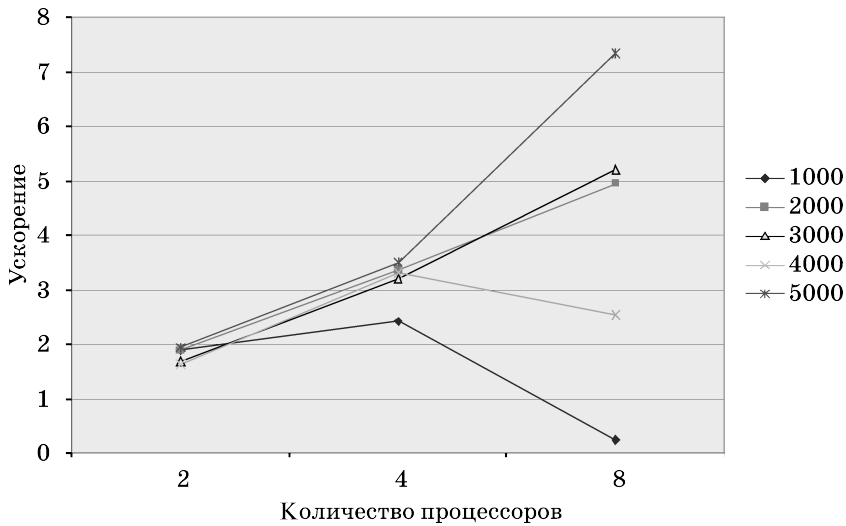


Рис. 6.4. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение по строкам) для разных размеров матриц

6.6. Умножение матрицы на вектор при разделении данных по столбцам

Рассмотрим теперь другой подход к параллельному умножению матрицы на вектор, основанный на разделении исходной матрицы на непрерывные наборы (вертикальные полосы) столбцов.

6.6.1. Определение подзадач и выделение информационных зависимостей

При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция умножения столбца матрицы A на один из элементов вектора b. Для организации вычислений в этом случае каждая базовая подзадача i, $0 \le i < n$, должна содержать i-й столбец матрицы A и i-е элементы b_i и c_i векторов b и c.

Параллельный алгоритм умножения матрицы на вектор начинается с того, что каждая базовая задача i выполняет умножение своего столбца матрицы A на элемент b_i, в итоге в каждой подзадаче получается вектор $c'(i)$ промежуточных результатов. Далее для получения элементов результирующего вектора c подзадачи должны обменяться своими промежуточными данными между собой (элемент j, $0 \le j < n$, частичного результата $c'(i)$ подзадачи i, $0 \le i < n$, должен быть передан подзадаче j). Данная обобщенная передача данных (*all-to-all communication* или *total exchange*) является наиболее общей коммуникационной процедурой и может быть реализована при помощи функции `MPI_Alltoall` библиотеки MPI. После выполнения передачи данных каждая базовая подзадача i, $0 \le i < n$, будет содержать n частичных значений $c'_i(j)$, $0 \le j < n$, сложением которых и определяется элемент c_i вектора результата c (см. рис. 6.5).

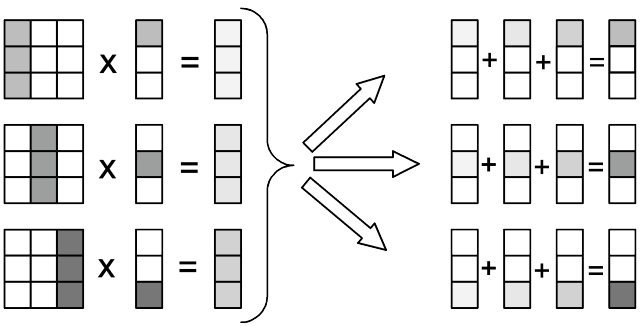


Рис. 6.5. Организация вычислений при выполнении параллельного алгоритма умножения матрицы на вектор с использованием разбиения матрицы по столбцам

6.6.2. Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае когда количество столбцов матрицы превышает число процессоров, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних столбцов (в этом случае исходная матрица A разбивается на ряд вертикальных полос). При соблюдении равенства размера полос такой способ агрегации вычислений обеспечивает равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

Как и в предыдущем алгоритме, распределение подзадач между процессорами вычислительной системы может быть выполнено произвольным образом.

6.6.3. Анализ эффективности

Пусть, как и ранее, матрица A является квадратной, то есть $m=n$. На первом этапе вычислений каждый процессор умножает принадлежащие ему столбцы матрицы A на элементы вектора b, после умножения полученные значения суммируются для каждой строки матрицы A в отдельности

$$c'_s(i) = \sum_{j=j_0}^{j_{l-1}} a_{sj} b_j, \; 0 \le s < n, \tag{6.9}$$

(j_0 и j_{l-1} есть начальный и конечный индексы столбцов базовой подзадачи i, $0 \le i < n$). Поскольку размеры полосы матрицы A и блока вектора b равны n/p, то трудоемкость таких вычислений может оцениваться как $T'=n^2/p$ операций. После обмена данными между подзадачами на втором этапе вычислений каждый процессор суммирует полученные значения для своего блока результирующего вектора c. Количество суммируемых значений для каждого элемента c_i вектора c совпадает с числом процессоров p, размер блока вектора c на одном процессоре равен n/p, и, тем самым, число выполняемых операций для второго этапа оказывается равным $T''=n$. С учетом полученных соотношений показатели ускорения и эффективности параллельного алгоритма могут быть выражены следующим образом:

$$S_p = \frac{n^2}{n^2/p} = p \;, \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1. \tag{6.10}$$

Теперь рассмотрим более точные соотношения для оценки времени выполнения параллельного алгоритма. С учетом ранее проведенных рас-

суждений время выполнения вычислительных операций алгоритма может быть оценено при помощи выражения

$$T_p(calc) = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau \qquad (6.11)$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Для выполнения операции обобщенной передачи данных рассмотрим два возможных способа реализации (см. также лекцию 3). Первый способ обеспечивается алгоритмом, согласно которому каждый процессор последовательно передает свои данные всем остальным процессорам вычислительном системы. Предположим, что процессоры могут одновременно отправлять и принимать сообщения и между любой парой процессоров имеется прямая линия связи, тогда оценка трудоемкости (время исполнения) такого алгоритма обобщенной передачи данных может быть определена как

$$T_p^1(comm) = (p-1)(\alpha + w\lceil n/p \rceil / \beta) \qquad (6.12)$$

(напомним, что α – латентность сети передачи данных, β – пропускная способность сети, w – размер элемента данных в байтах).

Второй способ выполнения операции обмена данными рассмотрен в лекции 3, когда топология вычислительной сети может быть представлена в виде гиперкуба. Как было показано, выполнение такого алгоритма может быть осуществлено за $\lceil \log_2 p \rceil$ шагов, на каждом из которых каждый процессор передает и получает сообщение из $n/2$ элементов. Как результат, время операции передачи данных при таком подходе составляет величину:

$$T_p^2(comm) = \lceil \log_2 p \rceil (\alpha \cdot + w(n/2)/\beta). \qquad (6.13)$$

С учетом (6.11) – (6.13) общее время выполнения параллельного алгоритма умножения матрицы на вектор при разбиении данных по столбцам выражается следующими соотношениями.

- Для первого способа выполнения операции передачи данных

$$T_p^1 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + (p-1)(\alpha + w\lceil n/p \rceil / \beta). \qquad (6.14)$$

- Для второго способа выполнения операции передачи данных

$$T_p^2 = [n \cdot (2 \cdot \lceil n/p \rceil - 1) + n] \cdot \tau + \lceil \log_2 p \rceil (\alpha \cdot + w(n/2)/\beta). \qquad (6.15)$$

6.6.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма умножения матрицы на вектор при разбиении данных по столбцам проводились при условиях, указанных в п. 6.5.5. Результаты вычислительных экспериментов приведены в таблице 6.3.

Таблица 6.3. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,0041	0,0022	1,8352	0,0013	3,1538	0,0008	4,9409
2000	0,016	0,0085	1,8799	0,0046	3,4246	0,0029	5,4682
3000	0,031	0,019	1,6315	0,0095	3,2413	0,0055	5,5456
4000	0,062	0,0331	1,8679	0,0168	3,6714	0,0090	6,8599
5000	0,11	0,0518	2,1228	0,0265	4,1361	0,0136	8,0580

Сравнение экспериментального времени T_p^* выполнения эксперимента и времени T_p, вычисленного по соотношениям (6.14), (6.15), представлено в таблице 6.4 и на рис. 6.6 и 6.7. Теоретическое время T_p^1 вычисляется согласно (6.14), а теоретическое время T_p^2 – в соответствии с (6.15).

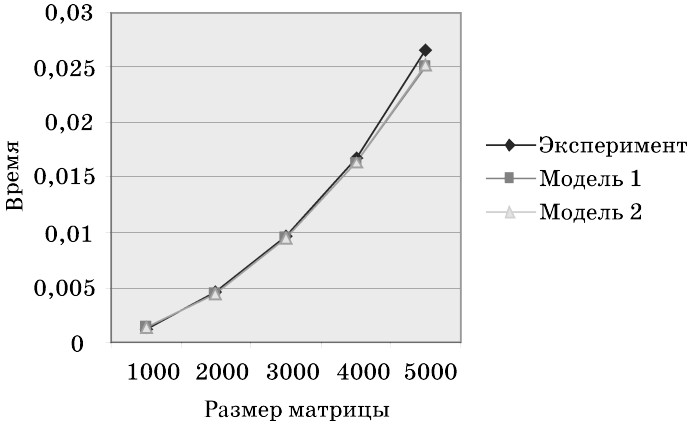


Рис. 6.6. График зависимости теоретического и экспериментального времени выполнения параллельного алгоритма на четырех процессорах от объема исходных данных (ленточное разбиение матрицы по столбцам)

Таблица 6.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор, основанного на разбиении матрицы по столбцам

Размер матрицы	2 процессора			4 процессора			8 процессоров		
	T_p^*	T_p^1	T_p^2	T_p^*	T_p^1	T_p^2	T_p^*	T_p^1	T_p^2
1000	0,0022	0,0021	0,0021	0,0013	0,0013	0,0014	0,0008	0,0011	0,0015
2000	0,0085	0,0080	0,0080	0,0046	0,0044	0,0044	0,0029	0,0027	0,0031
3000	0,019	0,0177	0,0177	0,0095	0,0094	0,0094	0,0055	0,0054	0,0056
4000	0,0331	0,0313	0,0313	0,0168	0,0163	0,0162	0,0090	0,0090	0,0091
5000	0,0518	0,0487	0,0487	0,0265	0,0251	0,0251	0,0136	0,0135	0,0136

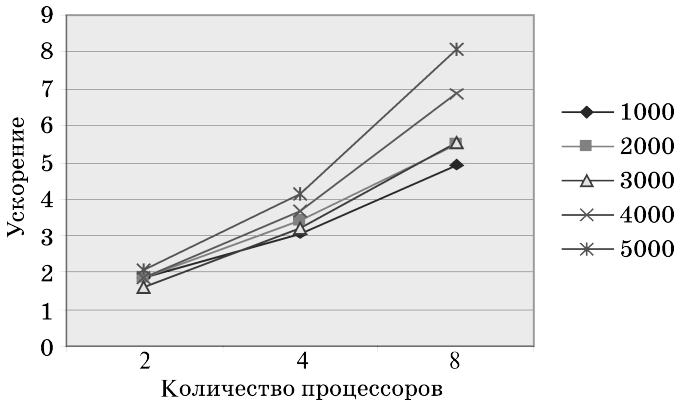


Рис. 6.7. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (ленточное разбиение матрицы по столбцам) для разных размеров матриц

6.7. Умножение матрицы на вектор при блочном разделении данных

Рассмотрим теперь параллельный алгоритм умножения матрицы на вектор, который основан на ином способе разделения данных – на разбиении матрицы на прямоугольные фрагменты (*блоки*).

6.7.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 6.1. При таком способе разделения данных исходная матрица A представляется в виде набора прямоугольных блоков:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots A_{0q-1} \\ & \dots & \\ A_{s-11} & A_{s-12} & \dots A_{s-1q-1} \end{pmatrix},$$

где A_{ij}, $0 \le i < s$, $0 \le j < q$, есть блок матрицы:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots a_{i_0 j_{l-1}} \\ & \dots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix}, \quad \begin{matrix} i_v = ik + v, 0 \le v < k, k = m/s, \\ j_u = jl + u, 0 \le u \le l, l = n/q \end{matrix}$$

(здесь, как и ранее, предполагается, что $p=s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m=k \cdot s$ и $n=l \cdot q$).

При использовании блочного представления матрицы A базовые подзадачи целесообразно определить на основе вычислений, выполняемых над матричными блоками. Для нумерации подзадач могут применяться индексы располагаемых в подзадачах блоков матрицы A, т.е. подзадача (i,j) содержит блок A_{ij}. Помимо блока матрицы A каждая подзадача должна содержать также и блок вектора b. При этом для блоков одной и той же подзадачи должны соблюдаться определенные правила соответствия – операция умножения блока матрицы A_{ij} может быть выполнена только, если блок вектора $b'(i,j)$ имеет вид

$$b'(i,j) = (b'_0(i,j), \dots, b'_{l-1}(i,j)), \text{ где } b'_u(i,j) = b_{j_u}, j_u = jl + u, 0 \le u < l, l = n/q.$$

6.7.2. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений для операции умножения матрицы на вектор при блочном разделении исходных данных. После перемножения блоков матрицы A и вектора b каждая подзадача (i,j) будет содержать вектор частичных результатов $c'(i,j)$, определяемый в соответствии с выражениями

$$c'_v(i,j) = \sum_{u=0}^{l-1} a_{i_v j_u} b_{j_u}, i_v = ik + v, 0 \le v < k, k = m/s, \quad j_u = jl + u, 0 \le u \le l, l = n/q.$$

Поэлементное суммирование векторов частичных результатов для каждой горизонтальной полосы (данная процедура часто именуется *операцией редукции* – см. лекцию 3) блоков матрицы A позволяет получить результирующий вектор c

$$c_\eta = \sum_{j=0}^{\bar{q}-1} c'_\nu\ (i, j),\ 0 \le \eta < m,\ i = \eta / s,\ \nu = \eta - i \cdot s.$$

Для размещения вектора *c* применим ту же схему, что и для исходного вектора *b*: организуем вычисления таким образом, чтобы при завершении расчетов вектор *c* располагался поблочно в каждой из вертикальных полос блоков матрицы *A* (тем самым, каждый блок вектора *c* должен быть продублирован по каждой горизонтальной полосе). Выполнение всех необходимых действий для этого – суммирование частичных результатов и дублирование блоков результирующего вектора — может быть обеспечено при помощи функции `MPI_Allreduce` библиотеки MPI.

Общая схема выполняемых вычислений для умножения матрицы на вектор при блочном разделении данных показана на рис. 6.8.

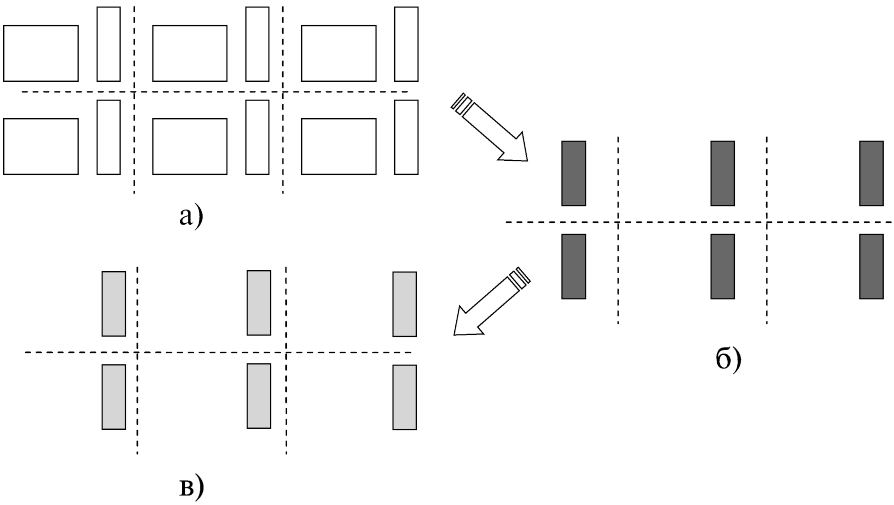


Рис. 6.8. Общая схема параллельного алгоритма умножения матрицы на вектор при блочном разделении данных: а) исходное распределение результатов, б) распределение векторов частичных результатов, в) распределение блоков результирующего вектора *c*

Рассмотрев представленную схему параллельных вычислений, можно сделать вывод, что информационная зависимость базовых подзадач проявляется только на этапе суммирования результатов перемножения блоков матрицы *A* и блоков вектора *b*. Выполнение таких расчетов может быть выполнено по обычной каскадной схеме (см. лекцию 3), и, как результат, характер имеющихся информационных связей для подзадач од-

ной и той же горизонтальной полосы блоков соответствует структуре двоичного дерева.

6.7.3. Масштабирование и распределение подзадач по процессорам

Размер блоков матрицы A может быть подобран таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров p. Так, например, если определить размер блочной решетки как $p=s \cdot q$, то

$$k=m/s,\ l=n/q,$$

где k и l есть количество строк и столбцов в блоках матрицы A. Такой способ определения размера блоков приводит к тому, что объем вычислений в каждой подзадаче является равным, и, тем самым, достигается полная балансировка вычислительной нагрузки между процессорами.

Возможность выбора остается при определении размеров блочной структуры матрицы A. Большое количество блоков по горизонтали приводит к возрастанию числа итераций в операции редукции результатов блочного умножения, а увеличение размера блочной решетки по вертикали повышает объем передаваемых данных между процессорами. Простое, часто применяемое решение состоит в использовании одинакового количества блоков по вертикали и горизонтали, т.е.

$$s = q = \sqrt{p}\ .$$

Следует отметить, что блочная схема разделения данных является обобщением всех рассмотренных в данной лекции подходов. Действительно, при $q=1$ блоки сводятся к горизонтальным полосам матрицы A, при $s=1$ исходные данные разбиваются на вертикальные полосы.

При решении вопроса распределения подзадач между процессорами должна учитываться возможность эффективного выполнения операции редукции. Возможный вариант подходящего способа распределения состоит в выделении для подзадач одной и той же горизонтальной полосы блоков множества процессоров, структура сети передачи данных между которыми имеет вид гиперкуба или полного графа.

6.7.4. Анализ эффективности

Выполним анализ эффективности параллельного алгоритма умножения матрицы на вектор при обычных уже предположениях, что матри-

ца A является квадратной, т.е. $m=n$. Будем предполагать также, что процессоры, составляющие многопроцессорную вычислительную систему, образуют прямоугольную решетку $p=s\times q$ (s – количество строк в процессорной решетке, q – количество столбцов).

Общий анализ эффективности приводит к идеальным показателям параллельного алгоритма:

$$S_p = \frac{n^2}{n^2/p} = p\,, \quad E_p = \frac{n^2}{p\cdot(n^2/p)} = 1\,. \tag{6.16}$$

Для уточнения полученных соотношений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

Общее время умножения блоков матрицы A и вектора b может быть определено как

$$T_p(calc) = \lceil n/s \rceil \cdot (2\cdot \lceil n/q \rceil - 1)\cdot \tau\,. \tag{6.17}$$

Операция редукции данных может быть выполнена с использованием каскадной схемы и включает, тем самым, $\log_2 q$ итераций передачи сообщений размера $w\lceil n/s \rceil$. Как результат, оценка коммуникационных затрат параллельного алгоритма при использовании модели Хокни может быть определена при помощи следующего выражения

$$T_p(comm) = (\alpha + w\lceil n/s \rceil/\beta)\lceil \log_2 q \rceil\,. \tag{6.18}$$

Таким образом, общее время выполнения параллельного алгоритма умножения матрицы на вектор при блочном разделении данных составляет

$$T_p = \lceil n/s \rceil \cdot (2\cdot \lceil n/q \rceil - 1)\cdot \tau + (\alpha + w\lceil n/s \rceil/\beta)\lceil \log_2 q \rceil\,. \tag{6.19}$$

6.7.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные расчеты (см. п. 6.5.5). Результаты экспериментов приведены в таблице 6.5. Вычисления проводились с использованием четырех и девяти процессоров.

Сравнение экспериментального времени T_p^* выполнения эксперимента и теоретического времени T_p, вычисленного в соответствии с выражением (6.19), представлено в таблице 6.6 и на рис. 6.10.

Таблица 6.5. Результаты вычислительных экспериментов по исследованию параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
1000	0,0041	0,0028	1,4260	0,0011	3,7998
2000	0,016	0,0099	1,6127	0,0042	3,7514
3000	0,031	0,0214	1,4441	0,0095	3,2614
4000	0,062	0,0381	1,6254	0,0175	3,5420
5000	0,11	0,0583	1,8860	0,0263	4,1755

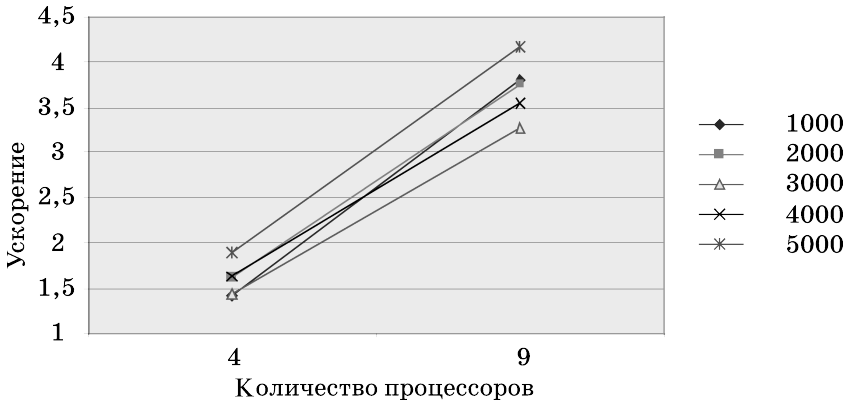


Рис. 6.9. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма умножения матрицы на вектор (блочное разбиение матрицы) для разных размеров матриц

Таблица 6.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма умножения матрицы на вектор при блочном разделении данных

Размер матриц	4 процессора		9 процессоров	
	T_p	T_p^*	T_p	T_p^*
1000	0,0025	0,0028	0,0012	0,0011
2000	0,0095	0,0099	0,0043	0,0042
3000	0,0212	0,0214	0,0095	0,0095
4000	0,0376	0,0381	0,0168	0,0175
5000	0,0586	0,0583	0,0262	0,0263

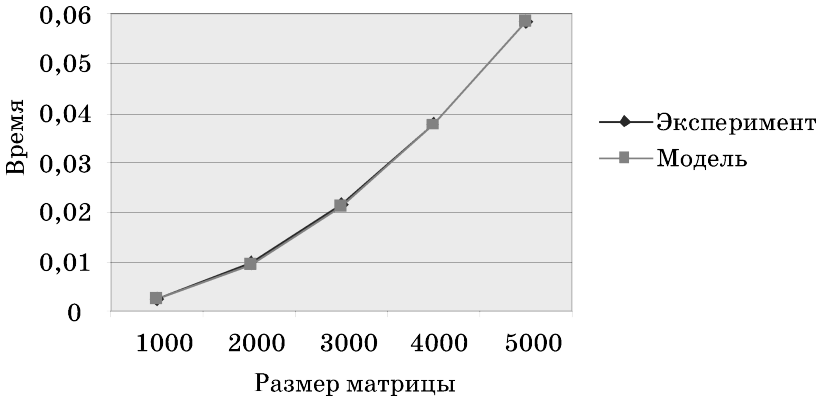


Рис. 6.10. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных (блочное разбиение матрицы)

6.8. Краткий обзор лекции

В данной лекции на примере задачи умножения матрицы на вектор рассматриваются возможные схемы разделения матриц между процессорами многопроцессорной вычислительной системы, которые могут быть использованы для организации параллельных вычислений при выполнении матричных операций. В числе излагаемых схем способы разбиения матриц на *полосы* (по вертикали или горизонтали) или на прямоугольные наборы элементов (*блоки*).

Далее в лекции с использованием рассмотренных способов разделения матриц подробно излагаются три возможных варианта параллельного выполнения операции умножения матрицы на вектор. Первый алгоритм основан на разделении матрицы между процессорами по строкам, второй – на разделении матрицы по столбцам, а третий – на блочном разделении данных. Каждый алгоритм представлен в соответствии с общей схемой, описанной в лекции 4, — вначале определяются базовые подзадачи, затем выделяются информационные зависимости подзадач, далее обсуждается масштабирование и распределение подзадач между процессорами. В завершение для каждого алгоритма проводится анализ эффективности получаемых параллельных вычислений и приводятся результаты вычислительных экспериментов. Для алгоритма умножения матрицы на вектор при ленточном разделении данных по строкам приводится возможный вариант программной реализации.

Полученные показатели эффективности показывают, что все используемые способы разделения данных приводят к равномерной балансировке вычислительной нагрузки, и отличия имеются только в трудоемкости выполняемых информационных взаимодействий между процессорами. В этом отношении представляется интересным проследить, как выбор способа разделения данных влияет на характер необходимых операций передачи данных, и выделить основные различия в коммуникационных действиях разных алгоритмов. Кроме того, важным является определение целесообразной структуры линий связи между процессорами для эффективного выполнения соответствующего параллельного алгоритма. Так, например, алгоритмы, основанные на ленточном разделении данных, ориентированы на топологию сети в виде гиперкуба или полного графа. Для реализации алгоритма, основанного на блочном разделении данных, необходимо наличие топологии решетки.

На рис. 6.11 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Следует отметить, что дополнительные расчеты показывают , что при большем количестве процессоров и при большом размере матриц более эффективным становится блочный алгоритм умножения.

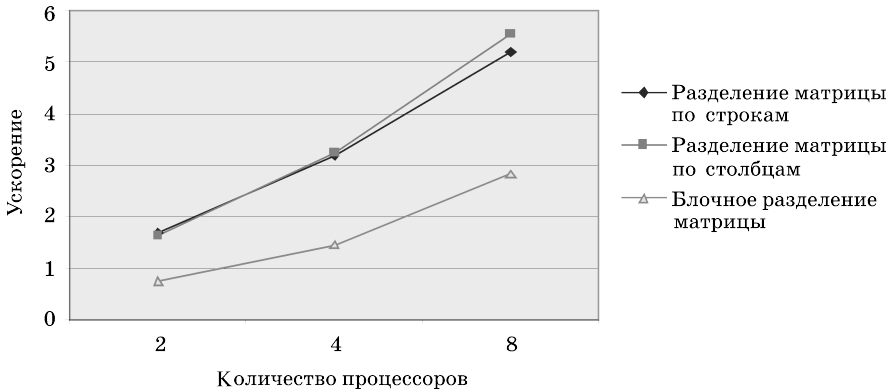


Рис. 6.11. Показатели ускорения рассмотренных параллельных алгоритмов умножения по результатам вычислительных экспериментов с матрицами размера 2000×2000 и векторами из 2000 элементов

6.9. Обзор литературы

Задача умножения матрицы на вектор часто используется как демонстрационный пример параллельного программирования и, как результат,

широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы [2, 51, 63]. Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе [30].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [23]. В ней рассматривается хорошо известная и широко применяемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

6.10. Контрольные вопросы

1. Назовите основные способы распределения элементов матрицы между процессорами вычислительной системы.
2. В чем состоит постановка задачи умножения матрицы на вектор?
3. Какова вычислительная сложность последовательного алгоритма умножения матрицы на вектор?
4. Почему при разработке параллельных алгоритмов умножения матрицы на вектор допустимо дублировать вектор-операнд на все процессоры?
5. Какие подходы могут быть предложены для разработки параллельных алгоритмов умножения матрицы на вектор?
6. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матрицы на вектор.
7. Проведите анализ и получите показатели эффективности для одного из рассмотренных алгоритмов.
8. Какой из представленных алгоритмов умножения матрицы на вектор обладает лучшими показателями ускорения и эффективности?
9. Может ли использование циклической схемы разделения данных повлиять на время работы каждого из представленных алгоритмов?
10. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных? В чем различие необходимых операций передачи данных при разделении матрицы по строкам и столбцам?
11. Какие информационные взаимодействия выполняются для блочного алгоритма умножения матрицы на вектор?
12. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
13. Дайте общую характеристику программной реализации алгоритма умножения матрицы на вектор при разделении данных по строкам. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?

14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

6.11. Задачи и упражнения

1. Выполните реализацию параллельного алгоритма, основанного на ленточном разбиении матрицы на вертикальные полосы. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.
2. Выполните реализацию параллельного алгоритма, основанного на разбиении матрицы на блоки. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее подготовленными теоретическими оценками.

Лекция 7. Параллельные методы матричного умножения

В лекции рассматривается одна из основных задач матричных вычислений — умножение матриц. Приводится постановка задачи и дается последовательный алгоритм ее решения. Далее описываются возможные подходы к параллельной реализации алгоритма и подробно рассматриваются наиболее широко известные алгоритмы: алгоритм, основанный на ленточной схеме разделения данных, алгоритм Фокса (Fox) и алгоритм Кэннона (Cannon).

Ключевые слова: умножение матриц, декомпозиция вычислений, ленточная и блочная схемы разделения данных, информационная зависимость и масштабирование вычислений, алгоритмы Фокса и Кэннона, виртуальная топология, декартова топология, вычислительная и коммуникационная сложность алгоритма, ускорение вычислений, программная реализация, операции передачи данных, MPI, вычислительный эксперимент.

7.1. Постановка задачи

Умножение матрицы A размера $m \times n$ и матрицы B размера $n \times l$ приводит к получению матрицы C размера $m \times l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l \ . \tag{7.1}$$

Как следует из (7.1), каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = (a_i, b_j^T), \ a_i = (a_{i0}, a_{i1}, ..., a_{in-1}), \ b_j^T = (b_{0j}, b_{1j}, ..., b_{n-1j})^T \ . \tag{7.2}$$

Этот алгоритм предполагает выполнение $m \cdot n \cdot l$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n \times n$ количество выполненных операций имеет порядок $O(n^3)$. Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью (например, алгоритм Страссена (*Strassen's algorithm*)), но эти алгоритмы требуют больших усилий для их освоения, и поэтому в данной лекции при разработке параллельных методов в качестве основы будет использоваться приведенный выше последовательный алгоритм. Также будем

предполагать далее, что все матрицы являются квадратными и имеют размер $n \times n$.

7.2. Последовательный алгоритм

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

Алгоритм 7.1. Последовательный алгоритм умножения двух квадратных матриц

```
// Алгоритм 7.1
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы *C*. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной *i*) вычисляется одна строка результирующей матрицы (см. рис. 7.1).

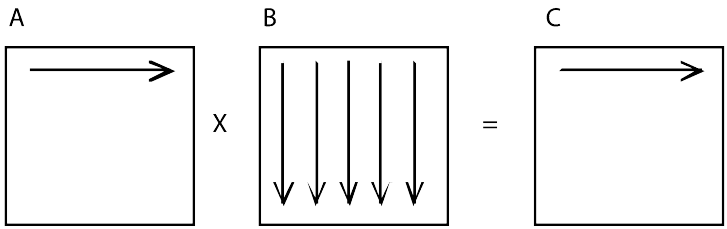


Рис. 7.1. На первой итерации цикла по переменной *i* используется первая строка матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы C размером $n \times n$ необходимо выполнить $n^2 \cdot (2n-1)$ скалярных операций и затратить время

$$T_1 = n^2 \cdot (2n-1) \cdot \tau , \qquad (7.3)$$

где τ есть время выполнения одной элементарной скалярной операции.

7.3. Умножение матриц при ленточной схеме разделения данных

Рассмотрим два параллельных алгоритма умножения матриц, в которых матрицы A и B разбиваются на непрерывные последовательности строк или столбцов (полосы).

7.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы C может быть выполнено независимо друг от друга. Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы C. Для проведения всех необходимых вычислений каждая подзадача должна содержать по одной строке матрицы A и одному столбцу матрицы B. Общее количество получаемых при таком подходе подзадач оказывается равным n^2 (по числу элементов матрицы C).

Рассмотрев предложенный подход, можно отметить, что достигнутый уровень параллелизма является в большинстве случаев избыточным. Обычно при проведении практических расчетов такое количество сформированных подзадач превышает число имеющихся процессоров и делает неизбежным этап укрупнения базовых задач. В этом плане может оказаться полезной агрегация вычислений уже на шаге выделения базовых подзадач. Возможное решение может состоять в объединении в рамках одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы C. Для дальнейшего рассмотрения определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы C. Такой подход приводит к снижению общего количества подзадач до величины n.

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы A и все столбцы матрицы B. Простое решение этой проблемы – дублирование матрицы B во всех подзадачах – является, как правило, неприемлемым в силу больших за-

трат памяти для хранения данных. Поэтому организация вычислений должна быть построена таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть данных, необходимых для проведения расчетов, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Два возможных способа выполнения параллельных вычислений подобного типа рассмотрены далее в п. 7.3.2.

7.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы C необходимо, чтобы в каждой подзадаче содержалась строка матрицы A и был обеспечен доступ ко всем столбцам матрицы B. Возможные способы организации параллельных вычислений состоят в следующем.

1. Первый алгоритм. Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач. На каждой итерации алгоритма 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. По завершении вычислений в конце каждой итерации столбцы матрицы B должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы B и могли быть вычислены новые элементы матрицы C. При этом данная передача столбцов между подзадачами должна быть организована таким образом, чтобы после завершения итераций алгоритма в каждой подзадаче последовательно оказались все столбцы матрицы B.

Возможная простая схема организации необходимой последовательности передач столбцов матрицы B между подзадачами состоит в представлении топологии информационных связей подзадач в виде кольцевой структуры. В этом случае на каждой итерации подзадача i, $0 \le i < n$, будет передавать свой столбец матрицы B подзадаче с номером $i+1$ (в соответствии с кольцевой структурой подзадача $n-1$ передает свои данные подзадаче с номером 0) – см. рис. 7.2. После выполнения всех итераций алгоритма необходимое условие будет обеспечено – в каждой подзадаче поочередно окажутся все столбцы матрицы B.

На рис. 7.2 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов ($n=4$). В начале вычислений в каждой подзадаче i, $0 \le i < n$, располагаются i-я строка матрицы A и i-й столбец матрицы B. В результате их перемножения подзадача получает элемент c_{ii} результирующей матрицы C.

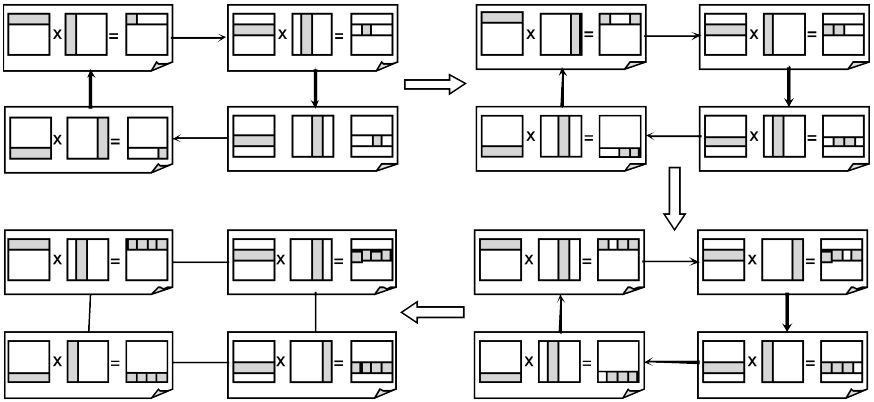


Рис. 7.2. Общая схема передачи данных для первого параллельного алгоритма матричного умножения при ленточной схеме разделения данных

Далее подзадачи осуществляют обмен столбцами, в ходе которого каждая подзадача передает свой столбец матрицы *B* следующей подзадаче в соответствии с кольцевой структурой информационных взаимодействий. Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

2. Второй алгоритм. Отличие второго алгоритма состоит в том, что в подзадачах располагаются не столбцы, а строки матрицы *B*. Как результат, перемножение данных каждой подзадачи сводится не к скалярному умножению имеющихся векторов, а к их поэлементному умножению. В результате подобного умножения в каждой подзадаче получается строка частичных результатов для матрицы *C*.

При рассмотренном способе разделения данных для выполнения операции матричного умножения нужно обеспечить последовательное получение в подзадачах всех строк матрицы *B*, поэлементное умножение данных и суммирование вновь получаемых значений с ранее вычисленными результатами. Организация необходимой последовательности передач строк матрицы *B* между подзадачами также может быть выполнена с использованием кольцевой структуры информационных связей (см. рис. 7.3).

На рис. 7.3 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов (n=4). В начале 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

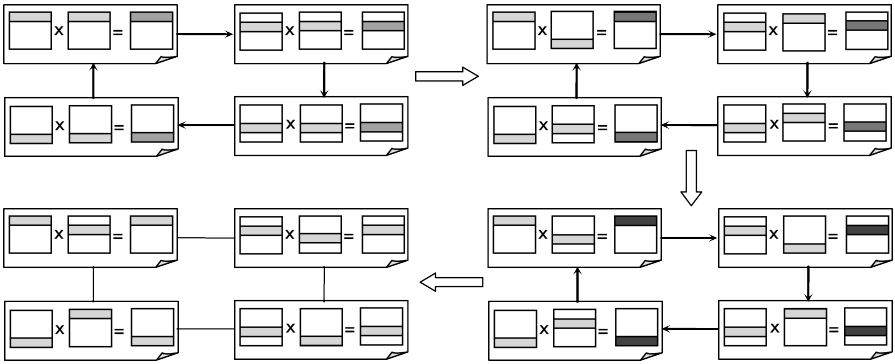


Рис. 7.3. Общая схема передачи данных для второго параллельного алгоритма матричного умножения при ленточной схеме разделения данных

каждая подзадача передает свою строку матрицы B следующей подзадаче в соответствии с кольцевой структурой информационных взаимодействий. Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

7.3.3. Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. Когда размер матриц n оказывается больше, чем число процессоров p, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк и столбцов перемножаемых матриц. В этом случае 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. Размер полос при этом следует выбрать равным $k=n/p$ (в предположении, что n кратно p), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

Для распределения подзадач между процессорами может быть использован любой способ, обеспечивающий эффективное представление кольцевой структуры информационного взаимодействия подзадач. Для этого достаточно, например, чтобы подзадачи, являющиеся соседними в кольцевой топологии, располагались на процессорах, между которыми имеются прямые линии передачи данных.

7.3.4. Анализ эффективности

Выполним анализ эффективности первого параллельного алгоритма умножения матриц.

Общая трудоемкость последовательного алгоритма, как уже отмечалось ранее, является пропорциональной n^3. Для параллельного алгоритма на каждой итерации каждый процессор выполняет умножение имеющихся на процессоре полос матрицы A и матрицы B (размер полос равен n/p, и, как результат, общее количество выполняемых при этом умножении операций равно n^3/p^2). Поскольку число итераций алгоритма совпадает с количеством процессоров, сложность параллельного алгоритма без учета затрат на передачу данных может быть определена при помощи выражения

$$T_p = (n^3 / p^2) \cdot p = n^3 / p \,. \tag{7.4}$$

С учетом этой оценки показатели ускорения и эффективности данного параллельного алгоритма матричного умножения принимают вид:

$$S_p = \frac{n^3}{(n^3 / p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3 / p)} = 1 \,. \tag{7.5}$$

Таким образом, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

С учетом числа и длительности выполняемых операций время выполнения вычислений параллельного алгоритма может быть оценено следующим образом:

$$T_p(calc) = (n^2 / p) \cdot (2n - 1) \cdot \tau \tag{7.6}$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Для оценки коммуникационной сложности параллельных вычислений будем предполагать, что все операции передачи данных между процессорами в ходе одной итерации алгоритма могут быть выполнены параллельно. Объем передаваемых данных между процессорами определяется размером полос и составляет n/p 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). Тем самым, оценка трудоемкости выполняемых операций передачи данных может быть определена как

$$T_p(comm) = (p-1)\cdot(\alpha + w\cdot n\cdot(n/p)/\beta), \quad (7.7)$$

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

С учетом полученных соотношений общее время выполнения параллельного алгоритма матричного умножения определяется следующим выражением:

$$T_p = (n^2/p)(2n-1)\cdot\tau + (p-1)\cdot(\alpha + w\cdot n\cdot(n/p)/\beta). \quad (7.8)$$

7.3.5. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном кластере на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server.

Для оценки длительности τ базовой скалярной операции проводилось решение задачи умножения матриц при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 6,4 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 130 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина w равна 8 байт.

Результаты вычислительных экспериментов приведены в таблице 7.1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров.

Таблица 7.1. Результаты вычислительных экспериментов по исследованию первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Размер матриц	Последовательный алгоритм	2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
500	0,8752	0,3758	2,3287	0,1535	5,6982	0,0968	9,0371
1000	12,8787	5,4427	2,3662	2,2628	5,6912	0,6998	18,4014
1500	43,4731	20,9503	2,0750	11,0804	3,9234	5,1766	8,3978
2000	103,0561	45,7436	2,2529	21,6001	4,7710	9,4127	10,9485
2500	201,2915	99,5097	2,0228	56,9203	3,5363	18,3303	10,9813
3000	347,8434	171,9232	2,0232	111,9642	3,1067	45,5482	7,6368

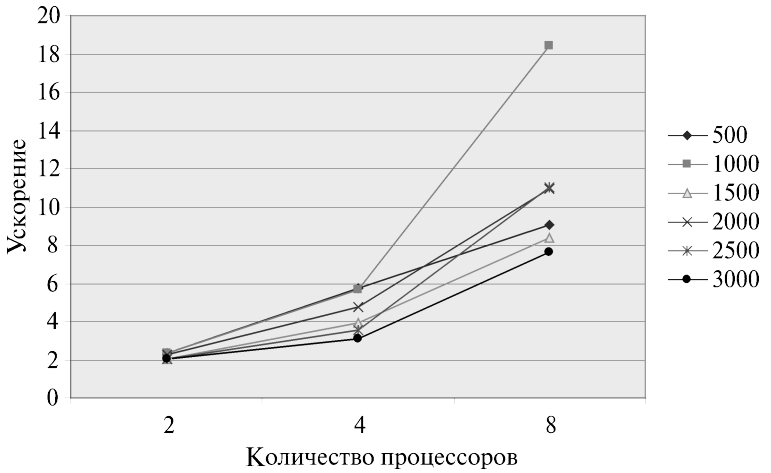


Рис. 7.4. Зависимость ускорения от количества процессоров при выполнении первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Сравнение экспериментального времени T_p^* выполнения эксперимента и теоретического времени T_p из формулы (7.8) представлено в таблице 7.2 и на рис. 7.5.

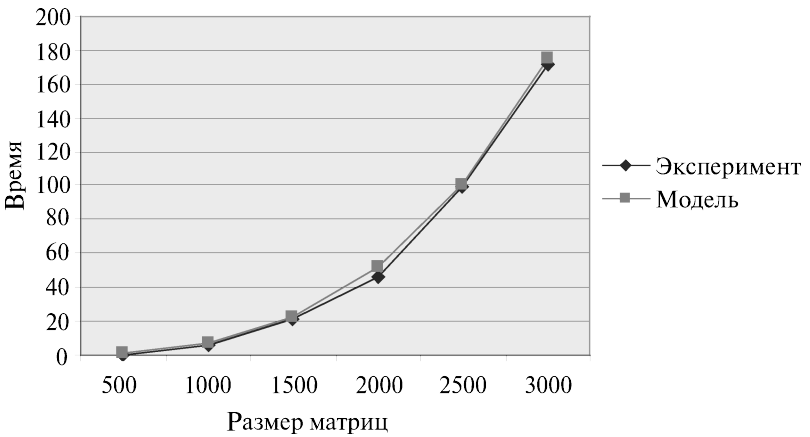


Рис. 7.5. График зависимости от объема исходных данных теоретического и экспериментального времени выполнения параллельного алгоритма на двух процессорах (ленточная схема разбиения данных)

Таблица 7.2. Сравнение экспериментального и теоретического времени выполнения первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Размер матриц	2 процессора		4 процессора		8 процессоров	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	0,8243	0,3758	0,4313	0,1535	0,2353	0,0968
1000	6,51822	5,4427	3,3349	2,2628	1,7436	0,6998
1500	21,9137	20,9503	11,1270	11,0804	5,7340	5,1766
2000	51,8429	45,7436	26,2236	21,6001	13,4144	9,4127
2500	101,1377	99,5097	51,0408	56,9203	25,9928	18,3303
3000	174,6301	171,9232	87,9946	111,9642	44,6772	45,5482

7.4. Алгоритм Фокса умножения матриц при блочном разделении данных

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Рассмотрим более подробно данный способ организации вычислений.

7.4.1. Определение подзадач

Блочная схема разбиения матриц подробно изложена в первом разделе лекции 6. При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, 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 $k \times k$, $k=n/q$). При таком представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена так:

$$\begin{pmatrix} A_{00}A_{01}\ldots A_{0q-1} \\ \cdots \\ A_{q-10}A_{q-11}\ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01}\ldots B_{0q-1} \\ \cdots \\ B_{q-10}B_{q-11}\ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01}\ldots C_{0q-1} \\ \cdots \\ c_{q-10}C_{q-11}\ldots C_{q-1q-1} \end{pmatrix},$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}.$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы *C*.

Для выполнения всех необходимых вычислений базовым подзадачам 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 используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Один из возможных подходов – алгоритм Фокса (*Fox*) – рассмотрен далее в данном подразделе. Второй способ – алгоритм Кэннона (*Cannon*) – приводится в подразделе 7.5.

7.4.2. Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы *C* и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц *A* и *B*. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы *C*, т.е. подзадача (*i*,*j*) отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы *C*.

Возможный способ организации вычислений при таких условиях состоит в применении широко известного *алгоритма Фокса* (*Fox*) — см., например, [34, 51].

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (i,j) располагается 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 метода включает:

- 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

$$j = (i+l) \bmod q,$$

где mod есть операция получения остатка от целочисленного деления;

— полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи (i,j) перемножаются и прибавляются к блоку C_{ij}

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$

— блоки B'_{ij} каждой подзадачи (i,j) пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки

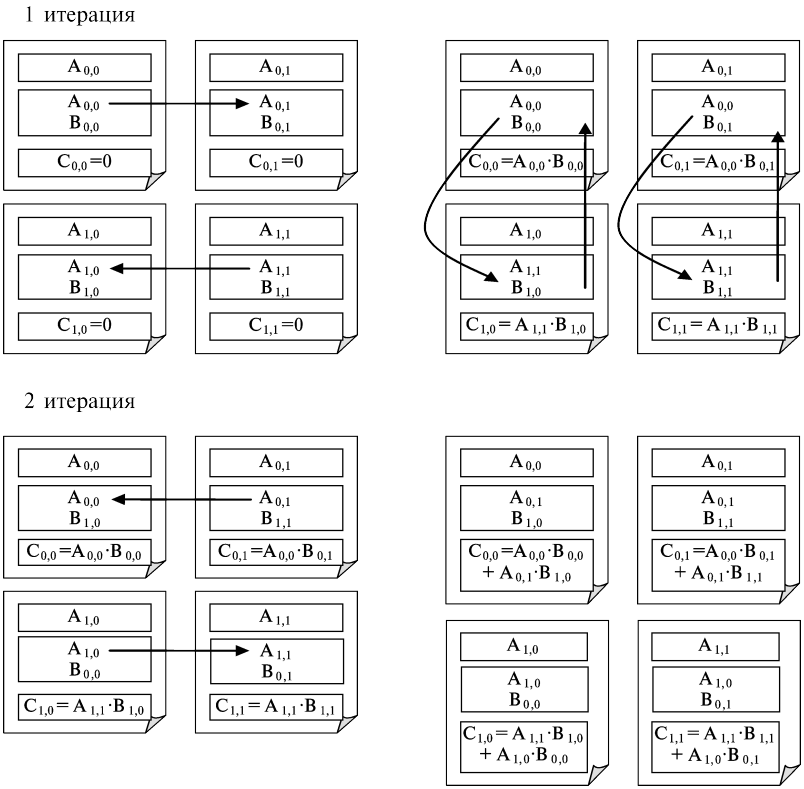


Рис. 7.6. Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса

подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Для пояснения этих правил параллельного метода на рис. 7.6 приведено состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач 2×2).

7.4.3. Масштабирование и распределение подзадач по процессорам

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров p. Так, например, в наиболее простом случае, когда число процессоров представимо в виде $p=\delta^2$ (т.е. является полным квадратом), можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q=\delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и тем самым достигается полная балансировка вычислительной нагрузки между процессорами. В более общем случае при произвольных количестве процессоров и размерах матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся процессоров также в виде квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i,j) следует располагать на процессоре $P_{i,j}$. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

7.4.4. Анализ эффективности

Определим вычислительную сложность данного алгоритма Фокса. Построение оценок будет происходить при условии выполнения всех ранее выдвинутых предположений: все матрицы являются квадратными размера $n\times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен $k\times k$, $k=n/q$), процессоры образуют квадратную решетку и их количество равно $p=q^2$.

Как уже отмечалось, алгоритм Фокса требует для своего выполнения q итераций, в ходе которых каждый процессор перемножает свои текущие блоки матриц A и B и прибавляет результаты умножения к текущему значению блока матрицы C. С учетом выдвинутых предположений общее количество выполняемых при этом операций будет иметь порядок n^3/p. Как результат, показатели ускорения и эффективности алгоритма имеют вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \tag{7.9}$$

Общий анализ сложности снова дает идеальные показатели эффективности параллельных вычислений. Уточним полученные соотношения — для этого укажем более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

Определим количество вычислительных операций. 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 и, как результат, трудоемкость операции блочного умножения оказывается равной $(n^2/p)(2n/q\text{-}1)$. Для сложения блоков требуется n^2/p операций. С учетом всех перечисленных выражений время выполнения вычислительных операций алгоритма Фокса может быть оценено следующим образом:

$$T_p(calc) = q[(n^2/p)\cdot(2n/q-1) + (n^2/p)]\cdot\tau. \tag{7.10}$$

(напомним, что τ есть время выполнения одной элементарной скалярной операции).

Оценим затраты на выполнение операций передачи данных между процессорами. На каждой итерации алгоритма перед умножением блоков один из процессоров строки процессорной решетки рассылает свой блок матрицы A остальным процессорам своей строки. Как уже отмечалось ранее, 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. Как результат, время выполнения операции передачи блоков матрицы A при использовании модели Хокни может оцениваться как

$$T_p^1(comm) = \log_2 q\,(\alpha + w(n^2/p)/\beta), \tag{7.11}$$

где α – латентность, β – пропускная способность сети передачи данных, а w есть размер элемента матрицы в байтах. В случае же когда топология строк процессорной решетки представляет собой кольцо, выражение для оценки времени передачи блоков матрицы A принимает вид:

$$\tilde{T}_p^1(comm) = (q/2)(\alpha + w(n^2/p)/\beta).$$

Далее после умножения матричных блоков процессоры передают свои блоки матрицы *B* предыдущим процессорам по столбцам процессорной решетки (первые процессоры столбцов передают свои данные последним процессорам в столбцах решетки). Эти операции могут быть выполнены процессорами параллельно, и, тем самым, длительность такой коммуникационной операции составляет:

$$T_p^2(comm) = \alpha + w \cdot (n^2/p)/\beta. \qquad (7.12)$$

Просуммировав все полученные выражения, можно получить, что общее время выполнения алгоритма Фокса может быть определено при помощи следующих соотношений:

$$\begin{aligned} T_p &= q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau + q\log_2 q\,(\alpha + w(n^2/p)/\beta) + \\ &+ (q-1)\cdot(\alpha + w(n^2/p)/\beta) = q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau + \\ &+ (q\log_2 q + (q-1))(\alpha + w(n^2/p)/\beta) \end{aligned} \qquad (7.13)$$

(напомним, что параметр q определяет размер процессорной решетки и $q=\sqrt{p}$).

7.4.5. Программная реализация

Представим возможный вариант программной реализации алгоритма Фокса для умножения матриц при блочном представлении данных. Приводимый программный код содержит основные модули параллельной программы, отсутствие отдельных вспомогательных функций не сказывается на общем понимании реализуемой схемы параллельных вычислений.

1. Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 7.1
// Алгоритм Фокса умножения матриц - блочное представление данных
//   Условия выполнения программы: все матрицы квадратные,
//   размер блоков и их количество по горизонтали и вертикали
//   одинаково, процессы образуют квадратную решетку
int ProcNum = 0;      // Количество доступных процессов
int ProcRank = 0;     // Ранг текущего процесса
int GridSize;         // Размер виртуальной решетки процессов
int GridCoords[2];    // Координаты текущего процесса в процессной
                      // решетке
```

```
MPI_Comm GridComm;   // Коммуникатор в виде квадратной решетки
MPI_Comm ColComm;    // коммуникатор – столбец решетки
MPI_Comm RowComm;    // коммуникатор – строка решетки

void main ( int argc, char * argv[] ) {
  double* pAMatrix;  // Первый аргумент матричного умножения
  double* pBMatrix;  // Второй аргумент матричного умножения
  double* pCMatrix;  // Результирующая матрица
  int Size;          // Размер матриц
  int BlockSize;     // Размер матричных блоков, расположенных
                     // на процессах
  double *pAblock;   // Блок матрицы A на процессе
  double *pBblock;   // Блок матрицы B на процессе
  double *pCblock;   // Блок результирующей матрицы C на процессе
  double *pMatrixAblock;
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  GridSize = sqrt((double)ProcNum);
  if (ProcNum != GridSize*GridSize) {
    if (ProcRank == 0) {
      printf ("Number of processes must be a perfect square \n");
    }
  }
  else {
    if (ProcRank == 0)
      printf("Parallel matrix multiplication program\n");

    // Создание виртуальной решетки процессов и коммуникаторов
    // строк и столбцов
    CreateGridCommunicators();

    // Выделение памяти и инициализация элементов матриц
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock,
      pBblock, pCblock, pMatrixAblock, Size, BlockSize );
```

```
    // Блочное распределение матриц между процессами
    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
      BlockSize);

    // Выполнение параллельного метода Фокса
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
      pCblock, BlockSize);

    // Сбор результирующей матрицы на ведущем процессе
    ResultCollection(pCMatrix, pCblock, Size, BlockSize);

    // Завершение процесса вычислений
    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock);
  }

  MPI_Finalize();
}
```

2. Функция `CreateGridCommunicators`**.** Данная функция создает коммуникатор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке, а также создает коммуникаторы отдельно для каждой строки и каждого столбца.

Создание решетки производится при помощи функции `MPI_Cart_create` (вектор `Periodic` определяет возможность передачи сообщений между граничными процессами строк и столбцов создаваемой решетки). После создания решетки каждый процесс параллельной программы будет иметь координаты своего положения в решетке; получение этих координат обеспечивается при помощи функции `MPI_Cart_coords`.

Формирование топологий завершается созданием множества коммуникаторов для каждой строки и каждого столбца решетки в отдельности (функция `MPI_Cart_sub`).

```
// Создание коммуникатора в виде двумерной квадратной решетки
// и коммуникаторов для каждой строки и каждого столбца решетки
void CreateGridCommunicators() {
  int DimSize[2];    // Количество процессов в каждом измерении
                     // решетки
  int Periodic[2];   // =1 для каждого измерения, являющегося
                     // периодическим
  int Subdims[2];    // =1 для каждого измерения, оставляемого
                     // в подрешетке
```

```
  DimSize[0] = GridSize;
  DimSize[1] = GridSize;
  Periodic[0] = 0;
  Periodic[1] = 0;

  // Создание коммуникатора в виде квадратной решетки
  MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

  // Определение координат процесса в решетке
  MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

  // Создание коммуникаторов для строк процессной решетки
  Subdims[0] = 0;  // Фиксация измерения
  Subdims[1] = 1;  // Наличие данного измерения в подрешетке
  MPI_Cart_sub(GridComm, Subdims, &RowComm);

  // Создание коммуникаторов для столбцов процессной решетки
  Subdims[0] = 1;
  Subdims[1] = 0;
  MPI_Cart_sub(GridComm, Subdims, &ColComm);
}
```

3. Функция `ProcessInitialization`. Данная функция определяет параметры решаемой задачи (размеры матриц и их блоков), выделяет память для хранения данных и осуществляет ввод исходных матриц (или формирует их при помощи какого-либо датчика случайных чисел). Всего в каждом процессе должна быть выделена память для хранения четырех блоков – для указателей на выделенную память используются переменные `pAblock`, `pBblock`, `pCblock`, `pMatrixAblock`. Первые три указателя определяют блоки матриц A, B и C соответственно. Следует отметить, что содержимое блоков `pAblock` и `pBblock` постоянно меняется в соответствии с пересылкой данных между процессами, в то время как блок `pMatrixAblock` матрицы A остается неизменным и применяется при рассылках блоков по строкам решетки процессов (см. функцию `AblockCommunication`).

Для определения элементов исходных матриц будем использовать функцию `RandomDataInitialization`, реализацию которой читателю предстоит выполнить самостоятельно.

```
// Функция для выделения памяти и инициализации исходных данных
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, double* &pAblock, double* &pBblock,
```

```
  double* &pCblock, double* &pTemporaryAblock, int &Size,
  int &BlockSize ) {
  if (ProcRank == 0) {
    do {
      printf("\nВведите размер матриц: ");
      scanf("%d", &Size);

      if (Size%GridSize != 0) {
        printf ("Размер матриц должен быть кратен размеру сетки! \n");
      }
    }
    while (Size%GridSize != 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  BlockSize = Size/GridSize;

  pAblock = new double [BlockSize*BlockSize];
  pBblock = new double [BlockSize*BlockSize];
  pCblock = new double [BlockSize*BlockSize];
  pTemporaryAblock = new double [BlockSize*BlockSize];

  for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
  }
  if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
  }
}
```

4. Функции `DataDistribution` и `ResultCollection`. После задания исходных матриц на нулевом процессе необходимо осуществить распределение исходных данных. Для этого предназначена функция `DataDistribution`. Может быть предложено два способа выполнения блочного разделения матриц между процессорами, организованными в двумерную квадратную решетку. В первом из них для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средст-

вами MPI производный тип данных. Во втором способе можно организовать двухэтапную процедуру. На первом этапе матрица разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки. Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки.

Для выполнения сбора результирующей матрицы из блоков предназначена функция `ResultCollection`. Сбор данных также можно выполнить либо с использованием производного типа данных, либо при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы.

Реализация функций `DataDistribution` и `ResultCollection` представляет собой задание для самостоятельной работы.

5. Функция `AblockCommunication`. Функция выполняет рассылку блоков матрицы A по строкам процессорной решетки. Для этого в каждой строке решетки определяется ведущий процесс `Pivot`, осуществляющий рассылку. Для рассылки используется блок `pMatrixAblock`, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции `MPI_Bcast`. Следует отметить, что данная операция является коллективной и ее локализация пределами отдельных строк решетки обеспечивается за счет использования коммуникаторов `RowComm`, определенных для набора процессов каждой строки решетки в отдельности.

```
// Рассылка блоков матрицы A по строкам решетки процессов
void ABlockCommunication (int iter, double *pAblock,
  double* pMatrixAblock, int BlockSize) {

  // Определение ведущего процесса в строке процессной решетки
  int Pivot = (GridCoords[0] + iter) % GridSize;

  // Копирование передаваемого блока в отдельный буфер памяти
  if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
      pAblock[i] = pMatrixAblock[i];
  }

  // Рассылка блока
  MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot,
    RowComm);
}
```

6. Функция `BlockMultiplication`. Функция обеспечивает перемножение блоков матриц *A* и *B*. Следует отметить, что для более легкого понимания рассматриваемой программы приводится простой вариант реализации функции – выполнение операции блочного умножения может быть существенным образом оптимизировано для сокращения времени вычислений. Данная оптимизация может быть направлена, например, на повышение эффективности использования кэша процессоров, векторизации выполняемых операций и т.п.

```
// Умножение матричных блоков
void BlockMultiplication (double *pAblock, double *pBblock,
  double *pCblock, int BlockSize) {
  // Вычисление произведения матричных блоков
  for (int i=0; i<BlockSize; i++) {
    for (int j=0; j<BlockSize; j++) {
      double temp = 0;
      for (int k=0; k<BlockSize; k++ )
        temp += pAblock [i*BlockSize + k] * pBblock [k*BlockSize + j];
      pCblock [i*BlockSize + j] += temp;
    }
  }
}
```

7. Функция `BblockCommunication`. Функция выполняет циклический сдвиг блоков матрицы *B* по столбцам процессорной решетки. Каждый процесс передает свой блок следующему процессу `NextProc` в столбце процессов и получает блок, переданный из предыдущего процесса `PrevProc` в столбце решетки. Выполнение операций передачи данных осуществляется при помощи функции `MPI_SendRecv_replace`, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти `pBblock`. Кроме того, эта функция гарантирует отсутствие возможных тупиков, когда операции передачи данных начинают одновременно выполняться несколькими процессами при кольцевой топологии сети.

```
// Циклический сдвиг блоков матрицы B вдоль столбца процессной
// решетки
void BblockCommunication (double *pBblock, int BlockSize) {
  MPI_Status Status;
  int NextProc = GridCoords[0] + 1;
  if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
  int PrevProc = GridCoords[0] - 1;
  if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;
```

```
    MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
      NextProc, 0, PrevProc, 0, ColComm, &Status);
}
```

8. Функция `ParallelResultCalculation`. Для непосредственного выполнения параллельного алгоритма Фокса умножения матриц предназначена функция `ParallelResultCalculation`, которая реализует логику работы алгоритма.

```
// Функция для параллельного умножения матриц
void ParallelResultCalculation(double* pAblock, double*
pMatrixAblock, double* pBblock, double* pCblock, int BlockSize) {
  for (int iter = 0; iter < GridSize; iter ++) {
    // Рассылка блоков матрицы A по строкам процессной решетки
    ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);
    // Умножение блоков
    BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
    // Циклический сдвиг блоков матрицы B в столбцах процессной
    // решетки
    BblockCommunication(pBblock, BlockSize);
  }
}
```

7.4.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные (см. п. 7.3.5). Результаты экспериментов с использованием четырех и девяти процессоров приведены в таблице 7.3.

Таблица 7.3. Результаты вычислительных экспериментов по исследованию параллельного алгоритма Фокса

Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
500	0,8527	0,2190	3,8925	0,1468	5,8079
1000	12,8787	3,0910	4,1664	2,1565	5,9719
1500	43,4731	10,8678	4,0001	7,2502	5,9960
2000	103,0561	24,1421	4,2687	21,4157	4,8121
2500	201,2915	51,4735	3,9105	41,2159	4,8838
3000	347,8434	87,0538	3,9957	58,2022	5,9764

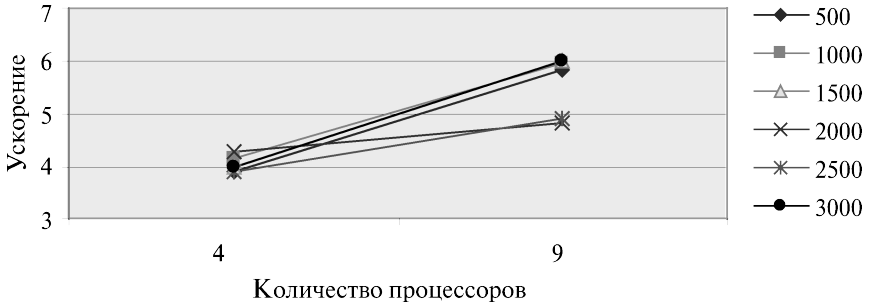


Рис. 7.7. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма Фокса

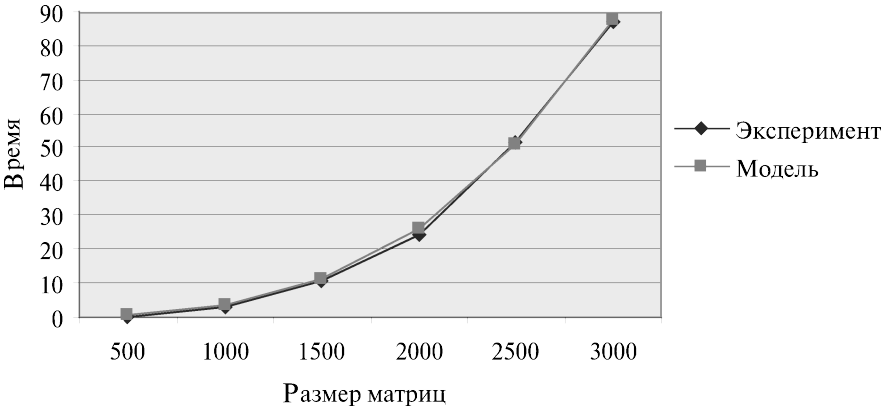


Рис. 7.8. График зависимости экспериментального и теоретического времени выполнения алгоритма Фокса на четырех процессорах

Таблица 7.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Фокса

Размер матриц	**4 процессора**		**9 процессоров**	
	T_p	T_p^*	T_p	T_p^*
500	0,4217	0,2190	0,2200	0,1468
1000	3,2970	3,0910	1,5924	2,1565
1500	11,0419	10,8678	5,1920	7,2502
2000	26,0726	24,1421	12,0927	21,4157
2500	50,8049	51,4735	23,3682	41,2159
3000	87,6548	87,0538	40,0923	58,2022

Сравнение времени выполнения эксперимента T_p^* и теоретического времени T_p, вычисленного в соответствии с выражением (7.13), представлено в таблице 7.4 и на рис. 7.8.

7.5. Алгоритм Кэннона умножения матриц при блочном разделении данных

Рассмотрим еще один параллельный алгоритм матричного умножения, основанный на блочном разбиении матриц, – *алгоритм Кэннона* (*Cannon*).

7.5.1. Определение подзадач

Как и при рассмотрении алгоритма Фокса, в качестве базовой подзадачи выберем вычисления, связанные с определением одного из блоков результирующей матрицы C. Как уже отмечалось ранее, для вычисления элементов этого блока подзадача должна иметь доступ к элементам горизонтальной полосы матрицы A и элементам вертикальной полосы матрицы B.

7.5.2. Выделение информационных зависимостей

Отличие алгоритма Кэннона от метода Фокса состоит в изменении схемы начального распределения блоков перемножаемых матриц между подзадачами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы блоки в подзадачах могли бы быть перемножены без каких-либо дополнительных передач данных. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между подзадачами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение следующих операций передач данных:

— в каждую подзадачу (i,j) передаются блоки A_{ij}, B_{ij};
— для каждой строки i решетки подзадач блоки матрицы A сдвигаются на $(i\text{-}1)$ позиций влево;
— для каждого столбца j решетки подзадач блоки матрицы B сдвигаются на $(j\text{-}1)$ позиций вверх.

Выполняемые при перераспределении матричных блоков процедуры передачи данных являются примером операции *циклического сдвига* – см. лекцию 3. Для пояснения используемого способа начального распределе-

ния данных на рис. 7.9 показан пример расположения блоков для решетки подзадач 3×3.

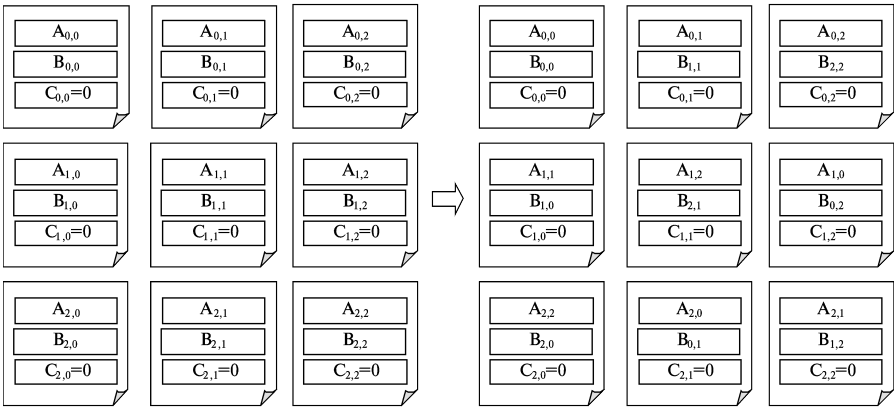


Рис. 7.9. Перераспределение блоков исходных матриц между процессорами при выполнении алгоритма Кэннона

В результате такого начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных. Кроме того, получение всех последующих блоков для подзадач может быть обеспечено при помощи простых коммуникационных действий — после выполнения 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. Как можно показать, последовательность таких циклических сдвигов и умножение получаемых блоков исходных матриц A и B приведет к получению в базовых подзадачах соответствующих блоков результирующей матрицы C.

7.5.3. Масштабирование и распределение подзадач по процессорам

Как и ранее в методе Фокса, для алгоритма Кэннона размер блоков может быть подобран таким образом, чтобы количество базовых подзадач совпадало с числом имеющихся процессоров. Поскольку объемы вычислений в каждой подзадаче являются равными, это обеспечивает полную балансировку вычислительной нагрузки между процессорами.

Для распределения подзадач между процессорами может быть применен подход, использованный в алгоритме Фокса, – множество имею-

щихся процессоров представляется в виде квадратной решетки и размещение базовых подзадач (i,j) осуществляется на процессорах $P_{i,j}$ соответствующих узлов процессорной решетки. Необходимая структура сети передачи данных, как и ранее, может быть обеспечена на физическом уровне при топологии вычислительной системы в виде решетки или полного графа.

7.5.4. Анализ эффективности

Перед проведением анализа эффективности следует отметить, что алгоритм Кэннона отличается от метода Фокса только видом выполняемых в ходе вычислений коммуникационных операций. Как результат, используя оценки времени выполнения вычислительных операций, приведенные в п. 7.4.4, проведем только анализ коммуникационной сложности алгоритма Кэннона.

В соответствии с правилами алгоритма на этапе инициализации производится перераспределение блоков матриц A и B при помощи циклического сдвига матричных блоков по строкам и столбцам процессорной решетки. Трудоемкость выполнения такой операции передачи данных существенным образом зависит от топологии сети. Для сети со структурой полного графа все необходимые пересылки блоков могут быть выполнены одновременно (т.е. длительность операции оказывается равной времени передачи одного матричного блока между соседними процессорами). Для сети с топологией гиперкуба операция циклического сдвига может потребовать выполнения $\log_2 q$ итераций. Для сети с кольцевой структурой связей необходимое количество итераций оказывается равным q-1 – более подробно методы выполнения операции циклического сдвига рассмотрены в лекции 3. Используем для построения оценки коммуникационной сложности этапа инициализации вариант топологии полного графа как более соответствующего кластерным вычислительным системам, время выполнения начального перераспределения блоков может оцениваться как

$$T_p^1(comm) = 2 \cdot (\alpha + w \cdot (n^2/p)/\beta) \qquad (7.14)$$

(выражение n^2/p определяет размер пересылаемых блоков, а коэффициент 2 соответствует двум выполняемым операциям циклического сдвига).

Оценим теперь затраты на передачу данных между процессорами при выполнении основной части алгоритма Кэннона. На каждой итерации алгоритма 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 процессорами параллельно, и, тем самым, длительность таких коммуникационных действий составляет:

$$T_p^2(comm) = 2 \cdot (\alpha + w \cdot (n^2/p)/\beta). \quad (7.15)$$

Поскольку количество итераций алгоритма Кэннона равно q, то с учетом оценки (7.10) общее время выполнения параллельных вычислений может быть определено при помощи следующего соотношения:

$$T_p = q[(n^2/p) \cdot (2n/q - 1) + (n^2/p)] \cdot \tau + (2q+2)(\alpha + w(n^2/p)/\beta). \quad (7.16)$$

(в используемых выражениях параметр $q=\sqrt{p}$ определяет размер процессорной решетки).

7.5.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные (см. п. 7.3.5). Результаты экспериментов для случаев четырех и девяти процессоров приведены в таблице 7.5.

Таблица 7.5 Результаты вычислительных экспериментов по исследованию параллельного алгоритма Кэннона

Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
1000	12,8787	3,0806	4,1805	1,1889	10,8324
1500	43,4731	11,1716	3,8913	4,6310	9,3872
2000	103,0561	24,0502	4,2850	14,4759	7,1191
2500	201,2915	53,1444	3,7876	23,5398	8,5511
3000	347,8434	88,2979	3,9394	36,3688	9,5643

Сравнение времени выполнения эксперимента T_p^* и теоретического времени T_p, вычисленного в соответствии с выражением (7.16), представлено в таблице 7.6 и на рис. 7.11.

Таблица 7.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Кэннона

Размер матриц	4 процессора		9 процессоров	
	T_p	T_p^*	T_p	T_p^*
1000	3,4485	3,0806	1,5669	1,1889
1500	11,3821	11,1716	5,1348	4,6310
2000	26,6769	24,0502	11,9912	14,4759
2500	51,7488	53,1444	23,2098	23,5398
3000	89,0138	88,2979	39,8643	36,3688

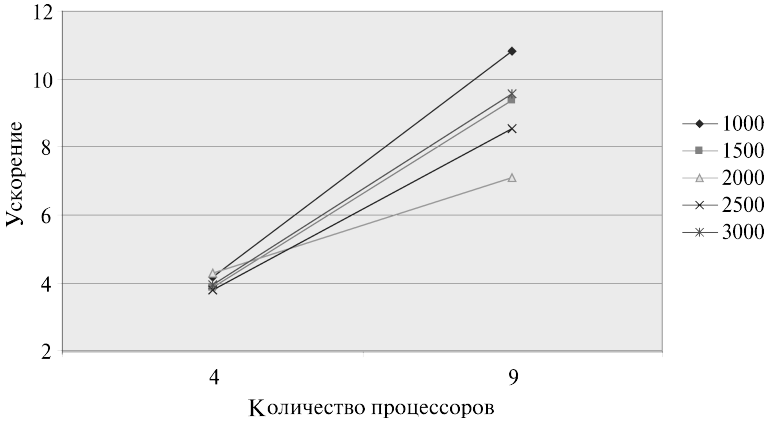


Рис. 7.10. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма Кэннона

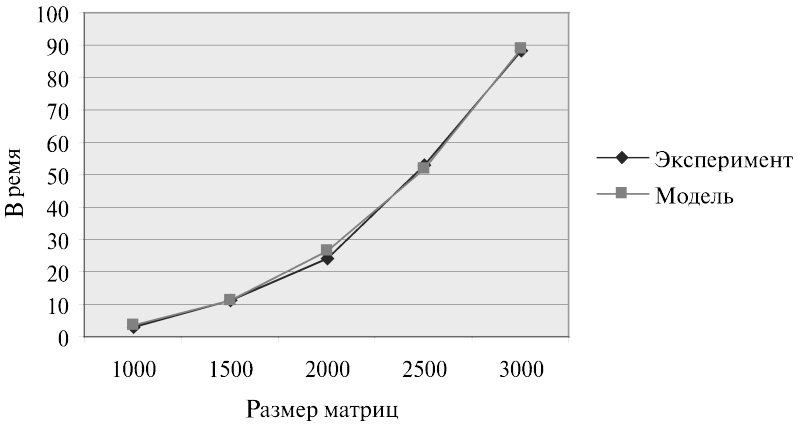


Рис. 7.11. График зависимости экспериментального и теоретического времени выполнения алгоритма Кэннона на четырех процессорах

7.6. Краткий обзор лекции

В данной лекции рассмотрены три параллельных метода для выполнения операции матричного умножения. Первый алгоритм основан на ленточном разделении матриц между процессорами. В лекции приведены два различных варианта этого алгоритма. Первый вариант алгоритма основан на различном разделении перемножаемых матриц – первая матри-

ца (матрица *A*) разбивается на горизонтальные полосы, а вторая матрица (матрица *B*) делится на вертикальные полосы. Второй вариант ленточного алгоритма использует разбиение обеих матриц на горизонтальные полосы.

Далее в лекции рассматриваются широко известные алгоритмы Фокса и Кэннона, основанные на блочном разделении матриц. При использовании одинаковой схемы разбиения матриц данные алгоритмы отличаются характером выполняемых операций передачи данных. Для алгоритма Фокса в ходе вычислений осуществляется рассылка и циклический сдвиг блоков матриц, в алгоритме Кэннона выполняется только операция циклического сдвига.

Различие в способах разбиения данных приводит к разным топологиям коммуникационной сети, при которых выполнение параллельных алгоритмов является наиболее эффективным. Так, алгоритмы, основанные на ленточном разделении данных, ориентированы на топологию сети в виде гиперкуба или полного графа. Для реализации алгоритмов, основанных на блочном разделении данных, необходимо наличие топологии решетки.

На рис. 7.12 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Выполненные расчеты показывают, что при большем количестве процессоров более эффективными становятся блочные алгоритмы умножения матриц.

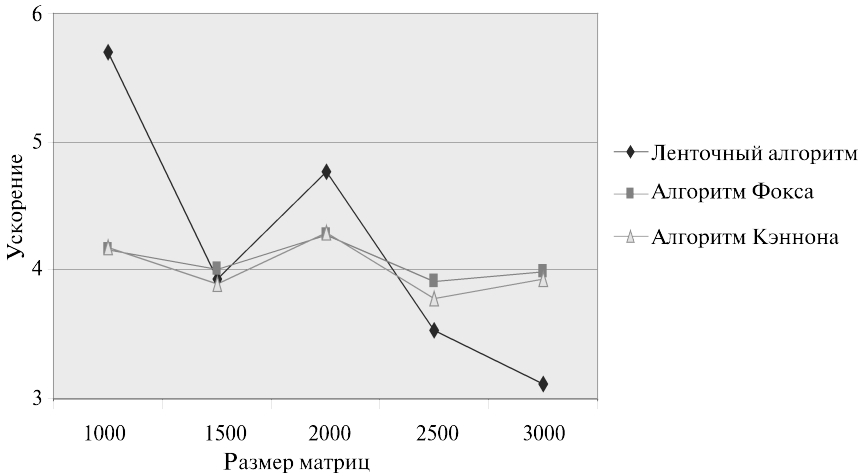


Рис. 7.12. Ускорение трех параллельных алгоритмов при умножении матриц с использованием четырех процессоров

7.7. Обзор литературы

Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы [2, 51, 63]. Широкое обсуждение вопросов параллельного выполнения матричных вычислений приводится в работе [30].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [23]. В ней рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

7.8. Контрольные вопросы

1. В чем состоит постановка задачи умножения матриц?
2. Приведите примеры задач, в которых используется операция умножения матриц.
3. Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
4. Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?
5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матриц.
6. Проведите анализ и получите показатели эффективности ленточного алгоритма при горизонтальном разбиении перемножаемых матриц.
7. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
8. Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?
9. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
10. Какой из рассмотренных алгоритмов характеризуется наименьшими и наибольшими требованиями к объему необходимой памяти?
11. Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
12. Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
13. Дайте общую характеристику программной реализации алгоритма Фокса. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?

14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

7.9. Задачи и упражнения

1. Выполните реализацию двух ленточных алгоритмов умножения матриц. Сравните время выполнения этих алгоритмов.
2. Выполните реализацию алгоритма Кэннона. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее полученными теоретическими оценками.
3. Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных процессорных решеток общего вида.
4. Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.

Лекция 8. Решение систем линейных уравнений

В лекции рассматривается задача решения систем линейных уравнений. Приводятся необходимые определения и постановка задачи. Описывается последовательный и параллельный варианты одного из прямых методов решения линейных систем общего вида – метода Гаусса. Далее дается описание последовательного и параллельного алгоритмов, реализующих итерационный метод сопряженных градиентов.

Ключевые слова: линейное уравнение, система линейных уравнений, решение системы линейных уравнений, прямые и итерационные методы решения систем линейных уравнений, декомпозиция вычислений, балансировка вычислений, ленточная циклическая схема разделения данных, информационная зависимость и масштабирование вычислений, алгоритм Гаусса, метод сопряженных градиентов, вычислительная и коммуникационная сложность алгоритма, ускорение вычислений, программная реализация, операции передачи данных, MPI, вычислительный эксперимент.

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых дифференциальными, интегральными или системами нелинейных (трансцендентных) уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и др.

Матрицы коэффициентов систем линейных уравнений могут иметь различные структуру и свойства. Матрицы решаемых систем могут быть плотными, и их порядок может достигать несколько тысяч строк и столбцов. При решении многих задач могут появляться системы, обладающие симметричными положительно определенными ленточными матрицами с порядком в десятки тысяч и шириной ленты в несколько тысяч элементов. И, наконец, при рассмотрении большого ряда задач могут возникать системы линейных уравнений с разреженными матрицами с порядком в миллионы строк и столбцов.

8.1. Постановка задачи

Линейное уравнение с n неизвестными $x_0, x_1, \ldots, x_{n-1}$ может быть определено при помощи выражения

$$a_0 x_0 + a_1 x_1 + \ldots + a_{n-1} x_{n-1} = b, \qquad (8.1)$$

где величины a_0, a_1, ..., a_{n-1} и b представляют собой постоянные значения.

Множество из n линейных уравнений

$$\begin{array}{llll} a_{0,0}x_0 & +a_{0,1}x_1 & +\ldots+a_{0,n-1}x_{n-1} & =b_0 \\ a_{1,0}x_0 & +a_{1,1}x_1 & +\ldots+a_{1,n-1}x_{n-1} & =b_1 \\ \ldots & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +\ldots+a_{n-1,n-1}x_{n-1} & =b_{n-1} \end{array} \qquad (8.2)$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (*матричном*) виде система может представлена как

$$Ax = b,$$

где $A=(a_{i,j})$ есть вещественная матрица размера $n \times n$, а векторы b и x состоят из n элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы A и вектора b обычно понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

8.2. Алгоритм Гаусса

Метод Гаусса – широко известный *прямой* алгоритм решения систем линейных уравнений, для которых матрицы коэффициентов являются плотными. Если система линейных уравнений *невырожденна*, то метод Гаусса гарантирует нахождение решения с погрешностью, определяемой точностью машинных вычислений. Основная идея метода состоит в приведении матрицы A посредством эквивалентных преобразований (не меняющих решение системы (8.2)) к треугольному виду, после чего значения искомых неизвестных могут быть получены непосредственно в явном виде.

В подразделе дается общая характеристика метода Гаусса, достаточная для начального понимания алгоритма и позволяющая рассмотреть возможные способы параллельных вычислений при решении систем линейных уравнений. Более полное изложение алгоритма со строгим обсуждением вопросов точности получаемых решений может быть получено, например, в работах [6, 22, 47] и др.

8.2.1. Последовательный алгоритм

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решения рас-

сматриваемой системы (такие преобразования носят наименование *эквивалентных*). К числу таких преобразований относятся:

- умножение любого из уравнений на ненулевую константу;
- перестановка уравнений;
- прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход* метода Гаусса – исходная система линейный уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду

$$Ux=c,$$

где матрица коэффициентов получаемой системы имеет вид

$$U=\begin{pmatrix} u_{0,0} & u_{0,1} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & \dots & u_{1,n-1} \\ & & \dots & \\ 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}.$$

На *обратном ходе* метода Гаусса (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д.

8.2.1.1. Прямой ход алгоритма Гаусса

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 – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a'_{kj} &= a_{kj} - (a_{ki}/a_{ii}) \cdot a_{ij}, \\ b'_k &= b_k - (a_{ki}/a_{ii}) \cdot b_i, \end{aligned} \quad i \le j \le n-1, i < k \le n-1, 0 \le i < n-1$$

(следует отметить, что аналогичные вычисления выполняются и над вектором b).

Поясним выполнение прямого хода метода Гаусса на примере системы линейных уравнений вида:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1,\\ 2x_0 &+7x_1 &+5x_2 &= 18,\\ x_0 &+4x_1 &+6x_2 &= 26. \end{aligned}$$

На первой итерации производится исключение неизвестной x_0 из второй и третьей строки. 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, и в окончательной форме система имеет следующий вид:

$$\begin{aligned} x_0 &+3x_1 &+2x_2 &= 1,\\ x_1 &+x_2 &= 16,\\ &3x_2 &= 9. \end{aligned}$$

На рис. 8.1 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 нужную ненулевую константу. После проведения (n-1) подобной итерации матрица, определяющая систему линейных уравнений, становится приведенной к верхнему треугольному виду.

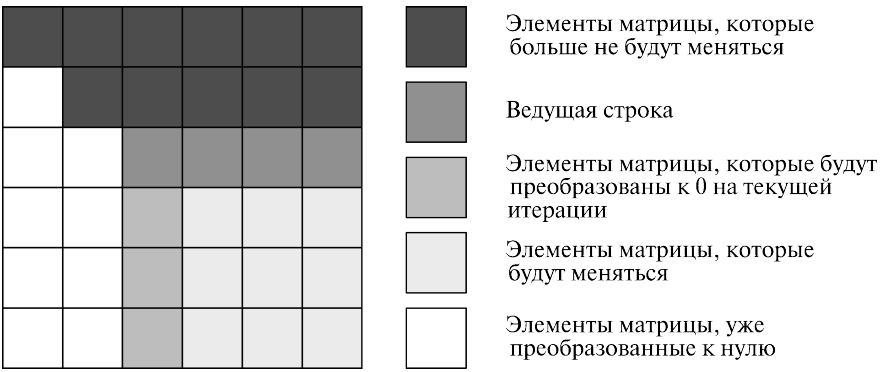


Рис. 8.1. Итерация прямого хода алгоритма Гаусса

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент $a_{i,i}$ имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Возможный способ избежать подобной проблемы может состоять в следующем: при выполнении каждой очередной итерации прямого хода метода Гаусса следует определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Вычислительная сложность прямого хода алгоритма Гаусса с выбором ведущей строки имеет порядок $O(n^3)$.

8.2.1.2. Обратный ход алгоритма Гаусса

После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д. В общем виде выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$
$$x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \quad i = n-2, n-3, ..., 0.$$

Поясним, как и ранее, выполнение обратного хода метода Гаусса на примере рассмотренной в п. 8.2.1.1 системы линейных уравнений

$$\begin{aligned} x_0 + 3x_1 + 2x_2 &= 1, \\ x_1 + x_2 &= 16, \\ 3x_2 &= 9. \end{aligned}$$

Из последнего уравнения системы можно определить, что неизвестная x_2 имеет значение 3. В результате становится возможным разрешение второго уравнения и определение значение неизвестной x_1=13, т.е.

$$\begin{aligned} x_0 \quad +3x_1 \quad +2x_2 \quad &= \quad 1, \\ x_1 \qquad\quad &= \quad 13, \\ x_2 \quad &= \quad 3. \end{aligned}$$

На последней итерации обратного хода метода Гаусса определяется значение неизвестной x_0, равное -44.

С учетом последующего параллельного выполнения можно отметить, что вычисление получаемых значений неизвестных может выполняться сразу во всех уравнениях системы (и эти действия могут выполняться в уравнениях одновременно и независимо друг от друга). Так, в рассматриваемом примере после определения значения неизвестной x_2 система уравнений может быть приведена к виду

$$\begin{aligned} x_0 \quad +3x_1 \quad &= \quad -5, \\ x_1 \quad &= \quad 13, \\ x_2 \quad &= \quad 3. \end{aligned}$$

Вычислительная сложность обратного хода алгоритма Гаусса составляет $O(n^2)$.

8.2.2. Определение подзадач

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве *базовой подзадачи* можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b.

8.2.3. Выделение информационных зависимостей

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения **прямого хода** метода Гаусса необходимо осуществить (n-1) итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации i, $0 \le i < n-1$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Поскольку строки матрицы A распределены по подзадачам, для поиска максимального значения подзадачи с номерами k, $k>i$, должны обменяться своими элементами при исключаемой переменной x_i. После сбора всех необходимых данных в каждой подзадаче может быть определено, какая из подзадач содержит ведущую строку и какое значение является ведущим элементом.

Далее 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.

При выполнении **обратного хода** метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $0 \le i < n-1$, определяет значение своей переменной x_i, это значение должно быть разослано всем подзадачам с номерами k, $k<i$. Далее подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

8.2.4. Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и сбалансированным объемом передаваемых данных. В случае когда размер матрицы, описывающей систему линейных уравнений, оказывается большим, чем число доступных процессоров (т.е. $p<n$), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. Однако применение использованной в лекциях 6 и 7 последовательной схемы разделения данных для параллельного решения систем линейных уравнений приведет к неравномерной вычислительной нагрузке процессоров: по мере исключения (на прямом ходе) или определения (на обратном ходе) неизвестных в методе Гаусса для все большей части процессоров все необходимые вычисления будут завершены и процессоры окажутся простаивающими. Возможное решение проблемы балансировки вычислений может состоять в использовании ленточной циклической схемы (см. лекцию 6) для распределения данных между укрупненными подзадачами. В этом случае матрица A делится на наборы (полосы) строк вида (см. рис. 8.2):

$$A = (A_0, A_2, ..., A_{p-1})^T,\ A_i = (a_{i_0}, a_{i_1}, ..., a_{i_{k-1}}),\ i_j = i + jp,\ 0 \le j < k,\ k = n/p$$

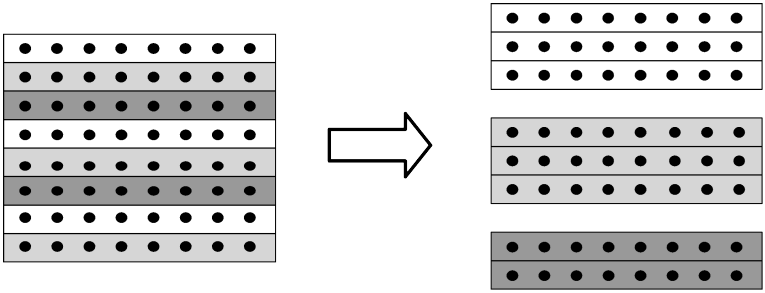


Рис. 8.2. Пример использования ленточной циклической схемы разделения строк матрицы между тремя процессорами

Сопоставив схему разделения данных и порядок выполнения вычислений в методе Гаусса, можно отметить, что использование циклического способа формирования полос позволяет обеспечить лучшую балансировку вычислительной нагрузки между подзадачами.

Распределение подзадач между процессорами должно учитывать характер выполняемых в методе Гаусса коммуникационных операций. Основным видом информационного взаимодействия подзадач является операция передачи данных от одного процессора всем процессорам вычислительной системы. Как результат, для эффективной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должны иметь структуру гиперкуба или полного графа.

8.2.5. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного варианта метода Гаусса. Пусть, как и ранее, n есть порядок решаемой системы линейных уравнений, а p, $p<n$, обозначает число используемых процессоров. Тем самым, матрица коэффициентов A имеет размер $n \times n$, и, соответственно, n/p есть размер полосы матрицы A на каждом процессоре.

Прежде всего, несложно показать, что общее время выполнения последовательного варианта метода Гаусса составляет:

$$T_1 = 2n^3/3 + n^2. \qquad (8.3)$$

Определим теперь сложность параллельного варианта метода Гаусса. При выполнении **прямого хода** алгоритма на каждой итерации для выбо-

ра ведущей строки каждый процессор должен осуществить выбор максимального значения в столбце с исключаемой неизвестной в пределах своей полосы. Начальный размер полос на процессорах равен n/p, но по мере исключения неизвестных количество строк в полосах для обработки постепенно сокращается. Текущий размер полос приближенно можно оценить как $(n-i)/p$, где i, $0 \le i < n-1$, есть номер выполняемой итерации прямого хода метода Гаусса. Далее после сбора полученных максимальных значений, определения и рассылки ведущей строки каждый процессор должен выполнить вычитание ведущей строки из каждой строки оставшейся части строк своей полосы матрицы A. Количество элементов строки, подлежащих обработке, также сокращается при исключении неизвестных, и текущее число элементов строки для вычислений оценивается величиной $(n-i)$. Тем самым, сложность процедуры вычитания строк оценивается как $2\cdot(n-i)$ операций (перед вычитанием ведущая строка умножается на масштабирующую величину a_{ik}/a_{ii}). С учетом выполняемого количества итераций общее число операций параллельного варианта прямого хода метода Гаусса определяется выражением:

$$T_p^1 = \sum_{i=0}^{n-2}\left[\frac{(n-i)}{p} + \frac{(n-i)}{p}\cdot 2(n-i)\right] = \frac{1}{p}\sum_{i=0}^{n-2}\left[(n-i)+2(n-i)^2\right].$$

На каждой итерации **обратного хода** алгоритма Гаусса после рассылки вычисленного значения очередной неизвестной каждый процессор должен обновить значения правых частей для всех строк, расположенных на этом процессоре. Отсюда следует, что трудоемкость параллельного варианта обратного хода алгоритма Гаусса оценивается как величина:

$$T_p^2 = \sum_{i=0}^{n-2} 2\cdot(n-i)/\,p.$$

Просуммировав полученные выражения, можно получить

$$T_p = \frac{1}{p}\sum_{i=0}^{n-2}\left[(n-i)+2(n-i)^2\right] + \frac{2}{p}\sum_{i=0}^{n-2}(n-i) = \frac{1}{p}\sum_{i=0}^{n-2}\left[3(n-i)+2(n-i)^2\right] = \frac{1}{p}\sum_{i=2}^{n}(3i+2i^2).$$

Как результат выполненного анализа, показатели ускорения и эффективности параллельного варианта метода Гаусса могут быть определены при помощи соотношений следующего вида:

$$S_p = \frac{(2n^3\ 3+n^2)}{\frac{1}{p}\sum_{i=2}^{n}(3i+2i^2)}\ ,\quad E_p = \frac{(2n^3\ 3+n^2)}{\sum_{i=2}^{n}(3i+2i^2)}\ . \qquad (8.4)$$

Полученные соотношения имеют достаточно сложный вид для оценивания. Вместе с тем можно показать, что сложность параллельного алгоритма имеет порядок $\sim(2n^3/3)/p$, и, тем самым, балансировка вычислительной нагрузки между процессорами в целом является достаточно равномерной.

Дополним сформированные показатели вычислительной сложности метода Гаусса оценкой затрат на выполнение операций передачи данных между процессорами. При выполнении **прямого хода** на каждой итерации для определения ведущей строки процессоры обмениваются локально найденными максимальными значениями в столбце с исключаемой переменной. Выполнение данного действия одновременно с определением среди собираемых величин наибольшего значения может быть обеспечено при помощи операции обобщенной редукции (функция `MPI_Allreduce` библиотеки MPI). Всего для выполнения такой операции требуется $\log_2 p$ шагов, что с учетом количества итераций позволяет оценить время, необходимое для проведения операций редукции, при помощи следующего выражения:

$$T_p^1(comm) = (n-1)\cdot \log_2 p \cdot (\alpha + w/\beta),$$

где, как и ранее, α – латентность сети передачи данных, β – пропускная способность сети, w – размер пересылаемого элемента данных.

Далее также на каждой итерации прямого хода метода Гаусса выполняется рассылка выбранной ведущей строки. Сложность данной операции передачи данных:

$$T_p^2(comm) = (n-1)\cdot \log_2 p \cdot (\alpha + wn/\beta).$$

При выполнении **обратного хода** алгоритма Гаусса на каждой итерации осуществляется рассылка между всеми процессорами вычисленного значения очередной неизвестной. Общее время, необходимое для выполнения подобных действий, можно оценить как:

$$T_p^3(comm) = (n-1)\cdot \log_2 p \cdot (\alpha + w/\beta).$$

В итоге, с учетом всех полученных выражений, трудоемкость параллельного варианта метода Гаусса составляет:

$$T_p = \frac{1}{p}\sum_{i=2}^{n}(3i+2i^2)\,\tau + (n-1)\cdot \log_2 p \cdot (3\alpha + w(n+2)/\beta), \qquad (8.5)$$

где τ есть время выполнения базовой вычислительной операции.

8.2.6. Программная реализация

Рассмотрим возможный вариант параллельной реализации метода Гаусса для решения систем линейных уравнений. Следует отметить, что для получения более простого вида программы для разделения матрицы между процессами используется ленточная последовательная схема (полосы матрицы образуют последовательные наборы соседних строк матрицы). В описании программы реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 9.1. – Алгоритм Гаусса решения систем линейных уравнений
int ProcNum;          // Число доступных процессоров
int ProcRank;         // Ранг текущего процессора
int *pParallelPivotPos; // Номера строк, которые были выбраны ведущими
int *pProcPivotIter;  // Номера итераций, на которых строки
                      // процессора использовались в качестве ведущих
void main(int argc, char* argv[]) {
  double* pMatrix;       // Матрица линейной системы
  double* pVector;       // Вектор правых частей линейной системы
  double* pResult;       // Вектор неизвестных
  double *pProcRows;     // Строки матрицы A
  double *pProcVector;   // Блок вектора b
  double *pProcResult;   // Блок вектора x
  int     Size;          // Размер матрицы и векторов
  int     RowNum;        // Количество строк матрицы
  double start, finish, duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank );
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum );

  if (ProcRank == 0)
    printf("Параллельный метод Гаусса для решения систем линейных
    уравнений\n");

  // Выделение памяти и инициализация данных
  ProcessInitialization(pMatrix, pVector, pResult,
```

```
    pProcRows, pProcVector, pProcResult, Size, RowNum);

  // Распределение исходных данных
  DataDistribution(pMatrix, pProcRows, pVector, pProcVector,
    Size, RowNum);

  // Выполнение параллельного алгоритма Гаусса
  ParallelResultCalculation(pProcRows, pProcVector, pProcResult, Size,
    RowNum);

  // Сбор найденного вектора неизвестных на ведущем процессе
  ResultCollection(pProcResult, pResult);

  // Завершение процесса вычислений
  ProcessTermination(pMatrix, pVector, pResult, pProcRows,
    pProcVector, pProcResult);

  MPI_Finalize();
}
```

Следует пояснить использование дополнительных массивов. Элементы массива `pParallelPivotPos` определяют номера строк матрицы, выбираемых в качестве ведущих, по итерациям прямого хода метода Гаусса. Именно в этом порядке должны выполняться затем итерации обратного хода для определения значений неизвестных системы линейных уравнений. Массив `pParallelPivotPos` является глобальным, и любое его изменение в одном из процессов требует выполнения операции рассылки измененных данных всем остальным процессам программы.

Элементы массива `pProcPivotIter` определяют номера итераций прямого хода метода Гаусса, на которых строки процесса использовались в качестве ведущих (т.е. строка *i* процесса выбиралась ведущей на итерации `pProcPivotIter[i]`). Начальное значение элементов массива устанавливается нулевым, и, тем самым, нулевое значение элемента массива `pProcPivotIter[i]` является признаком того, что строка *i* процесса все еще подлежит обработке. Кроме того, важно отметить, что запоминаемые в элементах массива `pProcPivotIter` номера итераций дополнительно означают и номера неизвестных, для определения которых будут использованы соответствующие строки уравнения. Массив `pProcPivotIter` является локальным для каждого процесса.

Функция `ProcessInitialization` определяет исходные данные решаемой задачи (число неизвестных), выделяет память для хранения данных,

осуществляет ввод матрицы коэффициентов системы линейных уравнений и вектора правых частей (или формирует эти данные при помощи какого-либо датчика случайных чисел).

Функция `DataDistribution` реализует распределение матрицы линейной системы и вектора правых частей между процессорами вычислительной системы.

Функция `ResultCollection` осуществляет сбор со всех процессов отдельных частей вектора неизвестных.

Функция `ProcessTermination` выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция `ParallelResultCalculation`. Реализует логику работы параллельного алгоритма Гаусса, последовательно вызывает функции, выполняющие прямой и обратный ход метода.

```
// Функция для параллельного выполнения метода Гаусса
void ParallelResultCalculation (double* pProcRows,
  double* pProcVector, double* pProcResult, int Size, int RowNum) {
  ParallelGaussianElimination (pProcRows, pProcVector, Size,
    RowNum);
  ParallelBackSubstitution (pProcRows, pProcVector, pProcResult,
    Size, RowNum);
}
```

3. Функция `ParallelGaussianElimination`. Функция выполняет параллельный вариант прямого хода алгоритма Гаусса.

```
// Функция для параллельного выполнения прямого хода метода Гаусса
void ParallelGaussianElimination (double* pProcRows,
  double* pProcVector, int Size, int RowNum) {
  double MaxValue;   // Значение ведущего элемента на процессоре
  int    PivotPos;   // Положение ведущей строки в полосе линейной
                     // системы даннного процессора
  // Структура для выбора ведущего элемента
  struct { double MaxValue; int ProcRank; } ProcPivot, Pivot;

  // pPivotRow используется для хранения ведущей строки матрицы и
  // соответствующего элемента вектора b
```

```
double* pPivotRow = new double [Size+1];
// Итерации прямого хода метода Гаусса
for (int i=0; i<Size; i++) {

  // Нахождение ведущей строки среди строк процесса
  double MaxValue = 0;
  for (int j=0; j<RowNum; j++) {
    if ((pProcPivotIter[j] == -1) &&
        (MaxValue < fabs(pProcRows[j*Size+i]))) {
      MaxValue = fabs(pProcRows[j*Size+i]);
      PivotPos = j;
    }
  }
  ProcPivot.MaxValue = MaxValue;
  ProcPivot.ProcRank = ProcRank;

  // Нахождение ведущего процесса (процесса, который содержит
  // максимальное значение переменной MaxValue)
  MPI_Allreduce(&ProcPivot, &Pivot, 1, MPI_DOUBLE_INT,
    MPI_MAXLOC, MPI_COMM_WORLD);

  // Рассылка ведущей строки
  if ( ProcRank == Pivot.ProcRank ){
    pProcPivotIter[PivotPos]= i; // номер итерации
    pParallelPivotPos[i]= pProcInd[ProcRank] + PivotPos;
  }
  MPI_Bcast(&pParallelPivotPos[i], 1, MPI_INT, Pivot.ProcRank,
    MPI_COMM_WORLD);

  if ( ProcRank == Pivot.ProcRank ){
    // заполнение ведущей строки
    for (int j=0; j<Size; j++) {
      pPivotRow[j] = pProcRows[PivotPos*Size + j];
    }
    pPivotRow[Size] = pProcVector[PivotPos];
  }
  MPI_Bcast(pPivotRow, Size+1, MPI_DOUBLE, Pivot.ProcRank,
    MPI_COMM_WORLD);
```

```
    ParallelEliminateColumns(pProcRows, pProcVector, pPivotRow,
      Size, RowNum, i);
    }
  }
```

Функция `ParallelEliminateColumns` проводит вычитание ведущей строки из строк процесса, которые еще не использовались в качестве ведущих (т.е. для которых элементы массива `pProcPivotIter` равны нулю).

4. Функция `ParallelBackSubstitution`. Функция реализует параллельный вариант обратного хода Гаусса.

```
// Функция для параллельного выполнения обратного хода метода Гаусса
void ParallelBackSubstitution (double* pProcRows, double* pProcVector,
  double* pProcResult, int Size, int RowNum) {
  int IterProcRank; // Ранг процессора, который содержит ведущую строку
  int IterPivotPos; // Положение ведущей строки в полосе процессора
  double IterResult;  // Вычисленное значение очередной неизвестной
  double val;

  // Итерации обратного хода метода Гаусса
  for (int i=Size-1; i>=0; i--) {

    // Вычисление ранга процессора, который содержит ведущую строку
    FindBackPivotRow(pParallelPivotPos[i], Size, IterProcRank,
      IterPivotPos);

    // Вычисление значения неизвестной
    if (ProcRank == IterProcRank) {
      IterResult =
pProcVector[IterPivotPos]/pProcRows[IterPivotPos*Size+i];
      pProcResult[IterPivotPos] = IterResult;
    }
    // Рассылка значения очередной неизвестной
    MPI_Bcast(&IterResult, 1, MPI_DOUBLE, IterProcRank,
       MPI_COMM_WORLD);

    // Обновление вектора правых частей
    for (int j=0; j<RowNum; j++)
      if ( pProcPivotIter[j] < i ) {
        val = pProcRows[j*Size + i] * IterResult;
        pProcVector[j]=pProcVector[j] - val;
    }
  }
}
```

Функция `FindBackPivotRow` определяет строку, из которой можно вычислить значение очередного элемента результирующего вектора. Номер этой строки хранится в массиве `pParallelPivotIter`. По номеру функция `FindBackPivotRow` определяет номер процесса, на котором эта строка хранится, и номер этой строки в полосе `pProcRows` этого процесса.

8.2.7. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода Гаусса для решения систем линейных уравнений проводились при условиях, указанных в п. 6.5.5, и состоят в следующем.

Эксперименты производились на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server (см. п. 1.2.3).

Для оценки длительности τ базовой скалярной операции проводилось решение системы линейных уравнений при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 4,7 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 47 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина w равна 8 байт.

Результаты вычислительных экспериментов приведены в таблице 8.1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров.

Таблица 8.1. Результаты вычислительных экспериментов для параллельного алгоритма Гаусса

Размер системы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
500	0,36	0,3302	1,0901	0,5170	0,6963	0,7504	0,4796
1000	3,313	1,5950	2,0770	1,6152	2,0511	1,8715	1,7701
1500	11,437	4,1788	2,7368	3,8802	2,9474	3,7567	3,0443
2000	26,688	9,3432	2,8563	7,2590	3,6765	7,3713	3,6204
2500	50,125	16,9860	2,9509	11,9957	4,1785	11,6530	4,3014
3000	85,485	28,4948	3,0000	19,1255	4,4696	17,6864	4,8333

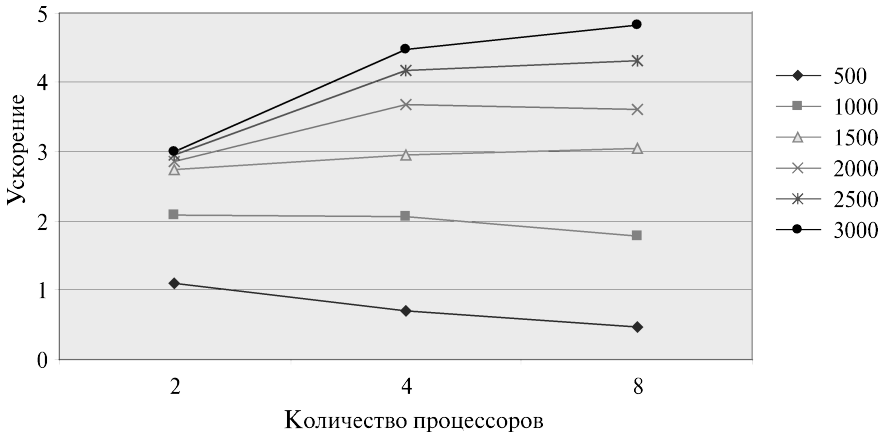


Рис. 8.3. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма Гаусса для разных размеров систем линейных уравнений

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (8.5) приведено в таблице 8.2 и на рис. 8.4.

Таблица 8.2. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Гаусса

Размер системы	2 процессора		4 процессора		8 процессоров	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	0,2393	0,3302	0,2819	0,5170	0,3573	0,7504
1000	1,3373	1,5950	1,1066	1,6152	1,1372	1,8715
1500	4,0750	4,1788	2,8643	3,8802	2,5345	3,7567
2000	9,2336	9,3432	5,9457	7,2590	4,7447	7,3713
2500	17,5941	16,9860	10,7412	11,9957	7,9628	11,6530
3000	29,9377	28,4948	17,6415	19,1255	12,3843	17,6864

8.3. Метод сопряженных градиентов

Рассмотрим теперь совершенно иной подход к решению систем линейных уравнений, при котором к искомому точному решению x* системы $Ax=b$ строится последовательность приближенных решений x^0, x^1,..., x^k,... При этом процесс вычислений организуется таким способом, что каждое очередное приближение дает оценку точного решения со все уменьшающейся погрешностью, и при продолжении расчетов оценка точного решения может быть получена с любой требуемой точностью. Подоб-

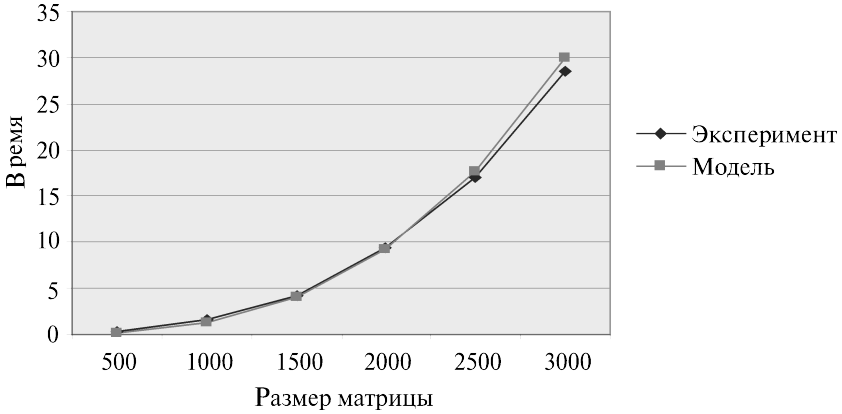


Рис. 8.4. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

ные *итерационные методы* решения систем линейных уравнений широко используются в практике вычислений. К преимуществам итерационных методов можно отнести меньший объем (по сравнению, например, с методом Гаусса) необходимых вычислений для решения разреженных систем линейных уравнений, возможность более быстрого получения начальных приближений искомого решения, наличие эффективных способов организации параллельных вычислений. Дополнительная информация с описанием таких методов, рассмотрением вопросов сходимости и точности получаемых решений может быть получена, например, в [6, 22].

Одним из наиболее известных итерационных алгоритмов является *метод сопряженных градиентов*, который может быть применен для решения симметричной положительно определенной системы линейных уравнений большой размерности.

Напомним, что матрица A является *симметричной*, если она совпадает со своей транспонированной матрицей, т.е. $A=A^T$. Матрица A называется *положительно определенной*, если для любого вектора x справедливо: $x^TAx>0$.

Известно (см., например, [6, 22]), что после выполнения n итераций алгоритма (n есть порядок решаемой системы линейных уравнений), очередное приближение x^n совпадает с точным решением.

8.3.1. Последовательный алгоритм

Если матрица A симметричная и положительно определенная, то функция

$$q(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T b + c \tag{8.6}$$

имеет единственный минимум, который достигается в точке x^*, совпадающей с решением системы линейных уравнений (8.2). *Метод сопряженных градиентов* является одним из широкого класса итерационных алгоритмов, которые позволяют найти решение (8.2) путем минимизации функции $q(x)$.

Итерация метода сопряженных градиентов состоит в вычислении очередного приближения к точному решению в соответствии с правилом:

$$x^k = x^{k-1} + s^k d^k . \tag{8.7}$$

Тем самым, новое значение приближения x^k вычисляется с учетом приближения, построенного на предыдущем шаге x^{k-1}, скалярного шага s^k и вектора направления d^k.

Перед выполнением первой итерации векторы x^0 и d^0 полагаются равными нулю, а для вектора g^0 устанавливается значение, равное $-b$. Далее каждая итерация для вычисления очередного значения приближения x^k включает выполнение четырех шагов:

Шаг 1: Вычисление градиента:

$$g^k = A \cdot x^{k-1} - b ; \tag{8.8}$$

Шаг 2: Вычисление вектора направления:

$$d^k = -g^k + \frac{\left((g^k)^T, g^k\right)}{\left((g^{k-1})^T, g^{k-1}\right)} d^{k-1} , \tag{8.9}$$

где (g^T,g) есть скалярное произведение векторов.

Шаг 3: Вычисление величины смещения по выбранному направлению:

$$s^k = \frac{\left(d^k, g^k\right)}{(d^k)^T \cdot A \cdot d^k} . \tag{8.10}$$

Шаг 4: Вычисление нового приближения:

$$x^k = x^{k-1} + s^k d^k . \tag{8.11}$$

Как можно заметить, данные выражения включают две операции умножения матрицы на вектор, четыре операции скалярного произведения и пять операций над векторами. Как результат, общее количество операций, выполняемых на одной итерации, составляет

$t_1=2n^2+13n.$

Как уже отмечалось ранее, для нахождения точного решения системы линейных уравнений с положительно определенной симметричной матрицей необходимо выполнить n итераций. Таким образом, для нахождения решения системы необходимо выполнить

$$T_1=2n^3+13n^2, \tag{8.12}$$

и, тем самым, сложность алгоритма имеет порядок $O(n^3)$.

Поясним выполнение метода сопряженных градиентов на примере решения системы линейных уравнений вида:

$$\begin{aligned} 3x_0 - x_1 &= 3, \\ -x_0 + 3x_1 &= 7. \end{aligned}$$

Эта система уравнений второго порядка обладает симметричной положительно определенной матрицей, для нахождения точного решения этой системы достаточно провести всего две итерации метода.

На первой итерации было получено значение градиента g^1=(-3, -7), значение вектора направления d^1=(3, 7), значение величины смещения s^1=0,439. Соответственно, очередное приближение к точному решению системы x^1=(1,318, 3,076).

На второй итерации было получено значение градиента g^2=(-2,121, 0,909), значение вектора направления d^2=(2,397, -0,266), а величина смещения – s^2=0,284. Очередное приближение совпадает с точным решением системы x^2=(2, 3).

На рис. 8.5 представлена последовательность приближений к точному решению, построенная методом сопряженных градиентов (в качестве начального приближения x^0 выбрана точка (0,0)).

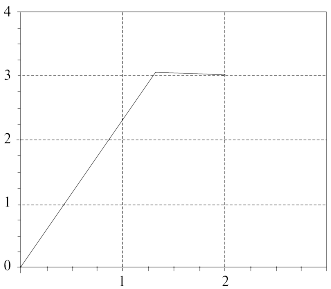


Рис. 8.5. Итерации метода сопряженных градиентов при решении системы второго порядка

8.3.2. Организация параллельных вычислений

При разработке параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений в первую очередь следует учесть, что выполнение итераций метода осуществляется последовательно и, тем самым, наиболее целесообразный подход состоит в распараллеливании вычислений, реализуемых в ходе выполнения итераций.

Анализ соотношений (8.8) – (8.11) показывает, что основные вычисления, выполняемые в соответствии с методом, состоят в умножении матрицы A на векторы x и d, и, как результат, при организации параллельных вычислений может быть полностью использован материал, изложенный в лекции 6.

Дополнительные вычисления в (8.8) – (8.11), имеющие меньший порядок сложности, представляют собой различные операции обработки векторов (скалярное произведение, сложение и вычитание, умножение на скаляр). Организация таких вычислений, конечно же, должна быть согласована с выбранным параллельным способом выполнения операции умножения матрицы на вектор. Общие же рекомендации могут состоять в следующем: при малом размере векторов можно применить дублирование векторов между процессорами, при большом порядке решаемой системы линейных уравнений более целесообразно осуществлять блочное разделение векторов.

8.3.3. Анализ эффективности

Выберем для дальнейшего анализа эффективности получаемых параллельных вычислений параллельный алгоритм матрично-векторного умножения при ленточном горизонтальном разделении матрицы и при полном дублировании всех обрабатываемых векторов.

Трудоемкость последовательного метода сопряженных градиентов была уже определена ранее в (8.12).

Определим время выполнения параллельной реализации метода сопряженных градиентов. Вычислительная сложность параллельных операций умножения матрицы на вектор при использовании схемы ленточного горизонтального разделения матрицы составляет:

$$T_p^1(calc) = 2n\lceil n/p \rceil \cdot (2n-1)$$ (см. лекцию 6).

Как результат, при условии дублирования всех вычислений над векторами общая вычислительная сложность параллельного варианта метода сопряженных градиентов равна:

$$T_p(calc) = n(2 \cdot \lceil n/p \rceil \cdot (2n-1) + 13n).$$

С учетом полученных оценок показатели ускорения и эффективности параллельного алгоритма могут быть выражены при помощи соотношений:

$$S_p = \frac{2n^3 + 13n^2}{n\left(2\lceil n/p \rceil \cdot (2n-1) + 13n\right)}, \quad E_p = \frac{2n^3 + 13n^2}{p \cdot n\left(2\lceil n/p \rceil \cdot (2n-1) + 13n\right)}.$$

Рассмотрев построенные показатели, можно отметить, что балансировка вычислительной нагрузки между процессорами в целом является достаточно равномерной.

Уточним теперь приведенные выражения – учтем длительность выполняемых вычислительных операций и оценим трудоемкость операции передачи данных между процессорами. Как можно заметить, информационное взаимодействие процессоров возникает только при выполнении операций умножения матрицы на вектор. С учетом результатов лекции 6 коммуникационная сложность рассматриваемых параллельных вычислений равна:

$$T_p(comm) = 2n\left(\alpha \cdot \lceil \log p \rceil + w(n/p)(p-1)/\beta\right),$$

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

Окончательно, время выполнения параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений принимает вид:

$$T_p = n \cdot \left[\left(2\lceil n/p \rceil \cdot (2n-1) + 13n\right) \cdot \tau + 2\left(\alpha \cdot \lceil \log p \rceil + w(n/p)(p-1)/\beta\right)\right], \qquad (8.13)$$

где τ есть время выполнения базовой вычислительной операции.

8.3.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода сопряженных градиентов для решения систем линейных уравнений проводились при условиях, указанных в п. 8.2.7.

Результаты вычислительных экспериментов приведены в таблице 8.3. Эксперименты проводились на вычислительных системах, состоящих из двух, четырех и восьми процессоров.

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (8.13) приведено в таблице 8.4 и на рис. 8.7.

Таблица 8.3. Результаты вычислительных экспериментов для параллельного метода сопряженных градиентов для решения систем линейных уравнений

Размер системы	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
500	0,5	0,4634	1,0787	0,4706	1,0623	1,3020	0,3840
1000	8,14	3,9207	2,0761	3,6354	2,2390	3,5092	2,3195
1500	31,391	17,9505	1,7487	14,4102	2,1783	20,2001	1,5539
2000	92,36	51,3204	1,7996	40,7451	2,2667	37,9319	2,4348
2500	170,549	125,3005	1,3611	85,0761	2,0046	87,2626	1,9544
3000	363,476	223,3364	1,6274	146,1308	2,4873	134,1359	2,7097

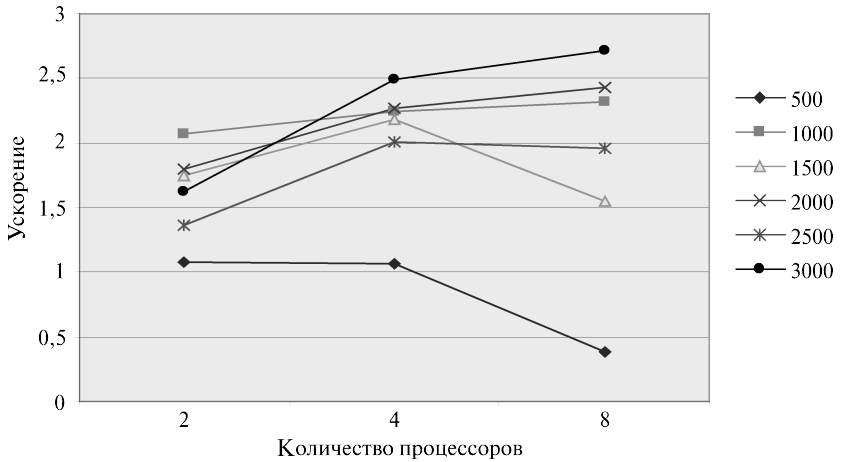


Рис. 8.6. Зависимость ускорения от количества процессоров при выполнении параллельного метода сопряженных градиентов для решения систем линейных уравнений

8.4. Краткий обзор лекции

Данная лекция посвящена проблеме параллельных вычислений при решении систем линейных уравнений. Изложение учебного материала проводится с использованием двух широко известных алгоритмов: *метода Гаусса* как примера прямого алгоритма решения задачи и итерационного *метода сопряженных градиентов*.

Параллельный вариант метода Гаусса (подраздел 8.2) основывается на ленточном разделении матрицы между процессорами с использова-

Таблица 8.4. Сравнение экспериментального и теоретического времени выполнения параллельного метода сопряженных градиентов решения системы линейных уравнений

Размер системы	2 процессора		4 процессора		8 процессоров	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	1,3042	0,4634	0,6607	0,4706	0,3390	1,3020
1000	10,3713	3,9207	5,2194	3,6354	2,6435	3,5092
1500	34,9333	17,9505	17,5424	14,4102	8,8470	20,2001
2000	82,7220	51,3204	41,4954	40,7451	20,8822	37,9319
2500	161,4695	125,3005	80,9446	85,0761	40,6823	87,2626
3000	278,9077	223,3364	139,7560	146,1308	70,1801	134,1359

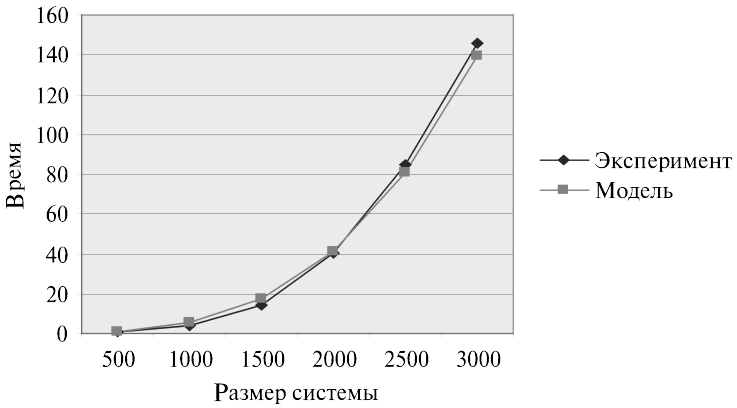


Рис. 8.7. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных

нием циклической схемы распределения строк, что позволяет сбалансировать вычислительную нагрузку. Для определения параллельного варианта метода проводится полный цикл проектирования – определяются базовые подзадачи, выделяются информационные взаимодействия, обсуждаются вопросы масштабирования, выводятся оценки показателей эффективности, предлагается схема программной реализации и приводятся результаты вычислительных экспериментов. Как видно из графика, представленного на рис. 8.8, параллельный алгоритм Гаусса демонстрирует достаточно высокие показатели ускорения и эффективности.

Важный момент при рассмотрении *параллельного варианта метода сопряженных градиентов* (подраздел 8.3) состоит в том, что способ параллельных вычислений для этого метода может быть получен через параллельные алгоритмы выполняемых вычислительных действий – операций умножения матрицы на вектор, скалярного произведения векторов, сложения и вычитания векторов. Выбранный для учебного примера подход состоит в распараллеливании вычислительно наиболее трудоемкой операции умножения матрицы на вектор, в то время как все вычисления над векторами дублируются на каждом процессоре для уменьшения числа выполняемых операций передачи данных — показатели эффективности подобного способа организации параллельных вычислений представлены на рис. 8.8.

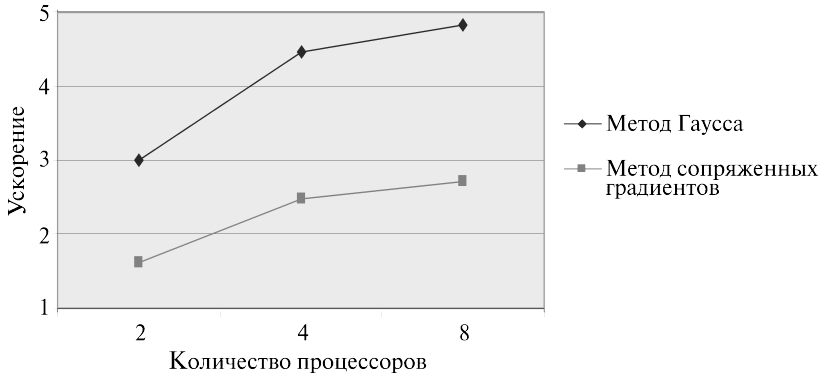


Рис. 8.8. Ускорение параллельных алгоритмов решения системы линейных уравнений с размером матрицы 3000×3000

8.5. Обзор литературы

Проблема численного решения систем линейных уравнений широко рассматривается в литературе. Для учебного рассмотрения могут быть рекомендованы работы [6, 13, 51, 63]. Широкое обсуждение вопросов параллельных вычислений для решения данного типа задач выполнено в работах [22, 30].

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа [23]. В ней рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

8.6. Контрольные вопросы

1. Что представляет собой система линейных уравнений? Какие типы систем вам известны? Какие методы могут быть использованы для решения систем разных типов?

2. В чем состоит постановка задачи решения системы линейных уравнений?
3. В чем идея параллельной реализации метода Гаусса?
4. Какие информационные взаимодействия имеются между базовыми подзадачами для параллельного варианта метода Гаусса?
5. Каковы показатели эффективности для параллельного варианта метода Гаусса?
6. В чем состоит схема программной реализации параллельного варианта метода Гаусса?
7. В чем состоит идея параллельной реализации метода сопряженных градиентов?
8. Какой из алгоритмов обладает большей коммуникационной сложностью?

8.7. Задачи и упражнения

1. Выполните анализ эффективности параллельных вычислений в отдельности для прямого и обратного этапов метода Гаусса. Оцените, на каком этапе происходит большее снижение показателей.
2. Выполните разработку параллельного варианта метода Гаусса при вертикальном разбиении матрицы по столбцам. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.
3. Выполните реализацию параллельного метода сопряженных градиентов. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.
2. Выполните разработку параллельных вариантов методов Якоби и Зейделя решения систем линейных уравнений (см., например, Kumar (1994)). Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты выполненных экспериментов с ранее полученными теоретическими оценками.

Лекция 9. Параллельные методы сортировки

В лекции рассматриваются различные алгоритмы сортировки данных. Излагаются как общие принципы, применяемые при распараллеливании, так и конкретные алгоритмы. Теоретически оценивается эффективность рассматриваемых алгоритмов. Приводятся и анализируются результаты вычислительных экспериментов.

Ключевые слова: сортировка, операция «сравнить и переставить», пузырьковая сортировка, чет-нечетная сортировка, алгоритм Шелла, быстрая сортировка, обобщенная быстрая сортировка, сортировка с использованием регулярного набора образцов, ускорение, эффективность, вычислительная и коммуникационная сложность, теоретическая оценка времени выполнения алгоритма, программная реализация, операции передачи данных, MPI, вычислительный эксперимент.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, ..., a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

(здесь и далее все пояснения для краткости будут даваться только на примере упорядочивания данных по возрастанию).

Возможные способы решения этой задачи широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов сортировки* содержится в работе [50], среди последних изданий может быть рекомендована работа [26].

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T \sim n^2.$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$T \sim n\log_2 n$.

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких ($p>1$) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

Оставляя подробный анализ проблемы сортировки для отдельного рассмотрения, здесь основное внимание мы уделим изучению параллельных способов выполнения для ряда широко известных *методов внутренней сортировки*, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ.

9.1. Принципы распараллеливания

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно заметить, что многие методы основаны на применении одной и той же *базовой операции «сравнить и переставить»* (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановке этих значений, если их порядок не соответствует условиям сортировки.

Пример 9.1. Операция «сравнить и переставить»

```
// Базовая операция "сравнить и переставить"
if ( A[i] > A[j] ) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения, собственно, и проявляется различие алгоритмов сортировки.

Для параллельного обобщения выделенной базовой операции сортировки рассмотрим первоначально ситуацию, когда количество процессоров совпадает с числом сортируемых значений (т. е. $p=n$) и на каждом из процессоров содержится только по одному значению исходного набора данных. Тогда сравнение значений a_i и a_j, располагаемых соответственно на процессорах P_i и P_j, можно организовать следующим образом (параллельное обобщение базовой операции сортировки):

- выполнить взаимообмен имеющихся на процессорах P_i и P_j значений (с сохранением на этих процессорах исходных элементов);
- сравнить на каждом процессоре P_i и P_j получившиеся одинаковые пары значений (a_i, a_j); результаты сравнения используются для разделения данных между процессорами – на одном процессоре (например, P_i) остается меньший элемент, другой процессор (т. е. P_j) запоминает для дальнейшей обработки большее значение пары

$$a_i' = \min(a_i, a_j), \quad a_j' = \max(a_i, a_j).$$

9.2. Масштабирование параллельных вычислений

Рассмотренное параллельное обобщение базовой операции сортировки может быть надлежащим образом адаптировано и для случая $p<n$, когда количество процессоров меньше числа упорядочиваемых значений. В данной ситуации каждый процессор будет содержать уже не единственное значение, а часть (блок размера n/p) сортируемого набора данных.

Определим в качестве *результата выполнения параллельного алгоритма* сортировки такое состояние упорядочиваемого набора данных, при котором имеющиеся на процессорах данные упорядочены, а порядок распределения блоков по процессорам соответствует линейному порядку нумерации (т. е. значение последнего элемента на процессоре P_i меньше значения первого элемента на процессоре P_{i+1}, где $0 \le i < p-1$ или равно ему).

Блоки обычно упорядочиваются в самом начале сортировки на каждом процессоре в отдельности при помощи какого-либо быстрого алгоритма (начальная стадия параллельной сортировки). Далее, следуя схеме одноэлементного сравнения, взаимодействие пары процессоров P_i и P_{i+1} для совместного упорядочения содержимого блоков A_i и A_{i+1} может быть осуществлено следующим образом:

- выполнить взаимообмен блоков между процессорами P_i и P_{i+1};
- объединить блоки A_i и A_{i+1} на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_{i+1} процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных);

- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i, а другую часть (с большими значениями, соответственно) – на процессоре P_{i+1}

$$[A_i \cup A_{i+1}]_{copm} = A_i^{'} \cup A_{i+1}^{'} : \forall a_i^{'} \in A_i^{'}, \forall a_j^{'} \in A_{i+1}^{'} \Rightarrow a_i^{'} \le a_j^{'} \ .$$

Рассмотренная процедура обычно именуется в литературе *операцией «сравнить и разделить»* (*compare-split*). Следует отметить, что сформированные в результате такой процедуры блоки на процессорах P_i и P_{i+1} совпадают по размеру с исходными блоками A_i и A_{i+1} и все значения, расположенные на процессоре P_i, не превышают значений на процессоре P_{i+1}.

Определенная выше операция «сравнить и разделить» может быть использована в качестве *базовой подзадачи* для организации параллельных вычислений. Как следует из построения, количество таких подзадач параметрически зависит от числа имеющихся процессоров, и, таким образом, проблема масштабирования вычислений для параллельных алгоритмов сортировки практически отсутствует. Вместе с тем следует отметить, что относящиеся к подзадачам блоки данных изменяются в ходе выполнения сортировки. В простых случаях размер блоков данных в подзадачах остается неизмененным. В более сложных ситуациях (как, например, в алгоритме быстрой сортировки – см. подраздел 9.5) объем располагаемых на процессорах данных может различаться, что может приводить к нарушению равномерной вычислительной загрузки процессоров.

9.3. Пузырьковая сортировка

9.3.1. Последовательный алгоритм

Последовательный *алгоритм пузырьковой сортировки* (*the bubble sort algorithm*) (см., например, [26, 50]) сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$(a_1, a_2, \ldots, a_n)$

алгоритм сначала выполняет n-1 базовых операций «сравнения-обмена» для последовательных пар элементов

$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$.

В результате после первой итерации алгоритма самый большой элемент перемещается («всплывает») в конец последовательности. Далее последний элемент в преобразованной последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности

$(a'_1, a'_2, ..., a'_{n-1})$.

Как можно увидеть, последовательность будет отсортирована после n-1 итерации. Эффективность пузырьковой сортировки может быть улучшена, если завершать алгоритм в случае отсутствия каких-либо изменений сортируемой последовательности данных в ходе какой-либо итерации сортировки.

Алгоритм 9.1. Последовательный алгоритм пузырьковой сортировки

```
// Алгоритм 9.1.
// Последовательный алгоритм пузырьковой сортировки
void BubbleSort(double A[], int n) {
  for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < n - i; j++)
      compare_exchange(A[j], A[j + 1]);
}
```

9.3.2. Алгоритм чет-нечетной перестановки

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как *метод чет-нечетной перестановки* (*the odd-even transposition method*) – см., например, [51]. Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода: в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами. Таким образом, на всех нечетных итерациях сравниваются пары

(a_1, a_2), (a_3, a_4), ..., (a_{n-1},a_n) (при четном n),

а на четных итерациях обрабатываются элементы

(a_2, a_3), (a_4, a_5), ..., (a_{n-2}, a_{n-1}).

После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

Алгоритм 9.2. Последовательный алгоритм чет-нечетной перестановки

```
// Алгоритм 9.2
// Последовательный алгоритм чет-нечетной перестановки
void OddEvenSort(double A[], int n) {
  for (int i = 1; i < n; i++) {
    if (i % 2 == 1) {    // нечетная итерация
      for (int j = 0; j < n/2 - 2; j++)
        compare_exchange(A[2*j + 1], A[2*j + 2]);
      if (n % 2 == 1) // сравнение последней пары при нечетном n
        compare_exchange(A[n - 2], A[n - 1]);
     }
     else    // четная итерация
       for (int j = 1; j < n/2 - 1; j++)
         compare_exchange(A[2*j], A[2*j + 1]);
  }
}
```

9.3.3. Определение подзадач и выделение информационных зависимостей

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. В случае $p<n$, когда количество процессоров меньше числа упорядочиваемых значений, процессоры содержат блоки данных размера n/p и в качестве *базовой подзадачи* может быть использована операция «сравнить и разделить» (см. подраздел 9.2).

Алгоритм 9.3. Параллельный алгоритм чет-нечетной перестановки

```
// Алгоритм 9.3
// Параллельный алгоритм чет-нечетной перестановки
ParallelOddEvenSort(double A[], int n) {
  int id = GetProcId();      // номер процесса
  int np = GetProcNum();     // количество процессов
```

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

Для пояснения такого параллельного способа сортировки в табл. 9.1 приведен пример упорядочения данных при n=16, p=4 (т. е. блок значений на каждом процессоре содержит n/p=4 элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары процессов, для которых параллельно выполняются операции «сравнить и разделить». Взаимодействующие пары процессов выделены в таблице рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

В общем случае выполнение параллельного метода может быть прекращено, если в течение каких-либо двух последовательных итераций сортировки состояние упорядочиваемого набора данных не было изменено. Как результат, общее количество итераций может быть сокращено, и для фиксации таких моментов необходимо введение некоторого управляющего процессора, который определял бы состояние набора данных после выполнения каждой итерации сортировки. Однако трудоемкость такой коммуникационной операции (сборка на одном процессоре сообщений от всех процессоров) может оказаться столь значительной, что весь эффект от возможного сокращения итераций сортировки будет поглощен затратами на реализацию операций межпроцессорной передачи данных.

Таблица 9.1. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
(1, 2), (3, 4)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
(2, 3)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 нечет	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
(1, 2), (3, 4)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
(2, 3)	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

9.3.4. Масштабирование и распределение подзадач по процессорам

Как отмечалось ранее, количество подзадач соответствует числу имеющихся процессоров, и поэтому необходимости в проведении масштабирования вычислений не возникает. Исходное распределение блоков упорядочиваемого набора данных по процессорам может быть выбрано совершенно произвольным образом. Для эффективного выполнения рассмотренного параллельного алгоритма сортировки нужно, чтобы процессоры с соседними номерами имели прямые линии связи.

9.3.5. Анализ эффективности

При анализе эффективности, как и ранее, вначале проведем общую оценку сложности рассмотренного параллельного алгоритма сортировки, а затем дополним полученные соотношения показателями трудоемкости выполняемых коммуникационных операций.

Определим первоначально трудоемкость последовательных вычислений. При рассмотрении данного вопроса алгоритм пузырьковой сортировки позволяет продемонстрировать следующий важный момент. Как уже отмечалось в начале этой лекции, использованный для распараллеливания последовательный метод упорядочивания данных характеризуется квадратичной зависимостью сложности от числа упорядочиваемых данных, т. е. $T_1 \sim n^2$. Однако применение подобной оценки сложности последовательного алгоритма приведет к искажению исходного целевого на-

значения критериев качества параллельных вычислений – показатели эффективности в этом случае будут характеризовать используемый способ параллельного выполнения данного конкретного метода сортировки, а не результативность использования параллелизма для задачи упорядочивания данных в целом как таковой. Различие состоит в том, что для сортировки могут быть применены более эффективные последовательные алгоритмы, трудоемкость которых имеет порядок

$$T_1 \sim n\log_2 n, \tag{9.1}$$

и чтобы сравнить, насколько быстрее могут быть упорядочены данные при использовании параллельных вычислений, в обязательном порядке должна применяться именно эта оценка сложности. Как основной результат приведенных рассуждений, можно сформулировать утверждение о том, что при определении показателей ускорения и эффективности параллельных вычислений в качестве оценки сложности последовательного способа решения рассматриваемой задачи следует использовать трудоемкость наилучших последовательных алгоритмов. Параллельные методы решения задач должны сравниваться с наиболее быстродействующими последовательными способами вычислений!

Определим теперь сложность рассмотренного параллельного алгоритма упорядочивания данных. Как отмечалось ранее, на начальной стадии работы метода каждый процессор проводит упорядочивание своих блоков данных (размер блоков при равномерном распределении данных равен n/p). Предположим, что данная начальная сортировка может быть выполнена при помощи быстродействующих алгоритмов упорядочивания данных, тогда трудоемкость начальной стадии вычислений можно определить выражением вида:

$$T_p^1 = (n/p)\log_2(n/p). \tag{9.2}$$

Далее, на каждой выполняемой итерации параллельной сортировки взаимодействующие пары процессоров осуществляют передачу блоков между собой, после чего получаемые на каждом процессоре пары блоков объединяются при помощи процедуры слияния. Общее количество итераций не превышает величины p, и, как результат, общее количество операций этой части параллельных вычислений оказывается равным

$$T_p^2 = 2p(n/p) = 2n. \tag{9.3}$$

С учетом полученных соотношений показатели эффективности и ускорения параллельного метода сортировки имеют вид:

$$S_p = \frac{n\log_2 n}{(n/p)\log_2(n/p)+2n} = \frac{p\log_2 n}{\log_2(n/p)+2p},\quad E_p = \frac{n\log_2 n}{p((n/p)\log_2(n/p)+2n)} = \frac{\log_2 n}{\log_2(n/p)+2p}. \quad (9.4)$$

Расширим приведенные выражения – учтем длительность выполняемых вычислительных операций и оценим трудоемкость операции передачи блоков между процессорами. При использовании модели Хокни общее время всех выполняемых в ходе сортировки операций передачи блоков можно оценить при помощи соотношения вида:

$$T_p(comm) = p\cdot(\alpha + w\cdot(n/p)/\beta), \quad (9.5)$$

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

С учетом трудоемкости коммуникационных действий общее время выполнения параллельного алгоритма сортировки определяется следующим выражением:

$$T_p = ((n/p)\log_2(n/p)+2n)\tau + p\cdot(\alpha + w\cdot(n/p)/\beta), \quad (9.6)$$

где τ есть время выполнения базовой операции сортировки.

9.3.6. Результаты вычислительных экспериментов

Эксперименты осуществлялись на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server.

Для оценки длительности τ базовой скалярной операции алгоритма сортировки проводилось решение задачи упорядочивания при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате выполненных экспериментов для величины τ было получено значение 10,41 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 130 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, размер которого на данной платформе равен 8 байт (следовательно w=8).

Результаты вычислительных экспериментов приведены в табл. 9.2. Эксперименты выполнялись с использованием двух и четырех процессоров.

Таблица 9.2. Результаты вычислительных экспериментов для параллельного алгоритма пузырьковой сортировки

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,002210	0,643439	0,003270	0,434862
20 000	0,002991	0,004428	0,675474	0,004596	0,650783
30 000	0,004612	0,006745	0,683766	0,006873	0,671032
40 000	0,006297	0,008033	0,783891	0,009107	0,691446
50 000	0,008014	0,009770	0,820266	0,010840	0,739299

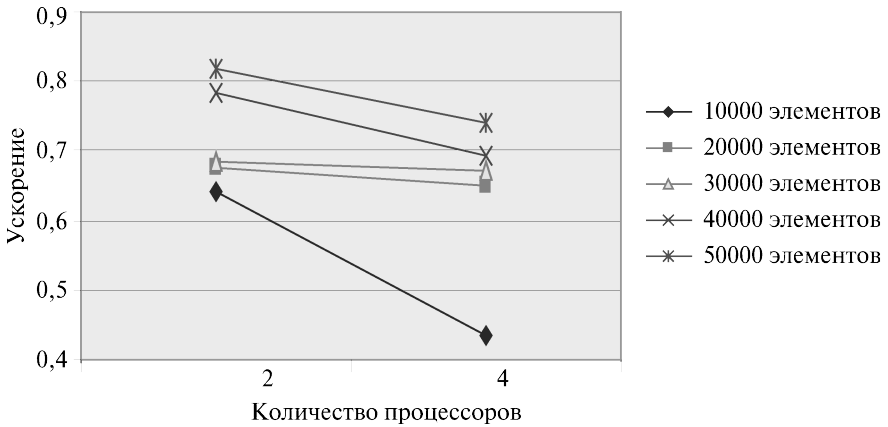


Рис. 9.1. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма пузырьковой сортировки

Как можно заметить из приведенных результатов вычислительных экспериментов, параллельный вариант алгоритма сортировки работает медленнее исходного последовательного метода пузырьковой сортировки, т. к. объем передаваемых данных между процессорами является достаточно большим и сопоставим с количеством выполняемых вычислительных операций (и этот дисбаланс объема вычислений и сложности операций передачи данных увеличивается с ростом числа процессоров).

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (9.6) приведено в таблице 9.3 и на рис. 9.2.

Таблица 9.3. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма пузырьковой сортировки

Количество элементов	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,002003	0,002210	0,002057	0,003270
20 000	0,003709	0,004428	0,003366	0,004596
30 000	0,005455	0,006745	0,004694	0,006873
40 000	0,007227	0,008033	0,006035	0,009107
50 000	0,009018	0,009770	0,007386	0,010840

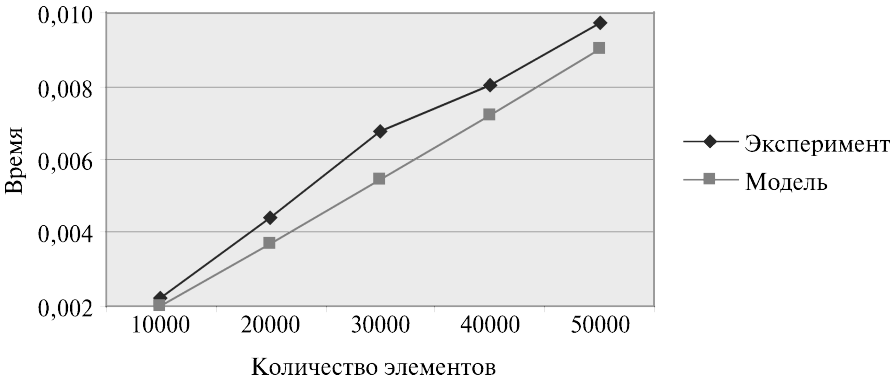


Рис. 9.2. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

9.4. Сортировка Шелла

9.4.1. Последовательный алгоритм

Общая идея *сортировки Шелла* (*the Shell sort*) (см., например, [26, 50]) состоит в сравнении на начальных стадиях сортировки пар значений, располагаемых достаточно далеко друг от друга в упорядочиваемом наборе данных. Такая модификация метода сортировки позволяет быстро переставлять далекие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

Общая схема метода состоит в следующем. На первом шаге алгоритма происходит упорядочивание элементов $n/2$ пар $(a_i, a_{n/2+i})$ для $1 \le i \le n/2$. Далее, на втором шаге упорядочиваются элементы в $n/4$ группах из четырех элементов $(a_i, a_{n/4+i}, a_{n/2+i}, a_{3n/4+i})$ для $1 \le i \le n/4$. На третьем шаге упорядочиваются элементы уже в $n/4$ группах из восьми элементов и т. д. На последнем шаге упорядочиваются элементы сразу во всем массиве $(a_1, a_2, \ldots, a_n)$. На каждом шаге для упорядочивания элементов в группах применяется метод сортировки вставками. Как можно заметить, общее количество итераций алгоритма Шелла равно $\log_2 n$.

В более полном виде алгоритм Шелла может быть представлен следующим образом.

Алгоритм 9.4. Последовательный алгоритм сортировки Шелла

```
// Алгоритм 9.4
// Последовательный алгоритм сортировки Шелла
void ShellSort(double A[], int n) {
  int incr = n / 2;
  while (incr > 0) {
    for (int i = incr + 1; i < n; i++) {
      int j = i - incr;
      while (j > 0)
        if (A[j] > A[j + incr]) {
          swap(A[j], A[j + incr]);
          j = j - incr;
        }
        else j = 0;
    }
    incr = incr/2;
  }
}
```

9.4.2. Организация параллельных вычислений

Для алгоритма Шелла может быть предложен параллельный аналог метода (см., например, [51]), если топология коммуникационной сети может быть эффективно представлена в виде N-мерного гиперкуба (т. е. количество процессоров равно $p = 2^N$). Выполнение сортировки в таком случае может быть разделено на два последовательных этапа. На первом этапе (N итераций) осуществляется взаимодействие процессоров, являющихся соседними в структуре гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации; для установления соответст-

вия двух систем нумерации процессоров может быть использован, как и ранее, код Грея – см. лекцию 3). Формирование пар процессоров, взаимодействующих между собой при выполнении операции «сравнить и разделить», может быть обеспечено при помощи следующего простого правила – на каждой итерации i, $0 \le i < N$, парными становятся процессоры, у которых различие в битовых представлениях их номеров имеется только в позиции N-i.

Второй этап состоит в реализации обычных итераций параллельного алгоритма чет-нечетной перестановки. Итерации данного этапа выполняются до прекращения фактического изменения сортируемого набора, и, тем самым, общее количество L таких итераций может быть различным – от 2 до p.

На рис. 9.3 показан пример сортировки массива из 16 элементов с помощью рассмотренного способа (процессоры показаны кружками, номера процессоров даны в битовом представлении). Нужно заметить, что данные оказываются упорядоченными уже после первого этапа и нет необходимости выполнять чет-нечетную перестановку.

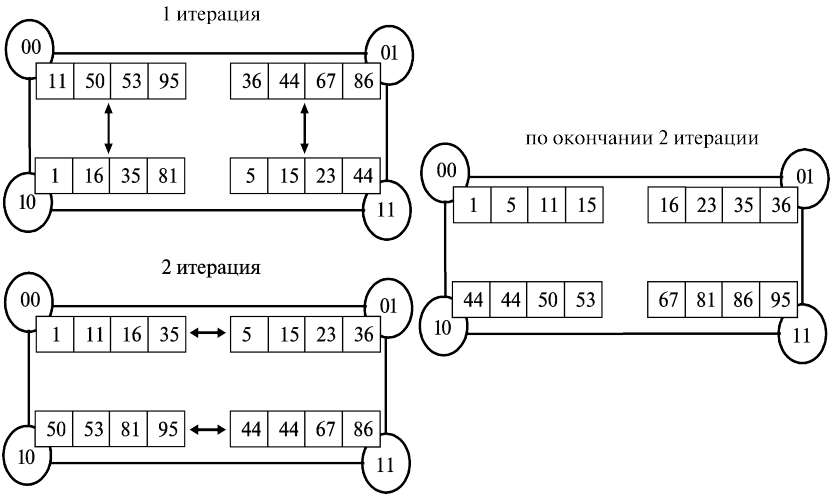


Рис. 9.3. Пример работы алгоритма Шелла для 4 процессоров

С учетом представленного описания параллельного варианта алгоритма Шелла *базовая подзадача* для организации параллельных вычислений, как и ранее (см. п. 9.3.3), может быть определена на основе операции «сравнить и разделить». Как результат, количество подзадач всегда совпадает с числом имеющихся процессоров (размер блоков данных в подзадачах равен n/p) и не возникает проблемы масштабирования. Распределе-

ние блоков упорядочиваемого набора данных по процессорам должно быть выбрано с учетом возможности эффективного выполнения операций «сравнить и разделить» при представлении топологии сети передачи данных в виде гиперкуба.

9.4.3. Анализ эффективности

Для оценки эффективности параллельного аналога алгоритма Шелла могут быть использованы соотношения, полученные для параллельного метода пузырьковой сортировки (см. п. 9.3.5). При этом следует только учесть двухэтапность алгоритма Шелла – с учетом данной особенности общее время выполнения нового параллельного метода может быть определено при помощи выражения

$$T_p = (n/p)\log_2(n/p)\tau + (\log_2 p + L)[(2n/p)\tau + (\alpha + w\cdot(n/p)/\beta)]. \qquad (9.7)$$

Как можно заметить, эффективность параллельного варианта сортировки Шелла существенно зависит от значения L – при малом значении величины L новый параллельный способ сортировки выполняется быстрее, чем ранее рассмотренный алгоритм чет-нечетной перестановки.

9.4.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта сортировки Шелла осуществлялись при тех же условиях, что и ранее выполненные (см. п. 9.3.6).

Результаты вычислительных экспериментов приведены в табл. 9.4. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 9.4. Результаты вычислительных экспериментов для параллельного алгоритма сортировки Шелла

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,002959	0,480568	0,007509	0,189373
20 000	0,002991	0,004557	0,656353	0,009826	0,304396
30 000	0,004612	0,006118	0,753841	0,012431	0,371008
40 000	0,006297	0,008461	0,744238	0,017009	0,370216
50 000	0,008014	0,009920	0,807863	0,019419	0,412689

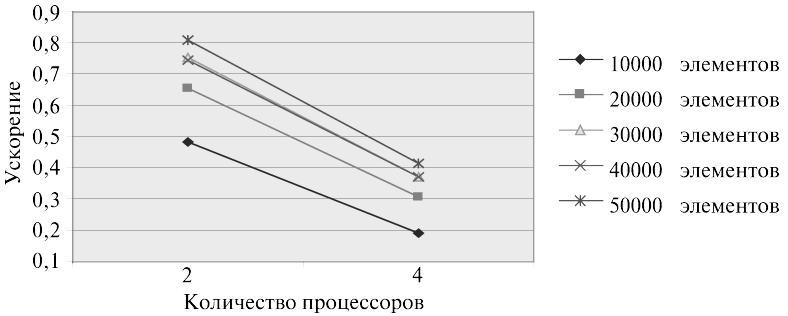


Рис. 9.4. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма сортировки Шелла

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (9.7) приведено в таблице 9.5 и на рис. 9.5.

Таблица 9.5. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сортировки Шелла

Количество элементов	**Параллельный алгоритм**			
	2 процессора		**4 процессора**	
	T_2	T_2^*	T_4	T_4^*
10 000	0,002684	0,002959	0,002938	0,007509
20 000	0,004872	0,004557	0,004729	0,009826
30 000	0,007100	0,006118	0,006538	0,012431
40 000	0,009353	0,008461	0,008361	0,017009
50 000	0,011625	0,009920	0,010193	0,019419

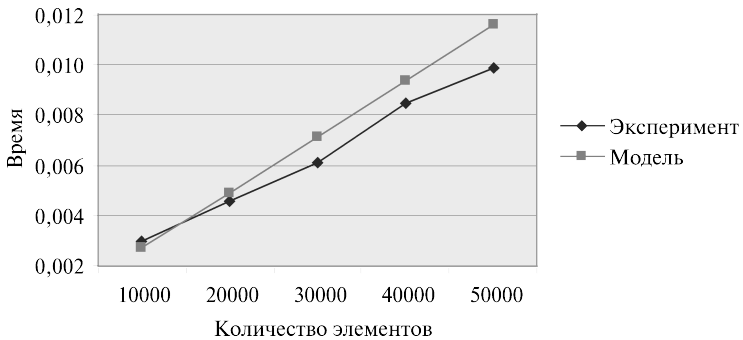


Рис. 9.5. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

9.5. Быстрая сортировка

9.5.1. Последовательный алгоритм

При общем рассмотрении *алгоритма быстрой сортировки* (*the quick sort algorithm*), предложенной Хоаром (*C.A.R. Hoare*), прежде всего следует отметить, что этот метод основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков все значения одного из этих блоков не превышают значений другого блока). На первой итерации метода осуществляется деление исходного набора данных на первые две части – для организации такого деления выбирается некоторый ведущий элемент и все значения набора, меньшие *ведущего элемента*, переносятся в первый формируемый блок, все остальные значения образуют второй блок набора. На второй итерации сортировки описанные правила применяются рекурсивно для обоих сформированных блоков и т. д. При надлежащем выборе ведущих элементов после выполнения $\log_2 n$ итераций исходный массив данных оказывается упорядоченным. Более подробное изложение метода может быть получено, например, в [26, 50].

Эффективность быстрой сортировки в значительной степени определяется правильностью выбора ведущих элементов при формировании блоков. В худшем случае трудоемкость метода имеет тот же порядок сложности, что и пузырьковая сортировка (т. е. $T_1 \sim n^2$). При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки ($T_1 = n\log_2 n$). В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением (см., например, [26, 50]):

$$T_1 = 1{,}4n\log_2 n.$$

Общая схема алгоритма быстрой сортировки может быть представлена в следующем виде (в качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных).

Алгоритм 9.5. Последовательный алгоритм быстрой сортировки

```
// Алгоритм 9.5
// Последовательный алгоритм быстрой сортировки
void QuickSort(double A[], int i1, int i2) {
```

```
  if (i1 < i2) {
    double pivot = A[i1];
    int is = i1;
    for (int i = i1 + 1; i < i2; i++)
      if (A[i] ≤ pivot) {
        is = is + 1;
        swap(A[is], A[i]);
      }
    swap(A[i1], A[is]);
    QuickSort(A, i1,      is);
    QuickSort(A, is + 1, i2);
  }
}
```

9.5.2. Параллельный алгоритм быстрой сортировки

9.5.2.1. Организация параллельных вычислений

Параллельное обобщение алгоритма быстрой сортировки (см., например, [63]) наиболее простым способом может быть получено, если топология коммуникационной сети может быть эффективно представлена в виде N-мерного гиперкуба (т. е. $p=2^N$). Пусть, как и ранее, исходный набор данных распределен между процессорами блоками одинакового размера n/p; результирующее расположение блоков должно соответствовать нумерации процессоров гиперкуба. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать каким-либо образом ведущий элемент и разослать его по всем процессорам системы (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем процессоре);
- разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
- образовать пары процессоров, для которых битовое представление номеров отличается только в позиции N, и осуществить взаимообмен данными между этими процессорами.

В результате выполнения такой итерации сортировки исходный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение ведущего элемента) располагается на процессорах, в битовом представлении номеров которых бит N равен 0. Таких процессоров всего $p/2$, и, таким образом, исходный N-мерный гиперкуб также оказывается разделенным на два гиперкуба размерности N-1. К

этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После N-кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы.

Для пояснения на рис. 9.6 представлен пример упорядочивания данных при n=16, p=4 (т. е. блок каждого процессора содержит 4 элемента). На этом рисунке процессоры изображены в виде прямоугольников, внутри которых показано содержимое упорядочиваемых блоков данных; значения блоков приводятся в начале и при завершении каждой итерации сортировки. Взаимодействующие пары процессоров соединены двунаправленными стрелками. Для разделения данных выбирались наилучшие значения ведущих элементов: на первой итерации для всех процессоров использовалось значение 0, на второй итерации для пары процессоров 0, 1 ведущий элемент равен -5, для пары процессоров 2, 3 это значение было принято равным 4.

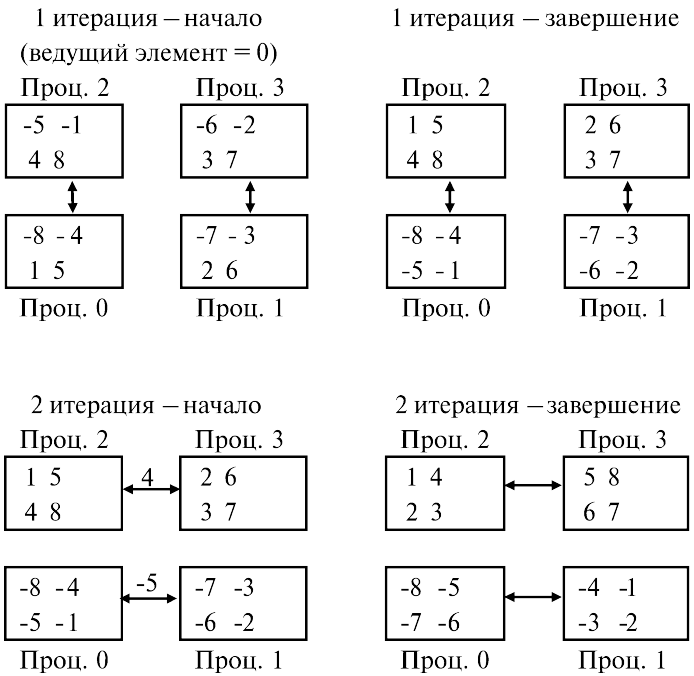


Рис. 9.6. Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

Как и ранее, в качестве *базовой подзадачи* для организации параллельных вычислений может быть выбрана операция «сравнить и разделить», а количество подзадач совпадает с числом используемых процессоров. Распределение подзадач по процессорам должно производиться с учетом возможности эффективного выполнения алгоритма при представлении топологии сети передачи данных в виде гиперкуба.

9.5.2.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть у нас имеется N-мерный гиперкуб, состоящий из $p=2^N$ процессоров, где $p<n$.

Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от правильности выбора значений ведущих элементов. Определение общего правила для выбора этих значений представляется затруднительным. Сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между процессорами вычислительной системы.

Определим вначале вычислительную сложность алгоритма сортировки. На каждой из $\log_2 p$ итераций сортировки каждый процессор осуществляет деление блока относительно ведущего элемента, сложность этой операции составляет n/p операций (будем предполагать, что на каждой итерации сортировки каждый блок делится на равные по размеру части).

При завершении вычислений процессор выполняет сортировку своих блоков, что может быть выполнено при использовании быстрых алгоритмов за $(n/p)\log_2 (n/p)$ операций.

Таким образом, общее время вычислений параллельного алгоритма быстрой сортировки составляет

$$T_p(calc) = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau, \qquad (9.8)$$

где τ есть время выполнения базовой операции перестановки.

Рассмотрим теперь сложность выполняемых коммуникационных операций. Общее количество межпроцессорных обменов для рассылки ведущего элемента на N-мерном гиперкубе может быть ограничено оценкой

$$\sum_{i=1}^{N} i = N(N+1)/2 = \log_2 p\,(\log_2 p + 1)/2 \sim (\log_2 p)^2. \qquad (9.9)$$

При используемых предположениях (выбор ведущих элементов осуществляется наилучшим образом) количество итераций алгоритма равно

$\log_2 p$, а объем передаваемых данных между процессорами всегда равен половине блока, т. е. величине $(n/p)/2$. При таких условиях коммуникационная сложность параллельного алгоритма быстрой сортировки определяется при помощи соотношения:

$$T_p(comm) = (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta), \quad (9.10)$$

где α – латентность, β – пропускная способность сети, а w есть размер элемента набора в байтах.

С учетом всех полученных соотношений общая трудоемкость алгоритма оказывается равной

$$T_p = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \quad (9.11)$$

9.5.2.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта быстрой сортировки производились при тех же условиях, что и ранее выполненные эксперименты (см. п. 9.3.6).

Результаты вычислительных экспериментов приведены в табл. 9.6. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 9.6. Результаты вычислительных экспериментов по исследованию параллельного алгоритма быстрой сортировки

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,001521	0,934911	0,003434	0,414094
20 000	0,002991	0,002234	1,338854	0,004094	0,730581
30 000	0,004612	0,003080	1,497403	0,005088	0,906447
40 000	0,006297	0,004363	1,443273	0,005906	1,066204
50 000	0,008014	0,005486	1,460809	0,006635	1,207837

Как можно заметить по результатам вычислительных экспериментов, параллельный алгоритм быстрой сортировки уже позволяет получить ускорение при решении задачи упорядочивания данных.

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (9.11) приведено в таблице 9.7 и на рис. 9.8.

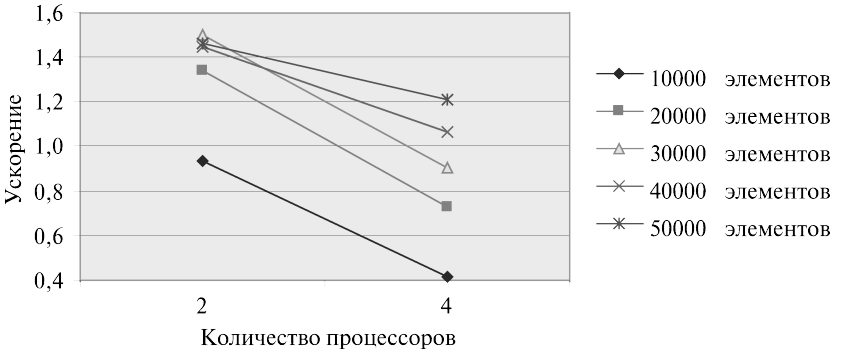


Рис. 9.7. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма быстрой сортировки

Таблица 9.7. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма быстрой сортировки

Количество элементов	**Параллельный алгоритм**			
	2 процессора		**4 процессора**	
	T_2	T_2^*	T_4	T_4^*
10 000	0,001280	0,001521	0,001735	0,003434
20 000	0,002265	0,002234	0,002321	0,004094
30 000	0,003289	0,003080	0,002928	0,005088
40 000	0,004338	0,004363	0,003547	0,005906
50 000	0,005407	0,005486	0,004175	0,006635

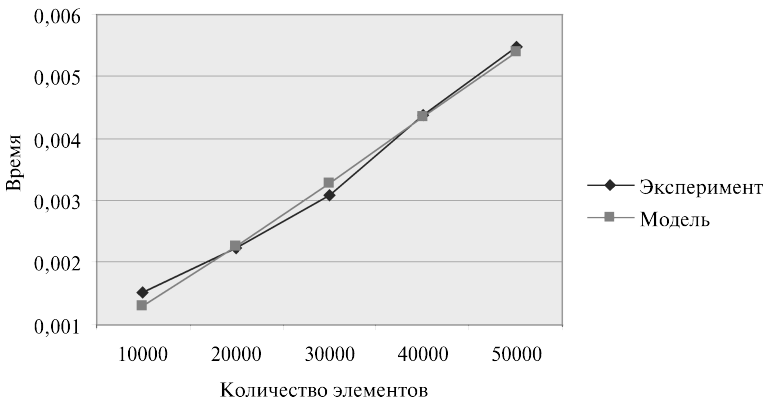


Рис. 9.8. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

9.5.3. Обобщенный алгоритм быстрой сортировки

В *обобщенном алгоритме быстрой сортировки* (*the HyperQuickSort algorithm*) в дополнение к обычному методу быстрой сортировки предлагается конкретный способ выбора ведущих элементов. Суть предложения состоит в том, что сортировка располагаемых на процессорах блоков происходит в самом начале выполнения вычислений. Кроме того, для поддержки упорядоченности в ходе вычислений процессоры должны выполнять операции слияния частей блоков, получаемых после разделения. Как результат, в силу упорядоченности блоков, при выполнении алгоритма быстрой сортировки в качестве ведущего элемента целесообразнее будет выбирать средний элемент какого-либо блока (например, на первом процессоре вычислительной системы). Выбираемый подобным образом ведущий элемент в отдельных случаях может оказаться более близким к реальному среднему значению всего сортируемого набора, чем какое-либо другое произвольно выбранное значение.

Все остальные действия в новом рассматриваемом алгоритме выполняются в соответствии с обычным методом быстрой сортировки. Более подробное описание данного способа распараллеливания быстрой сортировки может быть получено, например, в [63].

При анализе эффективности обобщенного алгоритма можно воспользоваться соотношением (9.11). Следует только учесть, что на каждой итерации метода теперь выполняется операция слияния частей блоков (будем, как и ранее, предполагать, что их размер одинаков и равен $(n/p)/2$). Кроме того, в силу упорядоченности блоков может быть усовершенствована процедура деления – вместо перебора всех элементов блока теперь достаточно будет выполнить для ведущего элемента бинарный поиск в блоке. С учетом всех высказанных замечаний трудоемкость обобщенного алгоритма быстрой сортировки может быть выражена при помощи следующего выражения:

$$T_p = [(n/p)\log_2(n/p) + (\log_2(n/p) + (n/p))\log_2 p]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \quad (9.12)$$

9.5.3.1. Программная реализация

Представим возможный вариант параллельной программы обобщенной быстрой сортировки. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 9.1.
// Обобщенная быстрая сортировка
int ProcRank;        // Ранг текущего процесса
int ProcNum;         // Количество процессов
int main(int argc, char *argv[]) {
  double *pProcData;      // Блок данных процесса
  int     ProcDataSize;  // Размер блока данных

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);

  // Инициализация данных и их распределение между процессами
  ProcessInitialization(&pProcData, &ProcDataSize);

  // Параллельная сортировка
  ParallelHyperQuickSort(pProcData, ProcDataSize);

  // Завершение вычислений процесса
  ProcessTermination(pProcData, ProcDataSize);

  MPI_Finalize();
}
```

Функция `ProcessInitialization` определяет исходные данные решаемой задачи (размер сортируемого массива), выделяет память для хранения данных, осуществляет генерацию сортируемого массива (например, при помощи датчика случайных чисел) и распределяет его между процессами.

Функция `ProcessTermination` осуществляет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция `ParallelHyperQuickSort`. Функция производит параллельную быструю сортировку согласно рассмотренному алгоритму.

```
// Функция для выполнения обощенного алгоритма быстрой сортировки
void ParallelHyperQuickSort ( double *pProcData,
  int ProcDataSize) {
  MPI_Status status;
```

```
int CommProcRank;  // Ранг процессора, с которым выполняется
                   // взаимодействие
double *pData,     // Часть блока, остающаяся на процессоре
       *pSendData, // Часть блока, передаваемая процессору
                   // CommProcRank
       *pRecvData, // Часть блока, получаемая от процессора
                   // CommProcRank
       *pMergeData; // Блок данных, получаемый после слияния
int     DataSize, SendDataSize, RecvDataSize, MergeDataSize;
int HypercubeDim = (int)ceil(log(ProcNum)/log(2));
                   // размерность гиперкуба
int Mask = ProcNum;
double Pivot;

// Первоначальная сортировка блоков данных на каждом процессоре
LocalDataSort(pProcData, ProcDataSize);

// Итерации обобщенной быстрой сортировки
for (int i = HypercubeDim; i > 0; i-- ) {

  // Определение ведущего значения и его рассылка всем процессорам
  PivotDistribution(pProcData, ProcDataSize, HypercubeDim,
    Mask, i,&Pivot);
  Mask = Mask >> 1;

  // Определение границы разделения блока
  int pos = GetProcDataDivisionPos(pProcData, ProcDataSize, Pivot);

  // Разделение блока на части
  if ( ( (rank & Mask) >> (i - 1) ) == 0 ) { // старший бит = 0
    pSendData = &pProcData[pos + 1];
    SendDataSize = ProcDataSize - pos - 1;
    if (SendDataSize < 0) SendDataSize = 0;
    CommProcRank = ProcRank + Mask
    pData = &pProcData[0];
    DataSize = pos + 1;
  }
  else { // старший бит = 1
    pSendData = &pProcData[0];
    SendDataSize = pos + 1;
    if (SendDataSize > ProcDataSize) SendDataSize = pos;
    CommProcRank = ProcRank - Mask
```

```
        pData = &pProcData[pos + 1];
        DataSize = ProcDataSize - pos - 1;
        if (DataSize < 0) DataSize = 0;
      }
      // Пересылка размеров частей блоков данных
      MPI_Sendrecv(&SendDataSize, 1, MPI_INT, CommProcRank, 0,
        &RecvDataSize, 1, MPI_INT, CommProcRank, 0, MPI_COMM_WORLD,
        &status);

      // Пересылка частей блоков данных
      pRecvData = new double[RecvDataSize];
      MPI_Sendrecv(pSendData, SendDataSize, MPI_DOUBLE,
        CommProcRank, 0, pRecvData, RecvDataSize, MPI_DOUBLE,
        CommProcRank, 0, MPI_COMM_WORLD, &status);

      // Слияние частей
      MergeDataSize = DataSize + RecvDataSize;
      pMergeData = new double[MergeDataSize];
      DataMerge(pMergeData, pMergeData, pData, DataSize,
        pRecvData, RecvDataSize);
      delete [] pProcData;
      delete [] pRecvData;
      pProcData = pMergeData;
      ProcDataSize = MergeDataSize;
    }
  }
```

Функция `LocalDataSort` выполняет сортировку блока данных на каждом процессоре, используя последовательный алгоритм быстрой сортировки.

Функция `PivotDistribution` определяет ведущий элемент и рассылает его значение всем процессорам.

Функция `GetProcDataDivisionPos` выполняет разделение блока данных относительно ведущего элемента. Ее результатом является целое число, обозначающее позицию элемента на границе двух блоков.

Функция `DataMerge` осуществляет слияние частей в один упорядоченный блок данных.

3. Функция `PivotDistribution`. Функция выбирает ведущий элемент и рассылает его все процессорам гиперкуба. Так как данные на процессорах отсортированы с самого начала, ведущий элемент выбирается как средний элемент блока данных.

```
// Функция выбора и рассылки ведущего элемента
void PivotDistribution (double *pProcData, int ProcDataSize, int Dim,
  int Mask, int Iter, double *pPivot) {
  MPI_Group WorldGroup;
  MPI_Group SubcubeGroup; // Группа процессов – подгиперкуб
  MPI_Comm  SubcubeComm;  // Коммуникатор подгиперкуба
  int j = 0;

  int  GroupNum = ProcNum /(int)pow(2, Dim-Iter);
  int *ProcRanks = new int [GroupNum];

  // формирование списка рангов процессов для гиперкуба
  int StartProc = ProcRank - GroupNum;
  if (StartProc < 0) StartProc = 0;
  int EndProc = ProcRank + GroupNum;
  if (EndProc > ProcNum) EndProc = ProcNum;
  for (int proc = StartProc; proc < EndProc; proc++) {
    if ((ProcRank & Mask)>>(Iter) == (proc & Mask)>>(Iter)) {
      ProcRanks[j++] = proc;
    }
  }
  // Объединение процессов подгиперкуба в одну группу
  MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
  MPI_Group_incl(WorldGroup, GroupNum, ProcRanks, &SubcubeGroup);
  MPI_Comm_create(MPI_COMM_WORLD, SubcubeGroup, &SubcubeComm);

  // Поиск и рассылка ведущего элемента всем процессам подгиперкуба
  if (ProcRank == ProcRanks[0])
    *pPivot = pProcData[ProcDataSize / 2];

  MPI_Bcast(pPivot, 1, MPI_DOUBLE, 0, SubcubeComm);
  MPI_Group_free(&SubcubeGroup);
  MPI_Comm_free(&SubcubeComm);
  delete [] ProcRanks;
}
```

9.5.3.2. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта обобщенной быстрой сортировки производились при тех же условиях, что и ранее выполненные (см. п. 9.3.6).

Результаты вычислительных экспериментов даны в табл. 9.8. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 9.8. Результаты вычислительных экспериментов для параллельного алгоритма обобщенной быстрой сортировки

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,001485	0,957576	0,002898	0,490683
20 000	0,002991	0,002180	1,372018	0,003770	0,793369
30 000	0,004612	0,003077	1,498863	0,004451	1,036172
40 000	0,006297	0,003859	1,631770	0,004721	1,333828
50 000	0,008014	0,005041	1,589764	0,005242	1,528806

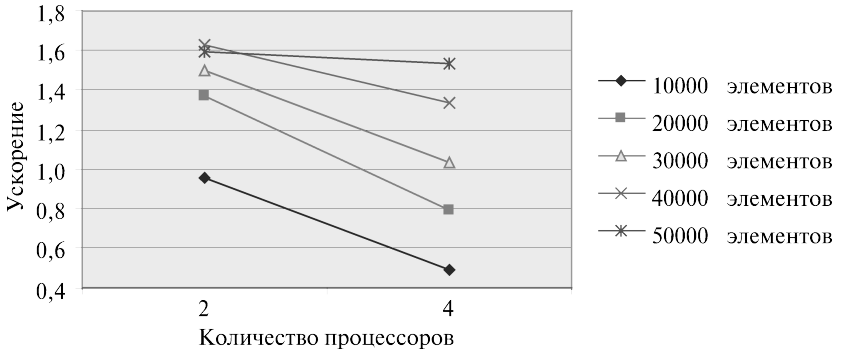


Рис. 9.9. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма обобщенной быстрой сортировки

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (9.12) приведено в таблице 9.9 и на рис. 9.10.

Таблица 9.9. Сравнение экспериментального и теоретического времен выполнения параллельного алгоритма обобщенной быстрой сортировки

Количество элементов	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,001281	0,001485	0,001735	0,002898
20 000	0,002265	0,002180	0,002322	0,003770
30 000	0,003289	0,003077	0,002928	0,004451
40 000	0,004338	0,003859	0,003547	0,004721
50 000	0,005407	0,005041	0,004176	0,005242

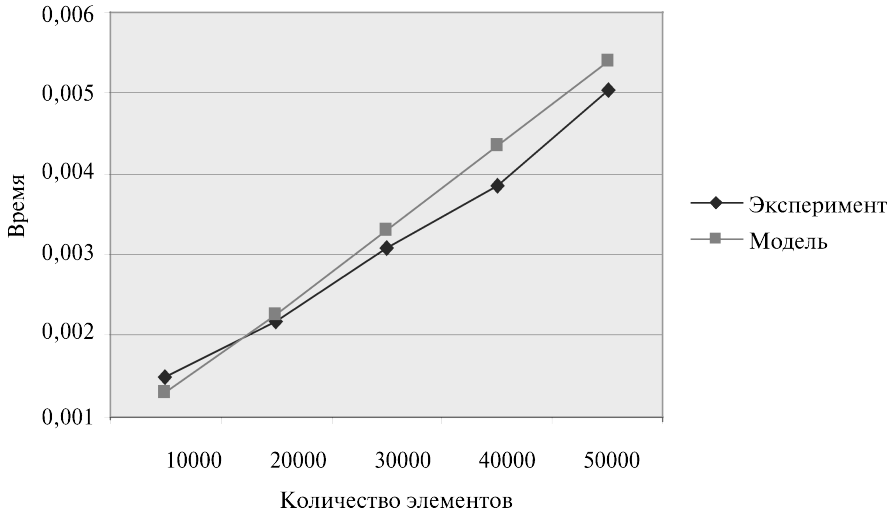


Рис. 9.10. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных

9.5.4. Сортировка с использованием регулярного набора образцов

9.5.4.1. Организация параллельных вычислений

Алгоритм сортировки с использованием регулярного набора образцов (*the parallel sorting by regular sampling*) также является обобщением метода быстрой сортировки (см., например, в [63]).

Упорядочивание данных в соответствии с данным вариантом алгоритма быстрой сортировки осуществляется в ходе выполнения следующих четырех этапов:

- на *первом этапе* сортировки производится упорядочивание имеющихся на процессорах блоков. Данная операция может быть выполнена каждым процессором независимо друг от друга при помощи обычного алгоритма быстрой сортировки; далее каждый процессор формирует набор из элементов своих блоков с индексами 0, m, $2m$,...,$(p-1)m$, где $m=n/p^2$;
- на *втором этапе* выполнения алгоритма все сформированные на процессорах наборы данных собираются на одном из процессоров системы и объединяются в ходе последовательного слияния в одно

упорядоченное множество. Далее из полученного множества значений из элементов с индексами

$$p+\lfloor p/2\rfloor-1,\quad 2p+\lfloor p/2\rfloor-1,\ldots,(p-1)p+\lfloor p/2\rfloor$$

формируется новый набор ведущих элементов, который передается всем используемым процессорам. В завершение этапа каждый про-

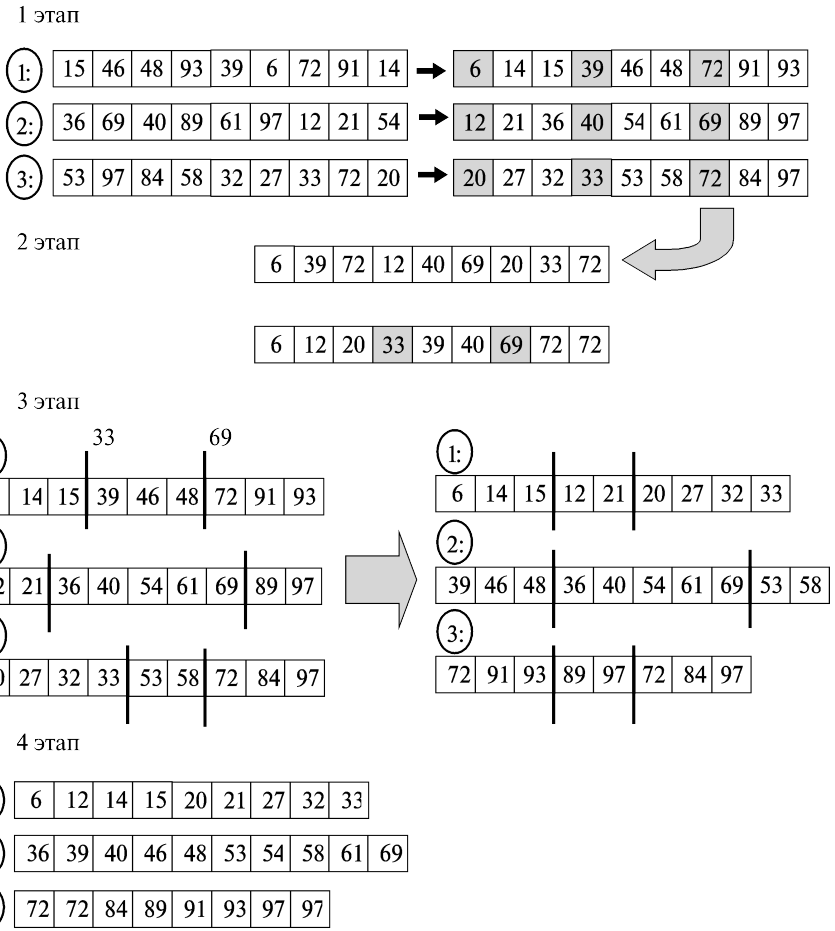


Рис. 9.11. Пример работы алгоритма сортировки с использованием регулярного набора образцов

цессор выполняет разделение своего блока на p частей с использованием полученного набора ведущих значений;

- на *третьем этапе* сортировки каждый процессор осуществляет рассылку выделенных ранее частей своего блока всем остальным процессорам системы; рассылка выполняется в соответствии с порядком нумерации – часть j, $0 \le j < p$, каждого блока пересылается процессору с номером j;
- на *четвертом этапе* выполнения алгоритма каждый процессор выполняет слияние p полученных частей в один отсортированный блок.

По завершении четвертого этапа исходный набор данных становится отсортированным.

На рис. 9.11 приведен пример сортировки массива данных с помощью алгоритма, описанного выше. Следует отметить, что число процессоров для данного алгоритма может быть произвольным, в данном примере оно равно 3.

9.5.4.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть, как и ранее, n есть количество сортируемых данных, p, $p<n$, обозначает число используемых процессоров и, соответственно, n/p есть размер блоков данных на процессорах.

В течение первого этапа алгоритма каждый процессор сортирует свой блок данных с помощью быстрой сортировки, тем самым, длительность выполняемых при этом операций равна

$$T_p^1 = (n/p)\log_2(n/p)\tau, \qquad (9.13)$$

где τ есть время выполнения базовой операции сортировки.

На втором этапе алгоритма один из процессоров собирает наборы из p элементов со всех остальных процессоров, выполняет слияние всех полученных данных (общее количество элементов составляет p^2), формирует набор из p-1 ведущих элементов и рассылает полученный набор всем остальным процессорам. С учетом всех перечисленных действий общая длительность второго этапа составляет

$$T_p^2 = [\alpha \log_2 p + wp(p-1)/\beta] + [p^2 \log_2 p\tau] + [p\tau] + [\log_2 p\,(\alpha + wp/\beta)] \qquad (9.14)$$

(в приведенном соотношении выделенные подвыражения соответствуют четырем перечисленным действиям алгоритма); здесь, как и ранее, α –

латентность, β – пропускная способность сети передачи данных, а w есть размер элемента упорядочиваемых данных в байтах.

В ходе выполнения третьего этапа алгоритма каждый процессор разделяет свои элементы относительно ведущих элементов на p частей (общее количество операций для этого может быть ограничено величиной n/p). Далее все процессоры выполняют рассылку сформированных частей блоков между собой – оценка трудоемкости такой коммуникационной операции рассмотрена в лекции 3 при представлении топологии вычислительной сети в виде гиперкуба. Как было показано, выполнение такой операции может быть осуществлено за $\log_2 p$ шагов, на каждом из которых каждый процессор передает и получает сообщение из $(n/p)/2$ элементов. Как результат, общая трудоемкость третьего этапа алгоритма может быть оценена как

$$T_p^3 = (n/p)\tau + \log_2 p(\alpha + w(n/2p)/\beta). \qquad (9.15)$$

На четвертом этапе алгоритма каждый процессор выполняет слияние p отсортированных частей в один объединенный блок. Оценка трудоемкости такой операции уже проводилась при рассмотрении второго этапа, и, тем самым, длительность выполнения процедуры слияния составляет

$$T_p^4 = (n/p)\log_2 p\tau. \qquad (9.16)$$

С учетом всех полученных соотношений общее время выполнения алгоритма сортировки с использованием регулярного набора образцов составляет

$$\begin{aligned} T_p \;=\; & (n/p)\log_2(n/p)\tau + (\alpha\log_2 p + wp(p-1)/\beta) + p^2\log_2 p\tau + (n/p)\tau + \\ & + \log_2 p(\alpha + wp/\beta) + p\tau + \log_2 p(\alpha + w(n/2p)/\beta) + (n/p)\log_2 p\tau. \end{aligned} \qquad (9.17)$$

9.5.4.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта сортировки с использованием регулярного набора образцов осуществлялись при тех же условиях, что и ранее выполненные (см. п. 9.3.6).

Результаты вычислительных экспериментов даны в табл. 9.10. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 9.10. Результаты вычислительных экспериментов для параллельного алгоритма сортировки с использованием регулярного набора образцов

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,001513	0,939855	0,001166	1,219554
20 000	0,002991	0,002307	1,296489	0,002081	1,437290
30 000	0,004612	0,003168	1,455808	0,003099	1,488222
40 000	0,006297	0,004542	1,386394	0,003819	1,648861
50 000	0,008014	0,005503	1,456297	0,004370	1,833867

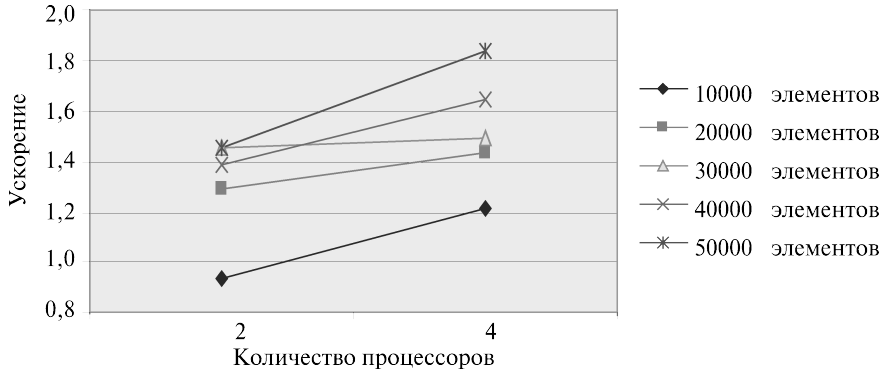


Рис. 9.12. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма сортировки с использованием регулярного набора образцов

Таблица 9.11. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сортировки с использованием регулярного набора образцов

Количество элементов	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,001533	0,001513	0,001762	0,001166
20 000	0,002569	0,002307	0,002375	0,002081
30 000	0,003645	0,003168	0,003007	0,003099
40 000	0,004747	0,004542	0,003652	0,003819
50 000	0,005867	0,005503	0,004307	0,004370

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (9.17) приведено в таблице 9.11 и на рис. 9.13.

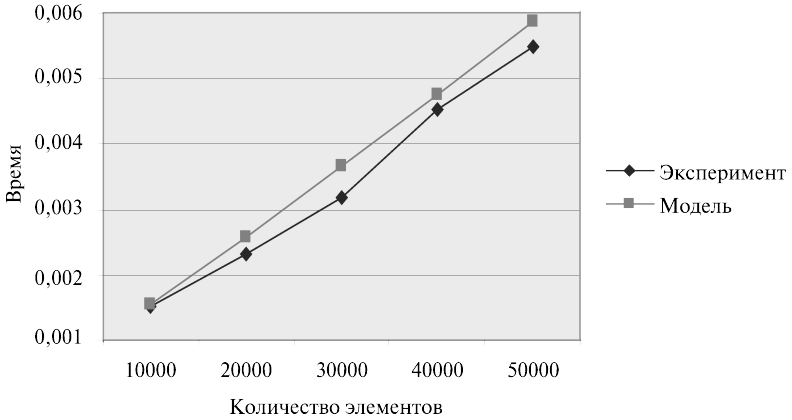


Рис. 9.13. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных

9.6. Краткий обзор лекции

В лекции рассматривается часто встречающаяся в приложениях *задача упорядочения данных*, для решения которой в рамках данного учебного материала выбраны широко известные алгоритмы пузырьковой сортировки, сортировки Шелла и быстрой сортировки. При изложении методов сортировки основное внимание уделяется возможным способам распараллеливания алгоритмов, анализу эффективности и сравнению получаемых теоретических оценок с результатами выполненных вычислительных экспериментов.

Алгоритм пузырьковой сортировки (подраздел 9.3) в исходном виде практически не поддается распараллеливанию в силу последовательного выполнения основных итераций метода. Для введения необходимого параллелизма рассматривается обобщенный вариант алгоритма – *метод чет-нечетной перестановки*. Суть обобщения состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода в зависимости от четности номера итерации сортировки. Сравнения пар значений упорядочиваемого набора данных на итерациях метода чет-нечетной перестановки являются независимыми и могут быть выполнены параллельно.

Для *алгоритма Шелла* (подраздел 9.4) рассматривается схема распараллеливания при представлении топологии сети в виде гиперкуба. При таком представлении топологии оказывается возможной организация взаимодействия процессоров, расположенных далеко друг от друга при линейной нумерации. Как правило, такая организация вычислений позволяет уменьшить количество выполняемых итераций алгоритма сортировки.

Для *алгоритма быстрой сортировки* (подраздел 9.5) приводятся три схемы распараллеливания. Первые две схемы также основываются на представлении топологии сети в виде гиперкуба. Основная итерация вычислений состоит в выборе одним из процессоров ведущего элемента, который далее рассылается всем остальным процессорам гиперкуба. После получения ведущего элемента процессоры проводят разделение своих блоков, и получаемые части блоков передаются между попарно связанными процессорами. В результате выполнения подобной итерации исходный гиперкуб оказывается разделенным на 2 гиперкуба меньшей размерности, к которым, в свою очередь, может быть применена приведенная выше схема вычислений.

При применении алгоритма быстрой сортировки одним из основных моментов является правильность выбора ведущего элемента. Оптимальная стратегия состоит в выборе такого значения ведущего элемента, при котором данные на процессорах разделяются на части одинакового размера. В общем случае при произвольно сгенерированных исходных данных достижение такой ситуации является достаточно сложной задачей. В первой схеме предлагается выбирать ведущий элемент, например, как среднее арифметическое элементов на управляющем процессоре. Во второй схеме (*обобщенный алгоритм быстрой сортировки*) данные на каждом процессоре предварительно упорядочиваются с тем, чтобы взять средний элемент блока данных как ведущее значение. Третья схема (*сортировка с использованием регулярного набора образцов*) распараллеливания алгоритма быстрой сортировки основывается на многоуровневой схеме формирования множества ведущих элементов. Такой подход может быть применен для произвольного количества процессоров, позволяет избежать множества пересылок данных и приводит, как правило, к более равномерному распределению данных между процессорами.

9.7. Обзор литературы

Возможные способы решения задачи упорядочения данных широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов* сортировки содержится в работе [50], среди последних изданий может быть рекомендована работа [26].

Параллельные варианты *алгоритма пузырьковой сортировки* и сортировки Шелла рассматриваются в [51].

Схемы распараллеливания *быстрой сортировки* при представлении топологии сети передачи данных в виде гиперкуба описаны в [51, 63]. *Сортировка с использованием регулярного набора образцов* представлена в работе [63].

Полезной при рассмотрении вопросов параллельных вычислений для сортировки данных может оказаться работа [17].

9.8. Контрольные вопросы

1. В чем состоит постановка задачи сортировки данных?
2. Приведите несколько примеров алгоритмов сортировки. Какова вычислительная сложность приведенных алгоритмов?
3. Какая операция является базовой для задачи сортировки данных?
4. В чем суть параллельного обобщения базовой операции задачи сортировки данных?
5. Что представляет собой алгоритм чет-нечетной перестановки?
6. В чем состоит параллельный вариант алгоритма Шелла? Каковы основные отличия этого параллельного алгоритма сортировки от метода чет-нечетной перестановки?
7. Что представляет собой параллельный вариант алгоритма быстрой сортировки?
8. Что зависит от правильного выбора ведущего элемента для параллельного алгоритма быстрой сортировки?
9. Какие способы выбора ведущего элемента могут быть предложены?
10. Для каких топологий могут применяться рассмотренные алгоритмы сортировки?
11. В чем состоит алгоритм сортировки с использованием регулярного набора образцов?

9.9. Задачи и упражнения

1. Выполните реализацию параллельного алгоритма пузырьковой сортировки. Проведите эксперименты. Постройте теоретические оценки с учетом тех операций пересылок данных, которые использовались при реализации, и параметров вычислительной системы. Сравните получаемые теоретические оценки с результатами экспериментов.
2. Выполните реализацию параллельного алгоритма быстрой сортировки по одной из приведенных схем. Определите значения параметров латентности, пропускной способности и времени выполнения базовой операции для используемой вычислительной сис-

темы и получите оценки показателей ускорения и эффективности для реализованного метода параллельных вычислений.

3. Разработайте параллельную схему вычислений для широко известного алгоритма сортировки слиянием (подробное описание метода может быть получено, например, в работах [26] или [50]). Выполните реализацию разработанного алгоритма и постройте все необходимые теоретические оценки сложности метода.

Лекция 10. Параллельные методы на графах

В лекции рассматриваются различные типовые задачи, возникающие при обработке графов. Приводятся алгоритмы, применяемые для решения этих задач, и обсуждаются пути их распараллеливания. Дается теоретическая оценка эффективности рассматриваемых алгоритмов. Анализируются результаты вычислительных экспериментов.

Ключевые слова: граф, задача поиска всех кратчайших путей, задача нахождения минимального охватывающего дерева, задача оптимального разделения графов, алгоритм Флойда, алгоритм Прима, геометрические и комбинаторные алгоритмы разделения графов, оценка сложности, ускорение и эффективность параллельных вычислений.

Математические модели в виде *графов* широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор *типовых алгоритмов обработки графов*. Рассмотрению вопросов теории графов, алгоритмов моделирования, анализу и решению задач на графах посвящено достаточно много различных изданий, в качестве возможного руководства по данной тематике может быть рекомендована работа [26].

Пусть G есть граф

$$G=(V,R),$$

для которого набор вершин V_i, $0 \le i \le n$, задается множеством V, а список дуг графа

$$r_j = (v_{s_j}, v_{t_j}),\ 1 \le j \le m$$

определяется множеством R. В общем случае дугам графа могут приписываться некоторые числовые характеристики (*веса*) w_j, $0 \le j \le m$ (*взвешенный граф*). Пример взвешенного графа приведен на рис. 10.1.

Известны различные способы задания графов. При малом количестве дуг в графе (т. е. $m << n^2$) целесообразно использовать для определения графов списки, перечисляющие имеющиеся в графах дуги. Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т. е. $m \sim n^2$), может быть эффективно обеспечено при помощи *матрицы смежности*:

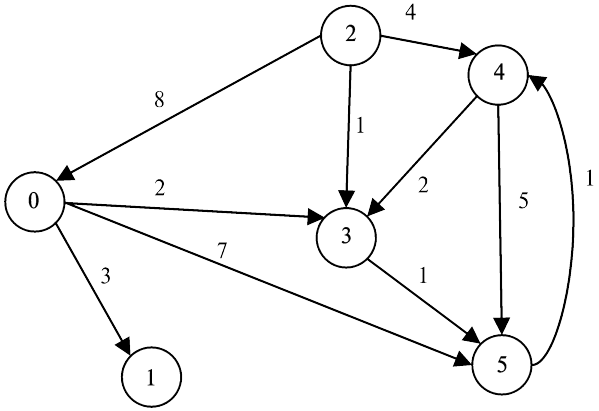


Рис. 10.1. Пример взвешенного ориентированного графа

$A=(a_{ij})$, $1 \le i, j \le n$,

ненулевые значения элементов которой соответствуют дугам графа:

$$a_{ij} = \begin{cases} w(v_i, v_j), & \text{если } (v_i, v_j) \in R, \\ 0, & \text{если } i = j, \\ \infty, & \text{иначе} \end{cases}$$

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 10.1, приведена на рис. 10.2.

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рис. 10.2. Матрица смежности для графа с рис. 10.1

Как положительный момент такого способа представления графов можно отметить, что использование матрицы смежности позволяет при-

менять при реализации вычислительных процедур анализа графов матричные алгоритмы обработки данных.

Далее мы рассмотрим способы параллельной реализации алгоритмов на графах на примере *задачи поиска кратчайших путей* между всеми парами пунктов назначения и *задачи выделения минимального охватывающего дерева* (*остова*) графа. Кроме того, мы рассмотрим *задачу оптимального разделения графов*, широко используемую для организации параллельных вычислений. Для представления графов при рассмотрении всех перечисленных задач будут применяться матрицы смежности.

10.1. Задача поиска всех кратчайших путей

Исходной информацией для задачи является взвешенный граф $G=(V,R)$, содержащий n вершин ($|V|=n$), в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать *ориентированным*, т. е. если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае если вершины все же соединены взаимообратными ребрами, веса, приписанные им, могут не совпадать. Рассмотрим задачу, в которой для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

В качестве метода, решающего задачу поиска кратчайших путей между всеми парами пунктов назначения, далее используется *алгоритм Флойда* (*the Floyd algorithm*) (см, например, [26]).

10.1.1. Последовательный алгоритм Флойда

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок n^3. В общем виде данный алгоритм может быть представлен следующим образом:

Алгоритм 10.1. Общая схема алгоритма Флойда

```
// Алгоритм 10.1
// Последовательный алгоритм Флойда
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i, j] = min(A[i, j], A[i, k] + A[k, j]);
```

(реализация операции выбора минимального значения *min* должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

Дополнительная информация и доказательство правильности алгоритма Флойда могут быть получены, например, в работе [26].

10.1.2. Разделение вычислений на независимые части

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значений (см. Алгоритм 10.1). Данная операция является достаточно простой, и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A.

Покажем корректность такого способа организации параллелизма. Для этого нужно доказать, что операции обновления значений матрицы A на одной и той же итерации внешнего цикла k могут выполняться независимо. Иными словами, следует показать, что на итерации k не происходит изменения элементов A_{ik} и A_{kj} ни для одной пары индексов (i, j). Рассмотрим выражение, по которому происходит изменение элементов матрицы A:

$$A_{ij} \leftarrow min\,(A_{ij}, A_{ik} + A_{kj}).$$

Для $i=k$ получим

$$A_{kj} \leftarrow min\,(A_{kj}, A_{kk} + A_{kj}),$$

но тогда значение A_{kj} не изменится, т. к. $A_{kk}=0$.

Для $j=k$ выражение преобразуется к виду

$$A_{ik} \leftarrow min\,(A_{ik}, A_{ik} + A_{kk}),$$

что также показывает неизменность значений A_{ik}. Как результат, необходимые условия для организации параллельных вычислений обеспечены, и, тем самым, в качестве *базовой подзадачи* может быть использована операция обновления элементов матрицы A (для указания подзадач будем применять индексы обновляемых в подзадачах элементов).

10.1.3. Выделение информационных зависимостей

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача (i, j) содержит необходимые для расчетов элементы A_{ij}, A_{ik}, A_{kj} матрицы A. Для исключения дублирования данных разместим в подзадаче (i, j) единственный элемент A_{ij}, тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким образом, каждый элемент A_{kj} строки k матрицы A должен быть передан всем подзадачам (k, j), $1 \le j \le n$, а каждый элемент A_{ik} столбца k матрицы A должен быть передан всем подзадачам (i, k), $1 \le i \le n$,– см. рис. 10.3.

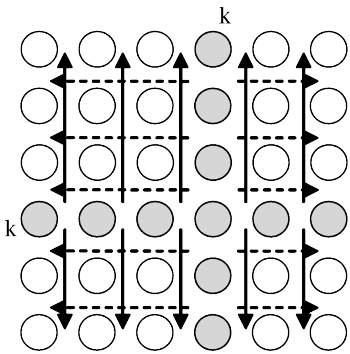


Рис. 10.3. Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

10.1.4. Масштабирование и распределение подзадач по процессорам

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p << n^2$). Возможный способ укрупнения вычислений состоит в использовании *ленточной схемы* разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк (*горизонтальное* разбиение) или столбцов (*вертикальное* разбиение) матрицы A. Эти два типа разбиения практически равноправны – учитывая дополнительный момент, что для алгоритмического языка С массивы располагаются по строкам, будем рассматривать далее только разбиение матрицы A на горизонтальные полосы.

Следует отметить, что при таком способе разбиения данных на каждой итерации алгоритма Флойда потребуется передавать между подзада-

чами только элементы одной из строк матрицы A. Для оптимального выполнения подобной коммуникационной операции топология сети должна обеспечивать эффективное представление структуры сети передачи данных в виде гиперкуба или полного графа.

10.1.5. Анализ эффективности параллельных вычислений

Выполним анализ эффективности параллельного алгоритма Флойда, обеспечивающего поиск всех кратчайших путей. Как и ранее, проведем этот анализ в два этапа. На первом оценим порядок вычислительной сложности алгоритма, затем на втором этапе уточним полученные оценки и учтем трудоемкость выполнения коммуникационных операций.

Общая трудоемкость последовательного алгоритма, как уже отмечалось ранее, имеет порядок сложности n^3. Для параллельного алгоритма на отдельной итерации каждый процессор выполняет обновление элементов матрицы A. Всего в подзадачах n^2/p таких элементов, число итераций алгоритма равно n – таким образом, показатели ускорения и эффективности параллельного алгоритма Флойда имеют вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \tag{10.1}$$

Следовательно, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений введем в полученные выражения время выполнения базовой операции выбора минимального значения и учтем затраты на выполнение операций передачи данных между процессорами.

Коммуникационная операция, выполняемая на каждой итерации алгоритма Флойда, состоит в передаче одной из строк матрицы A всем процессорам вычислительной системы. Как уже показывалось ранее, такая операция может быть выполнена за $\lceil \log_2 p \rceil$ шагов. С учетом количества итераций алгоритма Флойда при использовании модели Хокни общая длительность выполнения коммуникационных операций может быть определена при помощи следующего выражения

$$T_p(comm) = n\lceil \log_2 p \rceil(\alpha + wn/\beta), \tag{10.2}$$

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

С учетом полученных соотношений общее время выполнения параллельного алгоритма Флойда может быть определено следующим образом:

$$T_p = n^2 \cdot \lceil n/p \rceil \cdot \tau + n \cdot \lceil \log_2 p \rceil (\alpha + w \cdot n/\beta), \quad (10.3)$$

где τ есть время выполнения операции выбора минимального значения.

10.1.6. Программная реализация

Представим возможный вариант параллельной реализации алгоритма Флойда. При этом описание отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 10.1. – Алгоритм Флойда
int ProcRank;     // Ранг текущего процесса
int ProcNum;      // Количество процессов

// Функция вычисления минимума
int Min(int A, int B) {
  int Result = (A < B) ? A : B;

  if((A < 0) && (B >= 0)) Result = B;
  if((B < 0) && (A >= 0)) Result = A;
  if((A < 0) && (B < 0))  Result = -1;

  return Result;
}

// Главная функция программы
int main(int argc, char* argv[]) {
  int *pMatrix;      // Матрица смежности
  int  Size;         // Размер матрицы смежности
  int *pProcRows;    // Строки матрицы смежности текущего процесса
  int  RowNum;       // Число строк для текущего процесса

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  // Инициализация данных
  ProcessInitialization(pMatrix, pProcRows, Size, RowNum);

  // Распределение данных между процессами
```

```
  DataDistribution(pMatrix, pProcRows, Size, RowNum);

  // Параллельный алгоритм Флойда
  ParallelFloyd(pProcRows, Size, RowNum);

  // Сбор результатов работы алгоритма
  ResultCollection(pMatrix, pProcRows, Size, RowNum);

  // Завершение вычислений процесса
  ProcessTermination(pMatrix, pProcRows);

  MPI_Finalize();
  return 0;
}
```

Функция `Min` вычисляет меньшее из двух целых чисел, учитывая применяемый способ задания несуществующих дуг в матрице смежности (в рассматриваемой реализации для этого используется значение -1).

Функция `ProcessInitialization` определяет исходные данные решаемой задачи (количество вершин графа), выделяет память для хранения данных, осуществляет ввод матрицы смежности (или формирует ее при помощи какого-либо датчика случайных числе).

Функция `DataDistribution` распределяет исходные данные между процессами. Каждый процесс получает горизонтальную полосу матрицы смежности.

Функция `ResultCollection` осуществляет сбор со всех процессов горизонтальных полос результирующей матрицы кратчайших расстояний между любыми парами вершин графа.

Функция `ProcessTermination` выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция `ParallelFloyd`. Данная функция осуществляет итеративное изменение матрицы смежности в соответствии с алгоритмом Флойда.

```
// Параллельный алгоритм Флойда
void ParallelFloyd(int *pProcRows, int Size, int RowNum) {
  int *pRow = new int[Size];
  int t1, t2;

  for(int k = 0; k < Size; k++) {
```

```
    // Распределение k-й строки среди процессов
    RowDistribution(pProcRows, Size, RowNum, k, pRow);

    // Обновление элементов матрицы смежности
    for(int i = 0; i < RowNum; i++)
      for(int j = 0; j < Size; j++)
        if( (pProcRows[i * Size + k] != -1) && (pRow [j]!= -1)){
          t1 = pProcRows[i * Size + j];
          t2 = pProcRows[i * Size + k] + pRow[j];
          pProcRows[i * Size + j] = Min(t1, t2);
        }
  }

  delete []pRow;
}
```

3. Функция `RowDistribution`**.** Данная функция рассылает k-ю строку матрицы смежности всем процессам программы.

```
// Функция для рассылки строки всем процессам
void RowDistribution(int *pProcRows, int Size, int RowNum, int k,
  int *pRow) {
  int ProcRowRank; // Ранг процесса, которому принадлежит k-я строка
  int ProcRowNum; // Номер k-й строки в полосе матрицы

  // Нахождение ранга процесса - владельца k-й строки
  int RestRows = Size;
  int Ind = 0;
  int Num = Size / ProcNum;

  for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++)
  {
    if(k < Ind + Num ) break;
    RestRows  -= Num;
    Ind       += Num;
    Num        = RestRows / (ProcNum - ProcRowRank);
  }
  ProcRowRank = ProcRowRank - 1;
  ProcRowNum  = k - Ind;

  if(ProcRowRank == ProcRank)
```

```
      // Копирование строки в массив pRow
      copy(&pProcRows[ProcRowNum * Size], &pProcRows[(ProcRowNum + 1) *
        Size], pRow);

    // Распределение k-й строки между процессами
    MPI_Bcast(pRow, Size, MPI_INT, ProcRowRank, MPI_COMM_WORLD);
  }
```

10.1.7. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном кластере Нижегородского университета на базе процессоров Intel Xeon 4 EM64T, 3000 МГц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standard x64 Edition и системы управления кластером Microsoft Compute Cluster Server.

Для оценки длительности τ базовой скалярной операции выбора минимального значения проводилось решение задачи поиска всех кратчайших путей при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате для величины τ было получено значение 7,14 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 130 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа int, размер которого на данной платформе равен 4 байта (следовательно, w=4).

Результаты вычислительных экспериментов приведены в таблице 10.1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 10.1. Результаты вычислительных экспериментов для параллельного алгоритма Флойда

Кол-во вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	8,037	4,152	1,936	2,067	3,888	0,941	8,544
2000	59,812	30,323	1,972	15,375	3,890	8,058	7,423
3000	197,111	99,264	1,986	50,232	3,924	25,643	7,687
4000	461,785	232,507	1,986	117,220	3,939	69,946	6,602
5000	884,622	443,747	1,994	224,441	3,941	128,078	6,907

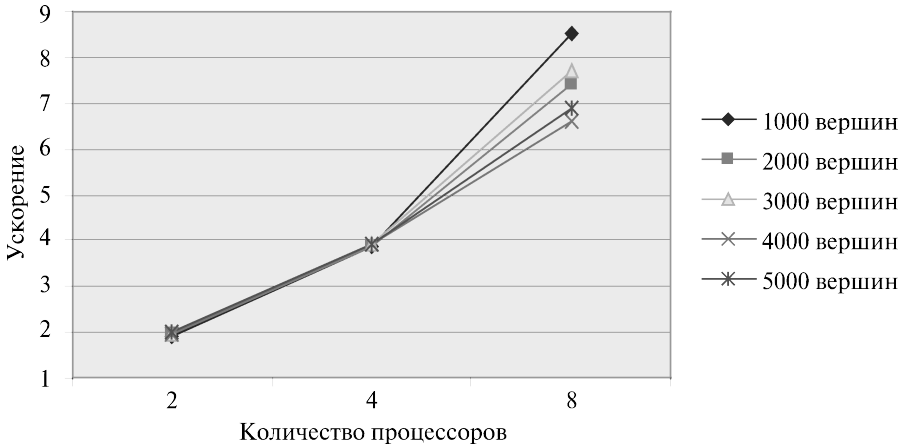


Рис. 10.4. Графики зависимости ускорения параллельного алгоритма Флойда от числа используемых процессоров при различном количестве вершин графа

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.3) приведено в таблице 10.2 и на рис. 10.5.

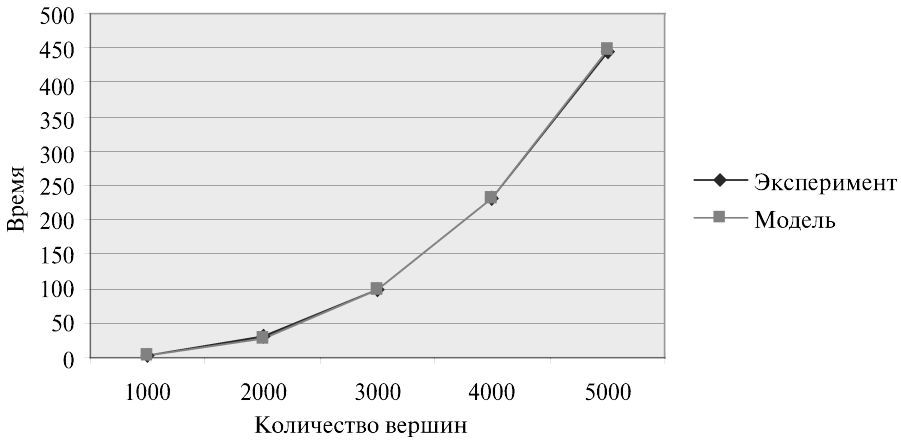


Рис. 10.5. Графики экспериментально установленного времени работы параллельного алгоритма Флойда и теоретической оценки в зависимости от количества вершин графа при использовании двух процессоров

Таблица 10.2. Сравнение экспериментального и теоретического времени работы алгоритма Флойда

Количество вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
	T_1^*	T_2	T_2^*	T_4	T_4^*	T_8	T_8^*
1000	8,037	3,776	4,152	2,196	2,067	1,509	0,941
2000	59,812	29,123	30,323	15,405	15,375	8,826	8,058
3000	197,111	97,465	99,264	50,336	50,232	27,307	25,643
4000	461,785	230,220	232,507	117,701	117,220	62,306	69,946
5000	884,622	448,811	443,747	228,211	224,441	119,179	128,078

10.2. Задача нахождения минимального охватывающего дерева

Охватывающим деревом (или *остовом*) неориентированного графа G называется подграф T графа G, который является деревом и содержит все вершины из G. Определив вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, под *минимально охватывающим деревом* (*МОД*) T будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием соединительных линий связи минимальной длины. Пример взвешенного неориентированного графа и соответствующего ему минимально охватывающего дерева приведен на рис. 10.6.

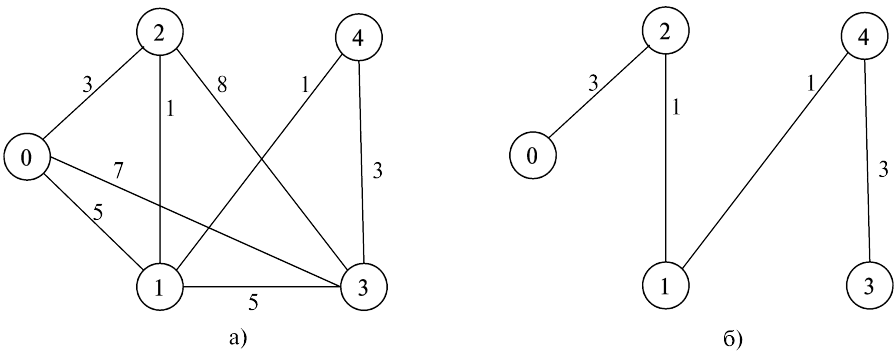


Рис. 10.6. Пример взвешенного неориентированного графа (а) и соответствующему ему минимально охватывающего дерева (б)

Дадим общее описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*the Prim method*); более полная информация может быть получена, например, в [26].

10.2.1. Последовательный алгоритм Прима

Алгоритм начинает работу с произвольной вершины графа, выбираемой в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Пусть V_T есть множество вершин, уже включенных алгоритмом в МОД, а величины d_i, $1 \le i \le n$, характеризуют дуги минимальной длины от вершин, еще не включенных в дерево, до множества V_T, т. е.

$$\forall i \notin V_T \Rightarrow d_i = \min\{w(i,u) : u \in V_T, (i,u) \in R\}$$

(если для какой-либо вершины $i \notin V_T$ не существует ни одной дуги в V_T, значение d_i устанавливается равным ∞). В начале работы алгоритма выбирается корневая вершина МОД s и полагается $V_T=\{s\}$, $d_s=0$.

Действия, выполняемые на каждой итерации алгоритма Прима, состоят в следующем:

- определяются значения величин d_i для всех вершин, еще не включенных в состав МОД;
- выбирается вершина t графа G, имеющая дугу минимального веса до множества V_T $t : d_t$, $i \notin V_T$;
- вершина t включается в V_T.

После выполнения n-1 итерации метода МОД будет сформировано. Вес этого дерева может быть получен при помощи выражения

$$W_T = \sum_{i=1}^{n} d_i \ .$$

Трудоемкость нахождения МОД характеризуется квадратичной зависимостью от числа вершин графа $T_1 \sim n^2$.

10.2.2. Разделение вычислений на независимые части

Оценим возможности параллельного выполнения рассмотренного алгоритма нахождения минимально охватывающего дерева.

Итерации метода должны выполняться последовательно и, тем самым, не могут быть распараллелены. С другой стороны, выполняемые на каждой итерации алгоритма действия являются независимыми и могут реализовываться одновременно. Так, например, определение величин d_i может осуществляться для каждой вершины графа в отдельности, нахож-

дение дуги минимального веса может быть реализовано по каскадной схеме и т. д.

Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть реализовано, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией. Соблюдение данного принципа приводит к тому, что при равномерной загрузке каждый процессор p_j, $1 \le j \le p$, должен содержать:

- набор вершин

$$V_j = \{v_{i_j+1}, v_{i_j+2}, ..., v_{i_j+k}\},\ i_j = k \cdot (j-1),\ k = \lceil n/p \rceil;$$

- соответствующий этому набору блок из k величин

$$\Delta_j = \{d_{i_j+1}, d_{i_j+2}, ..., d_{i_j+k}\};$$

- вертикальную полосу матрицы смежности графа G из k соседних столбцов

$$A_j = \{\alpha_{i_j+1}, \alpha_{i_j+2}, ..., \alpha_{i_j+k}\}$$ (α_s есть s-й столбец матрицы A);

- общую часть набора V_j и формируемого в процессе вычислений множества вершин V_T.

Как итог можем заключить, что базовой подзадачей в параллельном алгоритме Прима может служить процедура вычисления блока значений Δ_j для вершин V_j матрицы смежности A графа G.

10.2.3. Выделение информационных зависимостей

С учетом выбора базовых подзадач общая схема параллельного выполнения алгоритма Прима будет состоять в следующем:

- определяется вершина t графа G, имеющая дугу минимального веса до множества V_T. Для выбора такой вершины необходимо осуществить поиск минимума в наборах величин d_i, имеющихся на каждом из процессоров, и выполнить сборку полученных значений на одном из процессоров;
- номер выбранной вершины для включения в охватывающее дерево передается всем процессорам;
- обновляются наборы величин d_i с учетом добавления новой вершины.

Таким образом, в ходе параллельных вычислений между процессорами выполняются два типа информационных взаимодействий: сбор дан-

ных от всех процессоров на одном из процессоров и передача сообщений от одного процессора всем процессорам вычислительной системы.

10.2.4. Масштабирование и распределение подзадач по процессорам

По определению количество базовых подзадач всегда соответствует числу имеющихся процессоров, и, тем самым, проблема масштабирования для параллельного алгоритма не возникает.

Распределение подзадач между процессорами должно учитывать характер выполняемых в алгоритме Прима коммуникационных операций. Для оптимальной реализации требуемых информационных взаимодействий между базовыми подзадачами топология сети передачи данных должна обеспечивать эффективное представление в виде гиперкуба или полного графа.

10.2.5. Анализ эффективности параллельных вычислений

Общий анализ сложности параллельного алгоритма Прима для нахождения минимального охватывающего дерева дает идеальные показатели эффективности параллельных вычислений:

$$S_p = \frac{n^2}{(n^2/p)} = p \quad \text{и} \quad E_p = \frac{n^2}{p \cdot (n^2/p)} = 1 . \qquad (10.4)$$

При этом следует отметить, что в ходе параллельных вычислений идеальная балансировка вычислительной нагрузки процессоров может быть нарушена. В зависимости от вида исходного графа G количество выбранных вершин в охватывающем дереве на разных процессорах может оказаться различным, и распределение вычислений между процессорами станет неравномерным (вплоть до отсутствия вычислительной нагрузки на отдельных процессорах). Однако такие предельные ситуации нарушения балансировки в общем случае возникают достаточно редко, а организация динамического перераспределения вычислительной нагрузки между процессорами в ходе вычислений является интересной, но одновременно и очень сложной задачей.

Для уточнения полученных показателей эффективности параллельных вычислений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

При выполнении вычислений на отдельной итерации параллельного алгоритма Прима каждый процессор определяет номер ближайшей вершины из V_j до охватывающего дерева и осуществляет корректировку расстояний d_i после расширения МОД. Количество выполняемых операций в каждой из этих

вычислительных процедур ограничивается сверху числом вершин, имеющихся на процессорах, т. е. величиной $\lceil n/p \rceil$. Как результат, с учетом общего количества итераций n время выполнения вычислительных операций параллельного алгоритма Прима может быть оценено при помощи соотношения:

$$T_p(calc) = 2n\lceil n/p \rceil \cdot \tau \qquad (10.5)$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Операция сбора данных от всех процессоров на одном из процессоров может быть произведена за $\lceil \log_2 p \rceil$ итераций, при этом общая оценка длительности выполнения передачи данных определяется выражением (более подробное рассмотрение данной коммуникационной операции содержится в лекции 3):

$$T_p^1(comm) = n(\alpha \log_2 p + 3w(p-1)/\beta), \qquad (10.6)$$

где α – латентность сети передачи данных, β – пропускная способность сети, а w есть размер одного пересылаемого элемента данных в байтах (коэффициент 3 в выражении соответствует числу передаваемых значений между процессорами – длина минимальной дуги и номера двух вершин, которые соединяются этой дугой).

Коммуникационная операция передачи данных от одного процессора всем процессорам вычислительной системы также может быть выполнена за $\lceil \log_2 p \rceil$ итераций при общей оценке времени выполнения вида:

$$T_p^2(comm) = n \log_2 p(\alpha + w/\beta). \qquad (10.7)$$

С учетом всех полученных соотношений общее время выполнения параллельного алгоритма Прима составляет:

$$T_p = 2n\lceil n/p \rceil \cdot \tau + n(\alpha \cdot \log_2 p + 3w(p-1)/\beta + \log_2 p(\alpha + w/\beta)). \qquad (10.8)$$

10.2.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма Прима осуществлялись при тех же условиях, что и ранее выполненные (см. п. 10.1.7).

Для оценки длительности τ базовой скалярной операции проводилось решение задачи нахождения минимального охватывающего дерева при помощи последовательного алгоритма и полученное таким образом время вы-

числений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 4,76 нсек. Все вычисления производились над числовыми значениями типа int, размер которого на данной платформе равен 4 байта (следовательно, w=4).

Результаты вычислительных экспериментов даны в таблице 10.3. Эксперименты проводились с использованием двух, четырех и восьми процессоров. Время указано в секундах.

Таблица 10.3. Результаты вычислительных экспериментов для параллельного алгоритма Прима

Кол-во вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
	Время	Время	Ускорение	Время	Ускорение	Время	Ускорение
1000	0,044	0,248	0,176	0,932	0,047	1,574	0,028
2000	0,208	0,684	0,304	1,800	0,115	2,159	0,096
3000	0,485	1,403	0,346	2,214	0,219	3,195	0,152
4000	0,873	1,946	0,622	3,324	0,263	5,431	0,161
5000	1,432	2,665	0,736	2,933	0,488	4,119	0,348
6000	2,189	2,900	0,821	4,291	0,510	7,737	0,283
7000	3,042	3,236	0,940	6,327	0,481	8,825	0,345
8000	4,150	4,462	0,930	6,993	0,593	10,390	0,399
9000	5,622	5,834	0,964	7,475	0,752	10,764	0,522
10000	7,512	6,990	1,075	8,597	0,874	14,095	0,533

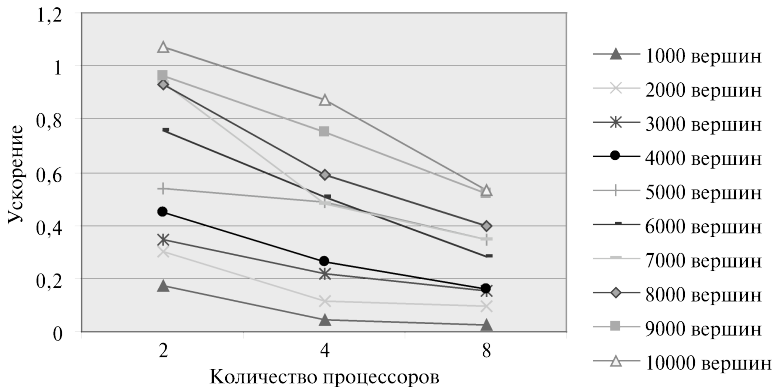


Рис. 10.7. Графики зависимости ускорения параллельного алгоритма Прима от числа используемых процессоров при различном количестве вершин в модели

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.8) приведено в таблице 10.4 и на рис. 10.8.

Таблица 10.4. Сравнение экспериментального и теоретического времени работы алгоритма Прима

Количество вершин	Последовательный алгоритм	Параллельный алгоритм					
		2 процессора		4 процессора		8 процессоров	
	T_1^*	T_2	T_2^*	T_4	T_4^*	T_8	T_8^*
1000	0,044	0,405	0,248	0,804	0,932	1,205	1,574
2000	0,208	0,820	0,684	1,613	1,800	2,412	2,159
3000	0,485	1,245	1,403	2,426	2,214	3,622	3,195
4000	0,873	1,679	1,946	3,245	3,324	4,834	5,431
5000	1,432	2,122	2,665	4,068	2,933	6,048	4,119
6000	2,189	2,575	2,900	4,896	4,291	7,265	7,737
7000	3,042	3,038	3,236	5,728	6,327	8,484	8,825
8000	4,150	3,510	4,462	6,566	6,993	9,705	10,390
9000	5,622	3,991	5,834	7,408	7,475	10,929	10,764
10000	7,512	4,482	6,990	8,255	8,597	12,155	14,095

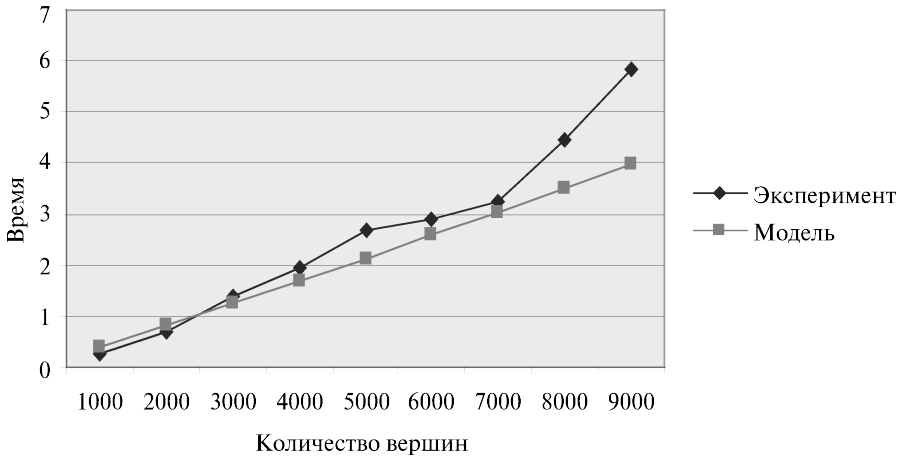


Рис. 10.8. Графики экспериментально установленного времени работы параллельного алгоритма Прима и теоретической оценки в зависимости от количества вершин в модели при использовании двух процессоров

Как можно заметить из табл. 10.4 и рис. 10.8, теоретические оценки определяют время выполнения алгоритма Прима с достаточно высокой погрешностью. Причина такого расхождения может состоять в том, что модель Хокни менее точна при оценке времени передачи сообщений с небольшим объемом передаваемых данных. Для уточнения получаемых оценок необходимым является использование других более точных моделей расчета трудоемкости коммуникационных операций – обсуждение этого вопроса проведено в лекции 3.

10.3. Задача оптимального разделения графов

Проблема оптимального разделения графов относится к числу часто возникающих задач при проведении различных научных исследований, использующих параллельные вычисления. В качестве примера можно привести задачи обработки данных, в которых области расчетов аппроксимируются двумерными или трехмерными вычислительными сетками. Получение результатов в таких задачах сводится, как правило, к выполнению тех или иных процедур обработки для каждого элемента (узла) сети. При этом в ходе вычислений между соседними элементами сети может происходить передача результатов обработки и т.п. Эффективное решение таких задач на многопроцессорных системах с распределенной памятью предполагает разделение сети между процессорами таким образом, чтобы каждому из процессоров выделялось примерно равное число элементов сети, а межпроцессорные коммуникации, необходимые для выполнения информационного обмена между соседними элементами, были минимальными. На рис. 10.9 показан пример нерегулярной сети, разделенной на 4 части (различные части разбиения сети выделены темным цветом различной интенсивности).

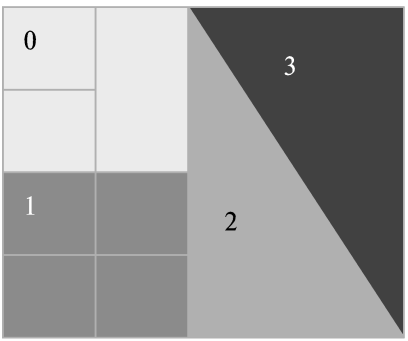


Рис. 10.9. Пример разделения нерегулярной сети

Очевидно, что такие задачи разделения сети между процессорами могут быть сведены к проблеме оптимального разделения графа. Данный подход целесообразен, потому что представление модели вычислений в виде графа позволяет легче решить вопросы хранения обрабатываемых данных и предоставляет возможность применения типовых алгоритмов обработки графов.

Для представления сети в виде графа каждому элементу сети можно поставить в соответствие вершину графа, а дуги графа использовать для отражения свойства близости элементов сети (например, определять дуги между вершинами графа тогда и только тогда, когда соответствующие элементы исходной сети являются соседними). При таком подходе, например, для сети на рис. 10.9, будет сформирован граф, приведенный на рис. 10.10.

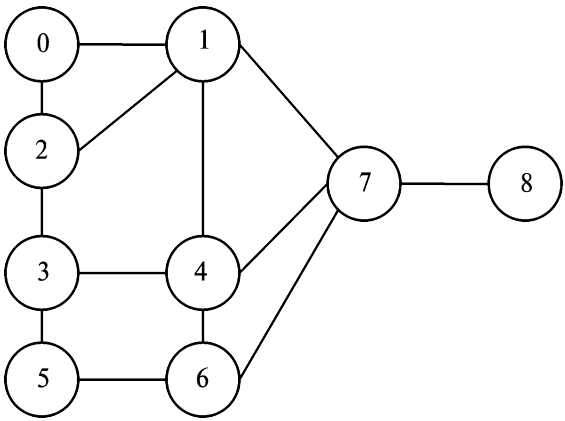


Рис. 10.10. Пример графа, моделирующего структуру сети на рис. 10.9

Дополнительная информация по проблеме разделения графов может быть получена, например, в [67].

Задача оптимального разделения графов сама может являться предметом распараллеливания. Это бывает необходимо в тех случаях, когда вычислительной мощности и объема оперативной памяти обычных компьютеров недостаточно для эффективного решения задачи. Параллельные алгоритмы разделения графов рассматриваются во многих научных работах: [20, 38, 44, 48, 49, 65, 74].

10.3.1. Постановка задачи оптимального разделения графов

Пусть дан взвешенный неориентированный граф $G=(V,E)$, каждой вершине $v \in V$ и каждому ребру $e \in E$ которого приписан вес. Задача оптимального разделения графа состоит в разбиении его вершин на непересе-

кающиеся подмножества с максимально близкими суммарными весами вершин и минимальным суммарным весом ребер, проходящих между полученными подмножествами вершин.

Следует отметить возможную противоречивость указанных критериев разбиения графа – равновесность подмножеств вершин может не соответствовать минимальности весов граничных ребер и наоборот. В большинстве случаев необходимым является выбор того или иного компромиссного решения. Так, в случае невысокой доли коммуникаций может оказаться эффективной оптимизация веса ребер только среди решений, обеспечивающих оптимальное разбиение множества вершин по весу.

Далее для простоты изложения учебного материала будем полагать веса вершин и ребер графа равными единице.

10.3.2. Метод рекурсивного деления пополам

Для решения задачи разбиения графа можно рекурсивно применить *метод бинарного деления* (*the binary bisection method*), при котором на первой итерации граф разделяется на две равные части, далее на втором шаге каждая из полученных частей также разбивается на две части и т. д. В данном подходе для разбиения графа на k частей необходимо $\log_2 k$ уровней рекурсии и выполнение k-1 деления пополам. В случае когда требуемое количество разбиений k не является степенью двойки, каждое деление пополам необходимо осуществлять в соответствующем соотношении.

Поясним схему работы метода деления пополам на примере разделения графа на рис. 10.11 на 5 частей. Сначала граф следует разделить на 2 части в оотношении 2:3 (непрерывная линия), затем правую часть разбиения – в отношении 1:3 (штриховая линия), после этого осталось разделить 2 крайние подобласти слева и справа в отношении 1:1 (штрих-пунктир).

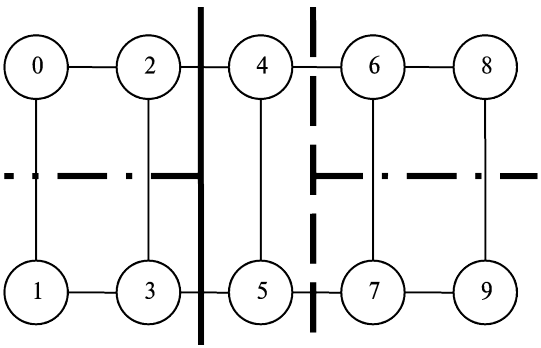


Рис. 10.11. Пример разбиения графа на 5 частей методом рекурсивного деления пополам

10.3.3. Геометрические методы

Геометрические методы (см., например, [21, 35, 44, 53, 55, 58, 61, 65]) выполняют разбиение сетей, основываясь исключительно на координатной информации об узлах сети. Так как эти методы не принимают во внимание информацию о связности элементов сети, они не могут явно привести к минимизации суммарного веса граничных ребер (в терминах графа, соответствующего сети). Для минимизации межпроцессорных коммуникаций геометрические методы оптимизируют некоторые вспомогательные показатели (например, длину границы между разделенными участками сети).

Обычно геометрические методы не требуют большого объема вычислений, однако качество их разбиения уступает методам, принимающим во внимание связность элементов сети.

10.3.3.1. Покоординатное разбиение

Покоординатное разбиение (*the coordinate nested dissection*) – это метод, основанный на рекурсивном делении пополам сети по наиболее длинной стороне. В качестве иллюстрации на рис. 10.12 показан пример сети, при разделении которой именно такой способ разбиения дает существенно меньшее количество информационных связей между разделенными частями, по сравнению со случаем, когда сеть делится по меньшей (вертикальной) стороне.

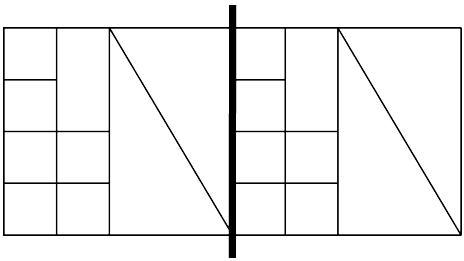


Рис. 10.12. Пример разделения сети графическим методом по наибольшей размерности (граница раздела показана жирной линией)

Общая схема выполнения метода состоит в следующем. Сначала вычисляются центры масс элементов сети. Полученные точки проектируются на ось, соответствующую наибольшей стороне разделяемой сети. Таким образом мы получаем упорядоченный список всех элементов сети. Делением списка пополам (возможно, в нужной пропорции) мы получаем требуемую бисекцию. Аналогичным способом полученные фрагменты разбиения рекурсивно делятся на нужное число частей.

Метод координатного вложенного разбиения работает очень быстро и требует небольшого количества оперативной памяти. Однако получаемое разбиение уступает по качеству более сложным и вычислительно трудоемким методам. Кроме того, в случае сложной структуры сети алгоритм может получать разбиение с несвязанными подсетями.

10.3.3.2. Рекурсивный инерционный метод деления пополам

Предыдущая схема могла производить разбиение сети только по линии, перпендикулярной одной из координатных осей. Во многих случаях такое ограничение оказывается критичным для построения качественного разбиения. Достаточно повернуть сеть на рис. 10.12 под острым углом к координатным осям (см. рис. 10.13), чтобы убедиться в этом. Для минимизации границы между подсетями желательна возможность проведения линии разделения с любым требуемым углом поворота. Возможный способ определения угла поворота, используемый в рекурсивном *инерционном методе деления пополам* (*the recursive inertial bisection*), состоит в использовании главной инерционной оси (см., например, [62]), считая элементы сети точечными массами. Линия бисекции, ортогональная полученной оси, как правило, дает границу наименьшей длины.

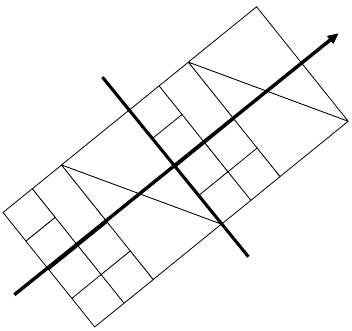


Рис. 10.13. Пример разделения сети методом рекурсивной инерционной бисекции. Стрелкой показана главная инерционная ось

10.3.3.3. Деление сети с использованием кривых Пеано

Одним из недостатков предыдущих графических методов является то, что при каждой бисекции эти методы учитывают только одну размерность. Таким образом, схемы, учитывающие больше размерностей, могут обеспечить лучшее разбиение.

Один из таких методов упорядочивает элементы в соответствии с позициями центров их масс вдоль кривых Пеано. Кривые Пеано – это кри-

вые, полностью заполняющие области больших размерностей (например, квадрат или куб). Применение таких кривых обеспечивает близость точек фигуры, которые соответствуют точкам, близким на кривой. После получения списка элементов сети, упорядоченного в зависимости от расположения на кривой, достаточно разделить список на необходимое число частей в соответствии с установленным порядком. Получаемый в результате такого подхода метод носит в литературе наименование *алгоритма деления сети с использованием кривых Пеано* (*the space-filling curve technique*). Подробнее о методе можно прочитать в работах [56, 58, 61].

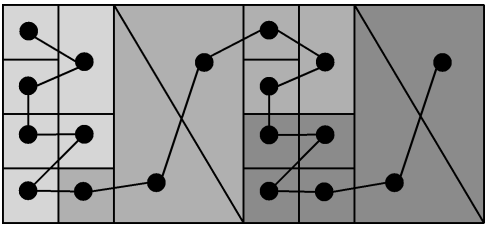


Рис. 10.14. Пример разделения сети на 3 части с использованием кривых Пеано

10.3.4. Комбинаторные методы

В отличие от геометрических методов, комбинаторные алгоритмы (см., например, [36,67]) обычно оперируют не с сетью, а с графом, построенным для этой сети. Соответственно, в отличие от геометрических схем, комбинаторные методы не принимают во внимание информацию о близости расположения элементов сети друг относительно друга, руководствуясь только смежностью вершин графа. Комбинаторные методы обычно обеспечивают более сбалансированное разбиение и меньшее информационное взаимодействие полученных подсетей. Однако комбинаторные методы имеют тенденцию работать существенно дольше, чем их геометрические аналоги.

10.3.4.1. Деление с учетом связности

С самых общих позиций понятно, что при разделении графа информационная зависимость между разделенными подграфами будет меньше, если соседние вершины (вершины, между которыми имеются дуги) будут находиться в одном подграфе. *Алгоритм деления графов с учетом связности* (*the levelized nested dissection algorithm*) пытается достичь этого, последовательно добавляя к формируемому подграфу соседей. На каждой итера-

ции алгоритма происходит разделение графа на 2 части. Таким образом, разделение графа на требуемое число частей достигается путем рекурсивного применения алгоритма.

Общая схема алгоритма может быть описана при помощи следующего набора правил.

Алгоритм 10.2. Общая схема выполнения алгоритма деления графов с учетом связности

1. Iteration = 0.
2. Присвоение номера Iteration произвольной вершине графа.
3. Присвоение ненумерованным соседям вершин с номером Iteration номера Iteration + 1.
5. Iteration = Iteration + 1.
6. Если еще есть неперенумерованные соседи, то переход на шаг 3.
7. Разделение графа на 2 части в порядке нумерации.

Для минимизации информационных зависимостей имеет смысл в качестве начальной выбирать граничную вершину. Поиск такой вершины можно осуществить методом, близким к рассмотренной схеме. Так, перенумеровав вершины графа в соответствии с алгоритмом 10.2 (начиная нумерацию из произвольной вершины), мы можем взять любую вершину с максимальным номером. Как нетрудно убедиться, она будет граничной.

Пример работы алгоритма приведен на рис. 10.15. Цифрами показаны номера, которые получили вершины в процессе разделения. Сплошной линией показана граница, разделяющая 2 подграфа. Также на рисунке показано лучшее решение (пунктирная линия). Очевидно, что полученное алгоритмом разбиение далеко от оптимального, так как в приведенном примере есть решение только с тремя пересеченными ребрами вместо пяти.

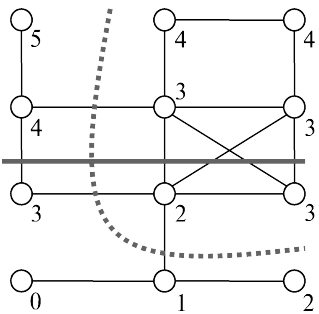


Рис. 10.15. Пример работы алгоритма деления графов с учетом связности

10.3.4.2. Алгоритм Кернигана – Лина

В *алгоритме Кернигана – Лина* (*the Kernighan – Lin algorithm*) используется несколько иной подход для решения проблемы оптимального разбиения графа – предполагается, что некоторое начальное разбиение графа уже существует, затем имеющееся приближение улучшается в течение некоторого количества итераций. Применяемый способ улучшения в алгоритме Кернигана – Лина состоит в обмене вершинами между подмножествами имеющегося разбиения графа (см. рис. 10.16). Для формирования требуемого количества частей графа может быть использована, как и ранее, рекурсивная процедура деления пополам.

Общая схема одной итерации алгоритма Кернигана – Лина может быть представлена следующим образом.

Алгоритм 10.3. Общая схема алгоритма Кернигана – Лина

1. Формирование множества пар вершин для перестановки.
   Из вершин, которые еще не были переставлены на данной итерации, формируются все возможные пары (в парах должно присутствовать по одной вершине из каждой части имеющегося разбиения графа).
2. Построение новых вариантов разбиения графа.
   Каждая пара, подготовленная на шаге 1, поочередно используется для обмена вершин между частями имеющегося разбиения графа для получения множества новых вариантов деления.
3. Выбор лучшего варианта разбиения графа.
   Для сформированного на шаге 2 множества новых делений графа выбирается лучший вариант. Этот вариант далее фиксируется как новое текущее разбиение графа, а соответствующая выбранному варианту пара вершин отмечается как использованная на текущей итерации алгоритма.
4. Проверка использования всех вершин.
   При наличии в графе вершин, еще не использованных при перестановках, выполнение итерации алгоритма снова продолжается с шага 1. Если же перебор вершин графа завершен, далее следует шаг 5.
5. Выбор наилучшего варианта разбиения графа.
   Среди всех разбиений графа, полученных на шаге 3 проведенных итераций, выбирается (и фиксируется) наилучший вариант разбиения графа.

Поясним дополнительно, что на шаге 2 итерации алгоритма перестановка вершин каждой очередной пары осуществляется для одного и того же раз-

биения графа, выбранного до начала выполнения итерации или определенного на шаге 3. Общее количество выполняемых итераций, как правило, фиксируется заранее и является параметром алгоритма (за исключением случая остановки при отсутствии улучшения разбиения на очередной итерации).

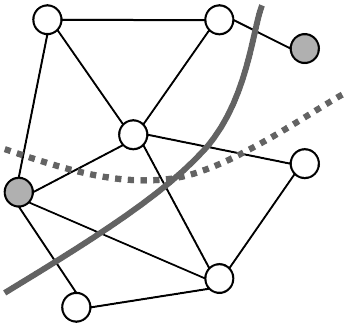


Рис. 10.16. Пример перестановки двух вершин (выделены серым) в методе Кернигана – Лина

10.3.5. Сравнение алгоритмов разбиения графов

Рассмотренные алгоритмы разбиения графов различаются точностью получаемых решений, временем выполнения и возможностями для распараллеливания (под точностью понимается величина близости получаемых при помощи алгоритмов решений к оптимальным вариантам разбиения графов). Выбор наиболее подходящего алгоритма в каждом конкретном

Таблица 10.5. Сравнительная таблица некоторых алгоритмов разделения графов

Алгоритмы		Необходимость координатной информации	Точность	Время выполнения	Возможности для распараллеливания
Покоординатное разбиение		Да	•	•	• • •
Рекурсивный инерционный метод деления пополам		Да	• •	•	• • •
Деление с учетом связности		Нет	• •	• •	• •
Алгоритм Кернигана – Лина	1 итерация	Нет	• •	• •	•
	10 итераций	Нет	• • •	• • •	• •
	50 итераций	Нет	• • • •	• • • •	• • • •

случае является достаточно сложной и неочевидной задачей. Проведению такого выбора может содействовать сведенная воедино в табл. 10.5 (см. [67]) общая характеристика ряда алгоритмов разделения графов, рассмотренных в данном разделе. Дополнительная информация по проблеме оптимального разбиения графов может быть получена, например, в [67].

Столбец «Необходимость координатной информации» отмечает использование алгоритмом координатной информации об элементах сети или вершинах графа.

Столбец «Точность» дает качественную характеристику величины приближения получаемых алгоритмом решений к оптимальным вариантам разбиения графов. Каждый дополнительный закрашенный кружок определяет примерно 10%-процентное улучшение точности приближения.

Столбец «Время выполнения» показывает относительное время, затрачиваемое различными алгоритмами разбиения. Каждый дополнительный закрашенный кружок соответствует увеличению времени разбиения примерно в 10 раз.

Столбец «Возможности для распараллеливания» характеризует свойства алгоритмов для параллельного выполнения. Алгоритм Кернигана – Лина при выполнении только одной итерации почти не поддается распараллеливанию. Этот же алгоритм при большем количестве итераций, а также метод деления с учетом связности могут быть распараллелены со средней эффективностью. Алгоритм покоординатного разбиения и рекурсивный инерционный метод деления пополам обладают высокими показателями для распараллеливания.

10.4. Краткий обзор лекции

В лекции рассмотрен ряд алгоритмов для решения типовых задач обработки графов. Кроме того, приведен обзор методов разделения графа.

В подразделе 10.1 представлен *алгоритм Флойда* (*the Floyd algorithm*) – дается общая вычислительная схема последовательного варианта метода, обсуждаются способы его распараллеливания, проводится анализ эффективности получаемых параллельных вычислений, рассматривается программная реализация метода и приводятся результаты вычислительных экспериментов. Используемый подход к распараллеливанию алгоритма Флойда состоит в разделении вершин графа между процессорами, а необходимое при этом информационное взаимодействие состоит в передаче одной строки матрицы смежности от одного процессора всем процессорам вычислительной системы на каждой итерации метода.

В подразделе 10.2 рассматривается *алгоритм Прима* (*the Prim algorithm*) для решения задачи поиска минимального охватывающего дерева (остова) неориентированного взвешенного графа. Остовом графа называют связный подграф без циклов (дерево), который содержит все вершины исходного графа и ребра, имеющие минимальный суммарный вес. Для алгоритма дается общее описание его исходного последовательного варианта, определяются возможные способы его параллельного выполнения, теоретические оценки ускорения и эффективности параллельных вычислений; также рассматриваются результаты проведенных вычислительных экспериментов. Параллельный вариант алгоритма Прима, как и в предыдущем случае, основывается на разделении вершин графа между процессорами при несколько большем объеме информационных взаимодействий – на каждой итерации алгоритма необходимой является операция сбора данных на одном процессоре и последующая рассылка номера выбранной вершины графа всем процессорам вычислительной системы.

Рассматриваемая в подразделе 10.3 задача оптимального разделения графов является важной для многих научных исследований, использующих параллельные вычисления. Для примера в подразделе приведен общий способ перехода от двумерной или трехмерной сети, моделирующей процесс вычислений, к соответствующему ей графу. Для решения задачи разбиения графов были рассмотрены *геометрические методы*, использующие при разделении сетей только координатную информацию об узлах сети, и *комбинаторные алгоритмы*, руководствующиеся смежностью вершин графа. К числу рассмотренных геометрических методов относятся *покоординатное разбиение* (*the coordinate nested dissection method*), *рекурсивный инерционный метод деления пополам* (*the recursive inertial bisection method*), *деление сети с использованием кривых Пеано* (*the space-filling curve techniques*). К числу рассмотренных комбинаторных алгоритмов относятся *деление с учетом связности* (*the levelized nested dissection*) *и алгоритм Кернигана – Лина* (*the Kernighan – Lin algorithm*). Для сопоставления рассмотренных подходов приводится общая сравнительная характеристика алгоритмов по времени выполнения, точности получаемого решения, возможностям для распараллеливания и т. п.

10.5. Обзор литературы

Дополнительная информация по алгоритмам Флойда и Прима может быть получена, например, в [26].

Подробное рассмотрение вопросов, связанных с проблемой разделения графов, содержится в работах [21, 36, 37, 44, 53, 55, 58, 61, 65, 67].

Параллельные алгоритмы разделения графов рассматриваются в [20, 38, 44, 48, 49, 65, 74].

10.6. Контрольные вопросы

1. Приведите определение графа. Какие основные способы используются для задания графов?
2. В чем состоит задача поиска всех кратчайших путей?
3. Приведите общую схему алгоритма Флойда. Какова трудоемкость алгоритма?
4. В чем состоит способ распараллеливания алгоритма Флойда?
5. В чем заключается задача нахождения минимального охватывающего дерева? Приведите пример использования задачи на практике.
6. Приведите общую схему алгоритма Прима. Какова трудоемкость алгоритма?
7. В чем состоит способ распараллеливания алгоритма Прима?
8. В чем отличие геометрических и комбинаторных методов разделения графа? Какие методы являются более предпочтительными? Почему?
9. Приведите описание метода покоординатного разбиения и алгоритма разделения с учетом связности. Какой из этих методов является более простым для реализации?

10.7. Задачи и упражнения

1. Используя приведенный программный код, выполните реализацию параллельного алгоритма Флойда. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
2. Выполните реализацию параллельного алгоритма Прима. Проведите вычислительные эксперименты. Постройте теоретические оценки с учетом параметров используемой вычислительной системы. Сравните полученные оценки с экспериментальными данными.
3. Разработайте программную реализацию алгоритма Кернигана – Лина. Дайте оценку возможности распараллеливания этого алгоритма.

Лекция 11. Параллельные методы решения дифференциальных уравнений в частных производных

В лекции рассматриваются вопросы организации параллельных вычислений для решения задач, в которых при математическом моделировании используются дифференциальные уравнения в частных производных. Для численного решения подобных задач обычно применяется метод конечных разностей (метод сеток), обладающий высокой вычислительной трудоемкостью. В лекции последовательно разбираются возможные способы распараллеливания сеточных методов на многопроцессорных вычислительных системах с общей и распределенной памятью. При этом большое внимание уделяется проблемам, возникающим при организации параллельных вычислений, анализу причин появления таких проблем и нахождению путей их преодоления. Для наглядной демонстрации излагаемого материала в качестве учебного примера рассматривается проблема численного решения задачи Дирихле для уравнения Пуассона.

Ключевые слова: параллельные алгоритмы решения дифференциальных уравнений в частных производных, принципы распараллеливания, алгоритм Гаусса – Зейделя, параллелизм по данным, ленточное и блочное разбиения данных, ускорение, эффективность, вычислительная и коммуникационная сложность, теоретическая оценка времени выполнения алгоритма, программная реализация, операции передачи данных, MPI, вычислительный эксперимент.

Дифференциальные уравнения в частных производных представляют собой широко применяемый математический аппарат при разработке моделей в самых разных областях науки и техники. К сожалению, явное решение этих уравнений в аналитическом виде оказывается возможным только в частных простых случаях, и, как результат, возможность анализа математических моделей, построенных на основе дифференциальных уравнений, обеспечивается при помощи приближенных численных методов решения.

Объем выполняемых при этом вычислений обычно является значительным, и использование высокопроизводительных вычислительных систем традиционно для данной области вычислительной математики.

Проблематика численного решения дифференциальных уравнений в частных производных является областью интенсивных исследований.

Рассмотрим в качестве учебного примера *проблему численного решения задачи Дирихле для уравнения Пуассона*, которая определяется как зада-

ча нахождения функции $u=u(x,y)$, удовлетворяющей в области определения D уравнению

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2}+\dfrac{\partial^2 u}{\partial y^2}=f(x,y), & (x,y)\in D, \\ u(x,y)=g(x,y), & (x,y)\in D^0, \end{cases}$$

и принимающей значения $g(x,y)$ на границе D^0 области D (f и g являются функциями, задаваемыми при постановке задачи). Подобная модель может применяться для описания установившегося течения жидкости, стационарных тепловых полей, процессов теплопередачи с внутренними источниками тепла и деформации упругих пластин. Данный пример часто используется в качестве учебно-практической задачи при изложении возможных способов организации эффективных параллельных вычислений (см. [12, 60]).

Для простоты изложения материала в качестве области задания D функции $u(x,y)$ далее будет использоваться единичный квадрат

$$D=\{(x,y)\in D:\ 0\le x,y\le 1\}.$$

11.1. Последовательные методы решения задачи Дирихле

Одним из наиболее распространенных подходов к численному решению дифференциальных уравнений является *метод конечных разностей* (*метод сеток*) (см., например, [6, 13, 60]). Следуя этому подходу, область решения D можно представить в виде дискретного (как правило, равномерного) набора (*сетки*) точек (*узлов*). Так, например, прямоугольная сетка в области D может быть задана в виде (рис. 11.1)

$$\begin{cases} D_h=\{(x_i,y_j): x_i=ih,\ y_i=jh,\ 0\le i,j\le N+1, \\ \qquad\qquad h=1/(N+1), \end{cases}$$

где величина N задает количество внутренних узлов по каждой из координат области D.

Обозначим оцениваемую при подобном дискретном представлении аппроксимацию функции $u(x,y)$ в точках (x_i, y_j) через u_{ij}. Тогда, используя *пятиточечный шаблон* (см. рис. 11.1) для вычисления значений производных, мы можем представить уравнение Пуассона в *конечно-разностной форме*

$$\frac{u_{i-1,j}+u_{i+1,j}+u_{i,j-1}+u_{i,j+1}-4u_{ij}}{h^2}=f_{ij}.$$

Данное уравнение может быть разрешено относительно u_{ij}:

$$u_{ij} = 0{,}25(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij}).$$

Разностное уравнение, записанное в подобной форме, позволяет определять значение u_{ij} по известным значениям функции $u(x,y)$ в соседних узлах используемого шаблона. Данный результат служит основой для построения различных *итерационных схем* решения задачи Дирихле, в которых в начале вычислений формируется некоторое приближение для значений u_{ij}, а затем эти значения последовательно уточняются в соответствии с приведенным соотношением. Так, например, *метод Гаусса – Зейделя* для проведения итераций уточнения использует правило

$$u_{ij}^k = 0{,}25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}),$$

по которому очередное k-е приближение значения u_{ij} вычисляется по последнему k-му приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему (k-1)-му приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Выполнение итераций обычно продолжается до тех пор, пока получаемые в результате итераций изменения значений u_{ij} не станут меньше некоторой заданной величины (*требуемой точности вычислений*). Сходимость описанной процедуры (получение решения с любой желаемой точностью) является предметом всестороннего математического анализа (см., например, [6, 13, 60]), здесь же отметим, что последовательность решений, получаемых методом сеток, равномерно сходится к решению задачи Дирихле, а погрешность решения имеет порядок h^2.

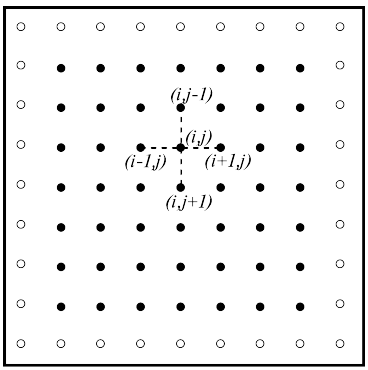


Рис. 11.1. Прямоугольная сетка в области D (темные точки представляют внутренние узлы сетки, нумерация узлов в строках слева направо, а в столбцах — сверху вниз)

Рассмотренный алгоритм (метод Гаусса – Зейделя) на псевдокоде, приближенном к алгоритмическому языку C++, может быть представлен в виде:

Алгоритм 11.1. Последовательный алгоритм Гаусса – Зейделя

```
// Алгоритм 11.1
do {
  dmax = 0; // максимальное изменение значений u
  for ( i=1; i<N+1; i++ )
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      dm = fabs(temp-u[i][j]);
      if ( dmax < dm ) dmax = dm;
    }
} while ( dmax > eps );
```

(напомним, что значения u_{ij} при индексах $i,j=0,N+1$ являются граничными, задаются при постановке задачи и не изменяются в ходе вычислений).

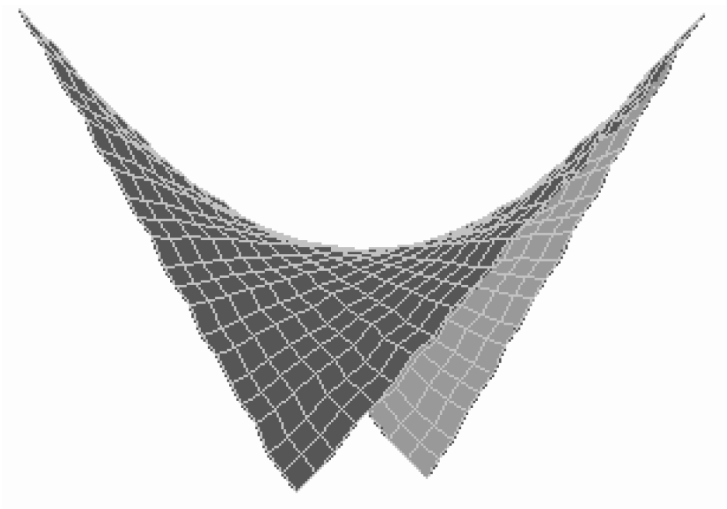


Рис. 11.2. Вид функции $u(x,y)$ в примере для задачи Дирихле

Для примера на рис. 11.2 приведен вид функции $u(x,y)$, полученной для задачи Дирихле при следующих граничных условиях:

$$\begin{cases} f(x,y)=0, & (x,y)\in D, \\ 100-200x, & y=0, \\ 100-200y, & x=0, \\ -100+200x, & y=1, \\ -100+200y, & x=1. \end{cases}$$

Общее количество итераций метода Гаусса – Зейделя составило 210 при точности решения *eps*=0,1 и *N*=100 (в качестве начального приближения величин u_{ij} использовались значения, сгенерированные датчиком случайных чисел из диапазона [-100, 100]).

11.2. Организация параллельных вычислений для систем с общей памятью

Как следует из приведенного описания, сеточные методы характеризуются значительной вычислительной трудоемкостью

$$T_1 = kmN^2,$$

где *N* есть количество узлов по каждой из координат области *D, m* — число операций, выполняемых методом для одного узла сетки, *k* — количество итераций метода до выполнения условия остановки.

11.2.1. Использование OpenMP для организации параллелизма

Рассмотрим возможные способы организации параллельных вычислений для сеточных методов на многопроцессорных вычислительных системах с общей памятью. При изложении материала будем предполагать, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечиваются на аппаратном уровне). Как уже отмечалось ранее, многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (*symmetric multiprocessors, SMP*) – см. п. 1.3.1.

Обычный подход при организации вычислений для подобных систем – создание новых параллельных версий на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг

от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки программирования, расширенные некоторым набором операторов для параллельных вычислений.

Оба указанных подхода приводят к необходимости значительной переработки существующего программного обеспечения, и это в значительной степени затрудняет широкое распространение параллельных вычислений. Как результат, в последнее время активно развивается еще один подход к разработке параллельных программ, когда указания программиста по организации параллельных вычислений добавляются в программу при помощи тех или иных внеязыковых средств языка программирования – например, в виде директив или комментариев, которые обрабатываются специальным препроцессором до начала компиляции программы. При этом исходный операторный текст программы остается неизменным, и по нему в случае отсутствия препроцессора компилятор построит исходный последовательный программный код. Препроцессор же, будучи примененным, заменяет директивы параллелизма на некоторый дополнительный программный код (как правило, в виде обращений к процедурам какой-либо параллельной библиотеки).

Рассмотренный выше подход является основой *технологии OpenMP* (см., например, [27]), наиболее широко применяемой в настоящее время для организации параллельных вычислений на многопроцессорных системах с общей памятью. В рамках данной технологии директивы параллелизма используются для выделения в программе *параллельных областей* (*parallel regions*), в которых последовательный исполняемый код может быть разделен на несколько раздельных командных *потоков* (*threads*). Далее эти потоки могут исполняться на разных процессорах вычислительной системы. В результате такого подхода программа представляется в виде набора последовательных (*однопотоковых*) и параллельных (*многопотоковых*) участков программного кода (см. рис. 11.3). Подобный принцип организации параллелизма получил наименование *«вилочного»* (*fork-join*) или *пульсирующего параллелизма*. Более полная информация по технологии OpenMP может быть получена в литературе (см., например, [27, 66]) или в информационных ресурсах сети Интернет. В данной лекции возможности OpenMP будут излагаться в объеме, необходимом для демонстрации возможных способов разработки параллельных программ для рассматриваемого учебного примера решения задачи Дирихле.

11.2.2. Проблема синхронизации параллельных вычислений

Первый вариант параллельного алгоритма для метода сеток может быть получен, если разрешить произвольный порядок пересчета значений u_{ij}. Программа для данного способа вычислений может быть представлена в следующем виде:

Алгоритм 11.2. Первый вариант параллельного алгоритма Гаусса – Зейделя

```
// Алгоритм 11.2
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,N,dmax) private(i,temp,d)
  for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,N,dmax) private(j,temp,d)
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      omp_set_lock(dmax_lock);
        if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
    } // конец вложенной параллельной области
  } // конец внешней параллельной области
} while ( dmax > eps );
```

Следует отметить, что программа получена из исходного последовательного кода путем добавления директив и операторов обращения к функциям библиотеки OpenMP (эти дополнительные строки в программе выделены темным шрифтом).

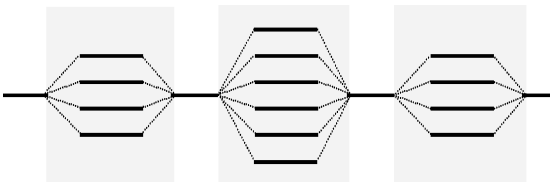


Рис. 11.3. Параллельные области, создаваемые директивами OpenMP

Как следует из текста программы, параллельные области в данном примере задаются директивой `parallel for`, являются вложенными и включают в свой состав операторы цикла `for`. Компилятор, поддерживающий технологию OpenMP, разделяет выполнение итераций цикла между несколькими потоками программы, количество которых обычно совпадает с числом процессоров в вычислительной системе. Параметры директивы `shared` и `private` определяют доступность данных в потоках программы – переменные, описанные как `shared`, являются общими для потоков, для переменных с описанием `private` создаются отдельные копии для каждого потока, которые могут использоваться в потоках независимо друг от друга.

Наличие общих данных обеспечивает возможность взаимодействия потоков. В этом плане разделяемые переменные могут рассматриваться как *общие ресурсы потоков*, и, как результат, их применение должно выполняться с соблюдением *правил взаимоисключения* (изменение каким-либо потоком значений общих переменных должно приводить к блокировке доступа к модифицируемым данным для всех остальных потоков). В данном примере таким разделяемым ресурсом является величина dmax, доступ потоков к которой регулируется специальной служебной переменной (*замком*) `dmax_lock` и функциями `omp_set_lock` (разрешение или блокировка доступа) и `omp_unset_lock` (снятие запрета на доступ). Подобная организация программы гарантирует единственность доступа потоков для изменения разделяемых данных. Участки программного кода (блоки между обращениями к функциям `omp_set_lock` и `omp_unset_lock`), для которых обеспечивается взаимоисключение, обычно именуются *критическими секциями*.

Результаты вычислительных экспериментов приведены в табл. 11.1 (здесь и далее для параллельных программ, разработанных с использованием технологии OpenMP, использовался четырехпроцессорный сервер кластера Нижегородского университета с процессорами Pentium III, 700 MHz, 512 RAM).

Оценим полученный результат. Разработанный параллельный алгоритм является корректным, т.е. обеспечивающим решение поставленной задачи. Использованный при разработке подход обеспечивает достижение практически максимально возможного параллелизма – для выполнения программы может быть задействовано вплоть до N^2 процессоров. Тем не менее результат не может быть признан удовлетворительным: программа будет работать медленно и вместо ускорения мы получим замедление вычислений. Основная причина такого положения дел – чрезмерно высокая *синхронизация* параллельных участков программы. В нашем примере каждый параллельный поток по-

Таблица 11.1. Результаты вычислительных экспериментов для параллельных вариантов алгоритма Гаусса – Зейделя (p=4)

Размер сетки	Последовательный метод Гаусса – Зейделя (алгоритм 11.1)		Параллельный алгоритм 11.2			Параллельный алгоритм 11.3		
	k	t	k	t	S	k	t	S
100	210	0,06	210	1,97	0,03	210	0,03	2,03
200	273	0,34	273	11,22	0,03	273	0,14	2,43
300	305	0,88	305	29,09	0,03	305	0,36	2,43
400	318	3,78	318	54,20	0,07	318	0,64	5,90
500	343	6,00	343	85,84	0,07	343	1,06	5,64
600	336	8,81	336	126,38	0,07	336	1,50	5,88
700	344	12,11	344	178,30	0,07	344	2,42	5,00
800	343	16,41	343	234,70	0,07	343	8,08	2,03
900	358	20,61	358	295,03	0,07	358	11,03	1,87
1000	351	25,59	351	366,16	0,07	351	13,69	1,87
2000	367	106,75	367	1585,84	0,07	367	56,63	1,89
3000	370	243,00	370	3598,53	0,07	370	128,66	1,89

(k – количество итераций, t – время (сек), S – ускорение)

сле усреднения значений u_{ij} должен проверить (и возможно, изменить) значение величины `dmax`. Разрешение на использование переменной может получить только один поток – все остальные потоки должны быть блокированы. После освобождения общей переменной управление может получить следующий поток и т.д. В результате необходимости синхронизации доступа многопотоковая параллельная программа превращается фактически в последовательно выполняемый код, причем менее эффективный, чем исходный последовательный вариант, т.к. организация синхронизации приводит к дополнительным вычислительным затратам – см. рис. 11.4. Следует обратить внимание, что, несмотря на идеальное распределение вычислительной нагрузки между процессорами, для приведенного на рис. 11.4 соотношения параллельных и последовательных вычислений, в каждый текущий момент времени (после момента первой синхронизации) только не более двух процессоров одновременно выполняют действия, связанные с решением задачи. Подобный эффект вырождения параллелизма из-за интенсивной синхронизации параллельных участков программы обычно именуется *сериализацией* (*serialization*).

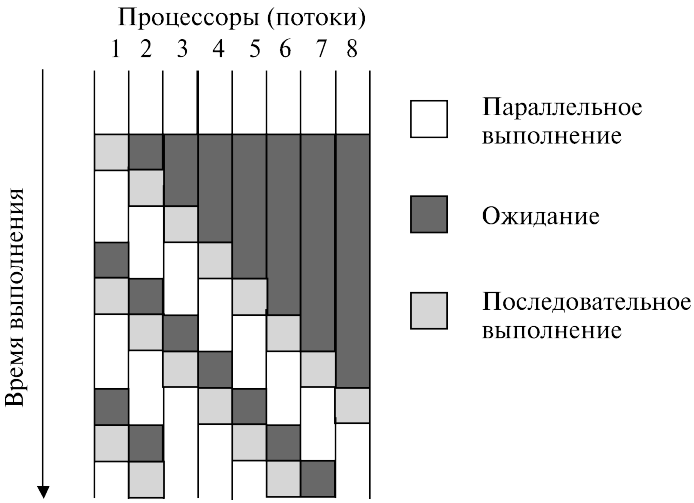


Рис. 11.4. Пример возможной схемы выполнения параллельных потоков при наличии синхронизации (взаимоисключения)

Как показывают выполненные рассуждения, путь для достижения эффективности параллельных вычислений лежит в уменьшении необходимых моментов синхронизации параллельных участков программы. Так, в нашем примере мы можем ограничиться распараллеливанием только одного внешнего цикла `for`. Кроме того, для снижения количества возможных блокировок применим для оценки максимальной погрешности многоуровневую схему расчета: пусть параллельно выполняемый поток первоначально формирует локальную оценку погрешности `dm` только для своих обрабатываемых данных (одной или нескольких строк сетки), затем при завершении вычислений поток сравнивает свою оценку `dm` с общей оценкой погрешности `dmax`.

Новый вариант программы решения задачи Дирихле имеет вид:

Алгоритм 11.3. Второй вариант параллельного алгоритма Гаусса – Зейделя

```
// Алгоритм 11.3
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,N,dmax) private(i,temp,d,dm)
```

```
    for ( i=1; i<N+1; i++ ) {
      dm = 0;
      for ( j=1; j<N+1; j++ ) {
        temp = u[i][j];
        u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                  u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
        d = fabs(temp-u[i][j])
        if ( dm < d ) dm = d;
      }
      omp_set_lock(dmax_lock);
        if ( dmax < dm ) dmax = dm;
      omp_unset_lock(dmax_lock);
      }
    } // конец параллельной области
  } while ( dmax > eps );
```

Как результат выполненного изменения схемы вычислений, количество обращений к общей переменной `dmax` уменьшается с N^2 до N раз, что должно приводить к существенному снижению затрат на синхронизацию потоков и уменьшению проявления эффекта сериализации вычислений. Результаты экспериментов с данным вариантом параллельного алгоритма, приведенные в табл. 11.1, показывают существенное изменение ситуации – ускорение в ряде экспериментов оказывается даже большим, чем используемое количество процессоров (такой эффект *сверхлинейного ускорения* достигается за счет наличия у каждого из процессоров вычислительного сервера своей быстрой кэш-памяти). Следует также обратить внимание, что улучшение показателей параллельного алгоритма достигнуто при снижении максимально возможного параллелизма (для выполнения программы может использоваться не более N процессоров).

11.2.3. Возможность неоднозначности вычислений в параллельных программах

Последний рассмотренный вариант организации параллельных вычислений для метода сеток обеспечивает практически максимально возможное ускорение выполняемых расчетов – так, в экспериментах данное ускорение достигало величины 5,9 при использовании четырехпроцессорного вычислительного сервера. Вместе с этим необходимо отметить, что разработанная вычислительная схема расчетов имеет важную принципиальную особенность: порождаемая при вычислениях последовательность обработки данных может различаться при разных запусках программы даже при одних и тех же исходных параметрах решаемой задачи.

Данный эффект может проявляться в силу изменения каких-либо условий выполнения программы (вычислительной нагрузки, алгоритмов синхронизации потоков и т.п.), что может повлиять на временные соотношения между потоками (см. рис. 11.5). Взаиморасположение потоков по области расчетов может быть различным: одни потоки могут опережать другие и, наоборот, часть потоков могут отставать (при этом характер взаиморасположения может меняться в ходе вычислений). Подобное поведение параллельных участков программы обычно именуется *состязанием потоков* (*race condition*) и отражает важный принцип параллельного программирования – временная динамика выполнения параллельных потоков не должна учитываться при разработке параллельных алгоритмов и программ.

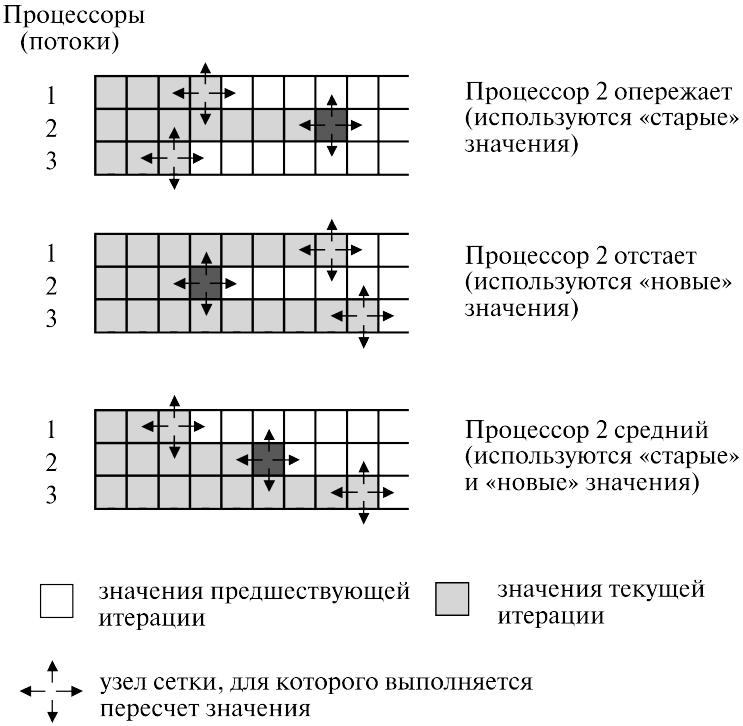


Рис. 11.5. Возможные различные варианты взаиморасположения параллельных потоков (состязание потоков)

В рассматриваемом примере при вычислении нового значения u_{ij} в зависимости от условий выполнения могут использоваться разные (от предыдущей или текущей итераций) оценки соседних значений по верти-

кали. Тем самым, количество итераций метода до выполнения условия остановки и, самое главное, конечное решение задачи могут различаться при повторных запусках программы. Получаемые оценки величин u_{ij}. будут соответствовать точному решению задачи в пределах задаваемой точности, но, тем не менее, могут быть различными. Применение вычислений такого типа для сеточных алгоритмов получило наименование *метода хаотической релаксации* (*chaotic relaxation*).

11.2.4. Проблема взаимоблокировки

Возможный подход для получения однозначных результатов (уход от состязания потоков) может состоять в ограничении доступа к узлам сетки, которые обрабатываются в параллельных потоках программы. Для этого можно ввести набор замков `row_lock[N]`, который позволит потокам закрывать доступ к «своим» строкам сетки.

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// обработка i строки сетки
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```

Закрыв доступ к своим данным, параллельный поток уже не будет зависеть от динамики выполнения других параллельных участков программы. Результат вычислений потока однозначно определяется значениями данных в момент начала расчетов.

Данный подход позволяет продемонстрировать еще одну проблему, которая может возникать в ходе параллельных вычислений. Эта проблема состоит в том, что при организации доступа к множественным общим переменным может появляться конфликт между параллельными потоками и этот конфликт не может быть разрешен успешно. Так, в приведенном фрагменте программного кода при обработке потоками двух последовательных строк (например, строк 1 и 2) может сложиться ситуация, когда потоки блокируют сначала строки 1 и 2 и только затем переходят к блокировке оставшихся строк (см. рис. 11.6). В этом случае доступ к необходимым строкам не может быть обеспечен ни для одного потока – возникает неразрешимая ситуация, обычно именуемая *тупиком*. Как можно показать, необходимым условием тупика является наличие цикла в графе распределения и запросов ресурсов. В рассматриваемом примере уход от

цикла может состоять в строго последовательной схеме блокировки строк потока.

```
// поток обрабатывает i строку сетки
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i-1]);
// <обработка i строки сетки>
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i-1]);
```

(следует отметить, что и эта схема блокировки строк может оказаться тупиковой, если рассматривать модифицированную задачу Дирихле, в которой горизонтальные границы являются «склеенными»).

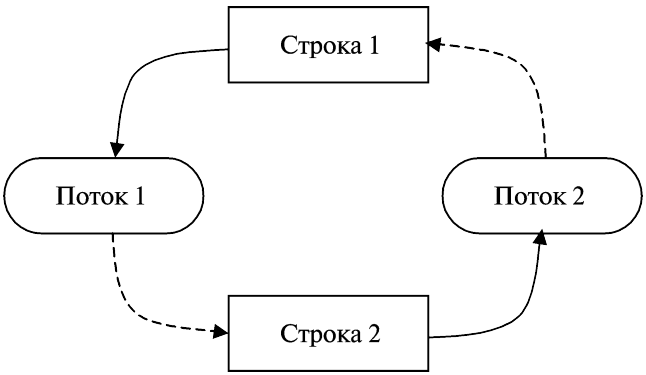


Рис. 11.6. Ситуация тупика при доступе к строкам сетки (поток 1 владеет строкой 1 и запрашивает строку 2, поток 2 владеет строкой 2 и запрашивает строку 1)

11.2.5. Исключение неоднозначности вычислений

Подход, рассмотренный в п. 11.2.4, уменьшает эффект состязания потоков, но не гарантирует единственности решения при повторении вычислений. Для достижения однозначности необходимо использование дополнительных вычислительных схем.

Возможный и широко применяемый в практике расчетов способ состоит в разделении места хранения результатов вычислений на предыдущей и текущей итерациях метода сеток. Схема такого подхода может быть представлена в следующем общем виде:

Алгоритм 11.4. Параллельная реализация сеточного метода Гаусса – Якоби

```
// Алгоритм 11.4
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
#pragma omp parallel for shared(u,un,N,dmax) private(i,temp,d,dm)
  for ( i=1; i<N+1; i++ ) {
    dm = 0;
    for ( j=1; j<N+1; j++ ) {
      temp = u[i][j];
      un[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                  u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-un[i][j])
      if ( dm < d ) dm = d;
    }
    omp_set_lock(dmax_lock);
      if ( dmax < dm ) dmax = dm;
    omp_unset_lock(dmax_lock);
    }
  } // конец параллельной области
  for ( i=1; i<N+1; i++ ) // обновление данных
    for ( j=1; j<N+1; j++ )
      u[i][j] = un[i][j];
} while ( dmax > eps );
```

Как следует из приведенного алгоритма, результаты предыдущей итерации запоминаются в массиве `u`, новые вычисления значения запоминаются в дополнительном массиве `un`. Как результат, независимо от порядка выполнения вычислений для проведения расчетов всегда используются значения величин u_{ij} от предыдущей итерации метода. Такая схема реализации сеточных алгоритмов обычно именуется *методом Гаусса – Якоби*. Этот метод гарантирует однозначность результатов независимо от способа распараллеливания, но требует использования большого дополнительного объема памяти и обладает меньшей (по сравнению с алгоритмом Гаусса – Зейделя) скоростью сходимости. Результаты расчетов с последовательным и параллельным вариантами метода приведены в табл. 11.2.

Таблица 11.2. Результаты вычислительных экспериментов для алгоритма Гаусса – Якоби (p=4)

Размер сетки	Последовательный метод Гаусса – Якоби		Параллельный метод (11.4), разработанный по аналогии с алгоритмом 11.3		
	k	t	k	t	S
100	5257	1,39	5257	0,73	1,90
200	23067	23,84	23067	11,00	2,17
300	26961	226,23	26961	29,00	7,80
400	34377	562,94	34377	66,25	8,50
500	56941	1330,39	56941	191,95	6,93
600	114342	3815,36	114342	2247,95	1,70
700	64433	2927,88	64433	1699,19	1,72
800	87099	5467,64	87099	2751,73	1,99
900	286188	22759,36	286188	11776,09	1,93
1000	152657	14258,38	152657	7397,60	1,93
2000	337809	134140,64	337809	70312,45	1,91
3000	655210	247726,69	655210	129752,13	1,91

(k – количество итераций, t – время (сек), S – ускорение)

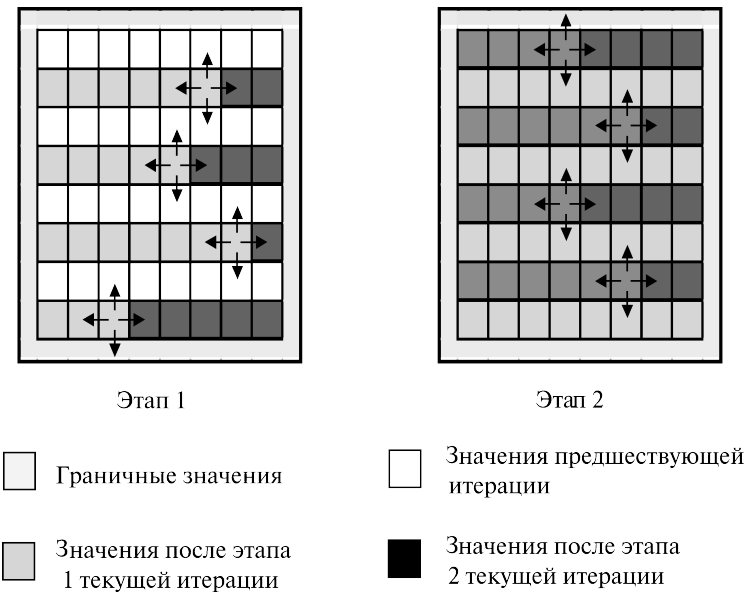


Рис. 11.7. Схема чередования обработки четных и нечетных строк

Иной возможный подход для устранения взаимозависимости параллельных потоков состоит в применении *схемы чередования обработки четных и нечетных строк* (*red/black row alternation scheme*), когда выполнение итерации метода сеток подразделяется на два последовательных этапа, на первом из которых обрабатываются строки только с четными номерами, а затем на втором этапе — строки с нечетными номерами (см. рис. 11.7). Данная схема может быть обобщена на применение одновременно и к строкам, и к столбцам (*шахматное разбиение*) области расчетов.

Рассмотренная схема чередования строк не требует, по сравнению с методом Якоби, какой-либо дополнительной памяти и обеспечивает однозначность решения при многократных запусках программы. Но следует заметить, что оба рассмотренных в данном пункте подхода могут получать результаты, которые не совпадут с решением задачи Дирихле, найденным при помощи последовательного алгоритма. Кроме того, эти вычислительные схемы имеют меньшую область и худшую скорость сходимости, чем исходный вариант метода Гаусса – Зейделя.

11.2.6. Волновые схемы параллельных вычислений

Рассмотрим теперь возможность построения параллельного алгоритма, который выполнял бы только те вычислительные действия, что и последовательный метод (может быть только в несколько ином порядке) и, как результат, обеспечивал бы получение точно таких же решений исходной вычислительной задачи. Как уже было отмечено выше, в последовательном алгоритме каждое очередное k-е приближение значения $u_{i,j}$ вычисляется по последнему k-му приближению значений $u_{i-1,j}$ и $u_{i,j-1}$ и предпоследнему (k-1)-му приближению значений $u_{i+1,j}$ и $u_{i,j+1}$. Таким образом, при требовании совпадения результатов вычислений последовательных и параллельных вычислительных схем в начале каждой итерации метода только одно значение u_{11} может быть пересчитано (возможности для распараллеливания нет). Но далее после пересчета u_{11} вычисления могут выполняться уже в двух узлах сетки u_{12} и u_{21} (в этих узлах выполняются условия последовательной схемы), затем после пересчета узлов u_{12} и u_{21} — в узлах u_{13}, u_{22} и u_{31} и т.д. Обобщая сказанное, можно увидеть, что выполнение итерации метода сеток можно разбить на последовательность шагов, на каждом из которых к вычислениям окажутся подготовленными узлы вспомогательной диагонали сетки с номером, определяемым номером этапа – см. рис. 11.8. Получаемая в результате вычислительная схема получила наименование волны или фронта вычислений, а алгоритмы, построенные на ее основе, — *методов волновой обработки данных* (*wavefront* или *hyperplane methods*). Следует отметить, что в нашем случае размер волны (степень возможного параллелизма) динамически

изменяется в ходе вычислений – волна нарастает до своего пика, а затем затухает при приближении к правому нижнему узлу сетки.

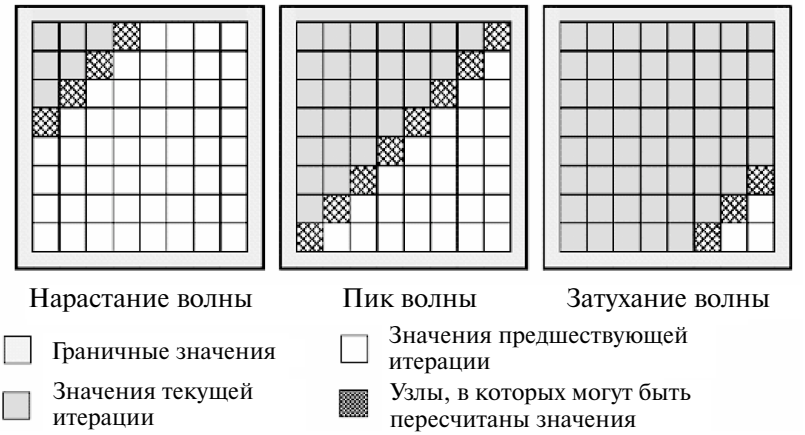


Рис. 11.8. Движение фронта волны вычислений

Возможная схема параллельного метода, основанного на эффекте волны вычислений, может быть представлена в следующей форме.

Алгоритм 11.5. Параллельный алгоритм, реализующий волновую схему вычислений

```
// Алгоритм 11.5
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
  dmax = 0; // максимальное изменение значений u
  // нарастание волны (nx - размер волны)
  for ( nx=1; nx<N+1; nx++ ) {
    dm[nx] = 0;
#pragma omp parallel for shared(u,N,nx,dm) private(i,j,temp,d)
    for ( i=1; i<nx+1; i++ ) {
      j    = nx + 1 - i;
      temp  = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
```

```
  }
  // затухание волны
  for ( nx=N-1; nx>0; nx-- ) {
#pragma omp parallel for shared(u,N,nx,dm) private(i,j,temp,d)
    for ( i=N-nx+1; i<N+1; i++ ) {
      j    = 2*N - nx - I + 1;
      temp = u[i][j];
      u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+
                u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
      d = fabs(temp-u[i][j])
      if ( dm[i] < d ) dm[i] = d;
    } // конец параллельной области
  }
#pragma omp parallel for shared(nx,dm,dmax) private(i)
  for ( i=1; i<nx+1; i++ ) {
    omp_set_lock(dmax_lock);
      if ( dmax < dm[i] ) dmax = dm[i];
    omp_unset_lock(dmax_lock);
  } // конец параллельной области
} while ( dmax > eps );
```

При разработке алгоритма, реализующего волновую схему вычислений, оценку погрешности решения можно осуществлять для каждой строки в отдельности (массив значений `dm`). Этот массив является общим для всех выполняемых потоков, однако синхронизации доступа к элементам не требуется, так как потоки используют всегда разные элементы массива (фронт волны вычислений содержит только по одному узлу строк сетки).

После обработки всех элементов волны в составе массива `dm` находится максимальная погрешность выполненной итерации вычислений. Однако именно эта последняя часть расчетов может оказаться наиболее неэффективной из-за высоких дополнительных затрат на синхронизацию. Улучшение ситуации, как и ранее, может быть достигнуто за счет увеличения размера последовательных участков и сокращения, тем самым, количества необходимых взаимодействий параллельных участков вычислений. Возможный вариант реализации такого подхода может состоять в следующем:

```
chunk = 200; // размер последовательного участка
#pragma omp parallel for shared(n,dm,dmax) private(i,d)
  for ( i=1; i<nx+1; i+=chunk ) {
    d = 0;
```

```
    for ( j=i; j<i+chunk; j++ )
      if ( d < dm[j] ) d = dm[j];
    omp_set_lock(dmax_lock);
      if ( dmax < d ) dmax = d;
    omp_unset_lock(dmax_lock);
  } // конец параллельной области
```

Таблица 11.3. Результаты экспериментов для параллельных вариантов алгоритма Гаусса – Зейделя с волновой схемой расчета (p=4)

Размер сетки	**Последовательный метод Гаусса – Зейделя (алгоритм 11.1)**		**Параллельный алгоритм 11.5**			**Параллельный алгоритм 11.6**		
	k	t	k	t	S	k	t	S
100	210	0,06	210	0,30	0,21	210	0,16	0,40
200	273	0,34	273	0,86	0,40	273	0,59	0,58
300	305	0,88	305	1,63	0,54	305	1,53	0,57
400	318	3,78	318	2,50	1,51	318	2,36	1,60
500	343	6,00	343	3,53	1,70	343	4,03	1,49
600	336	8,81	336	5,20	1,69	336	5,34	1,65
700	344	12,11	344	8,13	1,49	344	10,00	1,21
800	343	16,41	343	12,08	1,36	343	12,64	1,30
900	358	20,61	358	14,98	1,38	358	15,59	1,32
1000	351	25,59	351	18,27	1,40	351	19,30	1,33
2000	367	106,75	367	69,08	1,55	367	65,72	1,62
3000	370	243,00	370	149,36	1,63	370	140,89	1,72

(k – количество итераций, t – время (сек), S – ускорение)

Подобный прием укрупнения последовательных участков вычислений для снижения затрат на синхронизацию именуется *фрагментированием* (*chunking*). Результаты экспериментов для данного варианта параллельных вычислений приведены в табл. 11.3.

Следует обратить внимание еще на один момент при анализе эффективности разработанного параллельного алгоритма. Фронт волны вычислений плохо соответствует правилам использования *кэша* — быстродействующей дополнительной памяти компьютера, используемой для хранения копии данных, хранимых в оперативной памяти. Применение кэша может существенно повысить (в десятки раз) быстродействие вычислений. Размещение данных в кэше может происходить или

предварительно (при использовании тех или иных алгоритмов предсказания потребности в данных), или в момент извлечения значений из основной оперативной памяти. При этом подкачка данных в кэш осуществляется не одиночными значениями, а небольшими группами – *строками кэша* (*cache line*). Загрузка значений в строку кэша осуществляется из последовательных элементов памяти; типовые размеры строки кэша обычно равны 32, 64, 128, 256 байтам (дополнительная информация по организации памяти может быть получена, например, в [16]). Как результат, эффект наличия кэша будет наблюдаться, если выполняемые вычисления используют одни и те же данные многократно (*локальность обработки* данных) и осуществляют доступ к элементам памяти с последовательно возрастающими адресами (*последовательность доступа*).

В исследуемом нами алгоритме размещение данных в памяти осуществляется по строкам, а фронт волны вычислений располагается по диагонали сетки, и это приводит к низкой эффективности использования кэша. Возможный способ улучшения ситуации – опять же укрупнение вычислительных операций и рассмотрение в качестве распределяемых между процессорами действий процедуры обработки некоторой прямоугольной подобласти (*блока*) сетки области расчетов – (см. рис. 11.9).

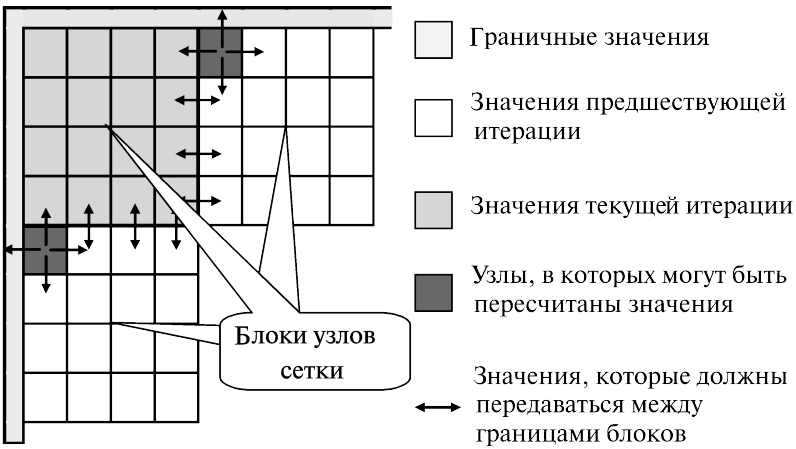


Рис. 11.9. Блочное представление сетки области расчетов

Порождаемый на основе такого подхода метод вычислений в самом общем виде может быть описан следующим образом (блоки образуют в области расчетов прямоугольную решетку размера NBxNB):

Алгоритм 11.6. Блочный подход к методу волновой обработки данных

```
// Алгоритм 11.6
// NB количество блоков
do {
  // нарастание волны (размер волны равен nx+1)
  for ( nx=0; nx<NB; nx++ ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = nx - i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // затухание волны
  for ( nx=NB-2; nx>-1; nx-- ) {
#pragma omp parallel for shared(nx) private(i,j)
    for ( i=0; i<nx+1; i++ ) {
      j = 2*(NB-1) - nx - i;
      // <обработка блока с координатами (i,j)>
    } // конец параллельной области
  }
  // <определение погрешности вычислений>
} while ( dmax > eps );
```

Вычисления в предлагаемом алгоритме происходят в соответствии с волновой схемой обработки данных: вначале вычисления выполняются только в левом верхнем блоке с координатами (0,0), далее для обработки становятся доступными блоки с координатами (0,1) и (1,0) и т.д. – см. результаты экспериментов в табл. 11.3.

Блочный подход к методу волновой обработки данных существенным образом меняет состояние дел – обработку узлов можно организовать построчно, доступ к данным осуществляется последовательно по элементам памяти, перемещенные в кэш значения используются многократно. Кроме того, поскольку обработка блоков будет выполняться на разных процессорах и блоки не пересекаются по данным, при таком подходе будут отсутствовать и накладные расходы для обеспечения однозначности (когерентности) кэшей разных процессоров.

Наилучшие показатели применения кэша будут достигаться, если в кэше будет достаточно места для размещения не менее трех строк блока (при обработке строки блока используются данные трех строк блока одновременно). Тем самым, исходя из размера кэша, можно определить ре-

комендуемый максимально возможный размер блока. Например, при кэше 8 Кб и 8-байтовых значениях данных этот размер составит приближенно 300 (8Кб/3)/8). Можно определить и минимально допустимый размер блока из условия совпадения размеров строк кэша и блока. Так, при размере строки кэша 256 байт и 8-байтовых значениях данных размер блока должен быть кратен 32.

Последнее замечание необходимо сделать о взаимодействии граничных узлов блоков. Учитывая граничное взаимодействие, соседние блоки целесообразно обрабатывать на одних и тех же процессорах. В противном случае можно попытаться так определить размеры блоков, чтобы объем пересылаемых между процессорами граничных данных был минимален. Например, при размере строки кэша в 256 байт, 8-байтовых значениях данных и размере блока 64x64 объем пересылаемых данных – 132 строки кэша, при размере блока 128x32 – всего 72 строки. Такая оптимизация имеет наиболее принципиальное значение при медленных операциях пересылки данных между кэшами процессоров, т.е. для систем с неоднородным доступом к памяти (*nonuniform memory access — NUMA*).

11.2.7. Балансировка вычислительной нагрузки процессоров

Как уже отмечалось ранее, вычислительная нагрузка при волновой обработке данных изменяется динамически в ходе вычислений. Данный момент следует учитывать при распределении вычислительной нагрузки между процессорами. Так, например, при фронте волны из 5 блоков и при использовании 4 процессоров обработка волны потребует двух параллельных итераций, во время второй из которых будет задействован только один процессор, а все остальные процессоры будут простаивать, дожидаясь завершения вычислений. Достигнутое ускорение расчетов в этом случае окажется равным 2,5 вместо потенциально возможного значения 4. Подобное снижение эффективности использования процессоров становится менее заметным при большой длине волны, размер которой может регулироваться размером блока. Как общий результат, можно отметить, что размер блока определяет *степень разбиения* (*granularity*) вычислений для распараллеливания и является параметром, подбирая значения которого можно управлять эффективностью параллельных вычислений.

Для обеспечения равномерности (*балансировки*) загрузки процессоров можно задействовать еще один подход, широко используемый для организации параллельных вычислений. Этот подход состоит в том, что все готовые к выполнению в системе вычислительные действия организуются в виде *очереди заданий*. В ходе вычислений освободившийся процессор может запросить для себя работу из этой очереди; появляющиеся по мере

обработки данных новые вычислительные задания пополняют очередь заданий. Такая схема балансировки вычислительной нагрузки между процессорами является простой, наглядной и эффективной. Это позволяет говорить об использовании очереди заданий как *общей модели организации параллельных вычислений* для систем с общей памятью.

Указанная схема балансировки может быть применена и для рассматриваемого примера. В самом деле, в ходе обработки фронта текущей волны происходит постепенное формирование блоков следующей волны вычислений. Эти блоки могут быть задействованы для обработки при нехватке достаточной вычислительной нагрузки для процессоров.

Общая схема вычислений с использованием очереди заданий может быть представлена в следующем виде:

Алгоритм 11.7. Общая схема вычислений с использованием очереди

```
// Алгоритм 11.7
// <инициализация служебных данных>
// <загрузка в очередь указателя на начальный блок>
// взять блок из очереди (если очередь не пуста)
while ( (pBlock=GetBlock()) != NULL ) {
  // <обработка блока>
  // отметка готовности соседних блоков
  omp_set_lock(pBlock->pNext.Lock); // сосед справа
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 )
      PutBlock(pBlock->pNext);
  omp_unset_lock(pBlock->pNext.Lock);
  omp_set_lock(pBlock->pDown.Lock); // сосед снизу
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 )
      PutBlock(pBlock->pDown);
  omp_unset_lock(pBlock->pDown.Lock);
} // завершение вычислений, т.к. очередь пуста
```

Для описания имеющихся в задаче блоков узлов сетки в алгоритме используется структура со следующим набором параметров:

— `Lock` – семафор, синхронизирующий доступ к описанию блока;
— `pNext` – указатель на соседний справа блок;
— `pDown` – указатель на соседний снизу блок;
— `Count` – счетчик готовности блока к вычислениям (количество готовых границ блока).

Операции для выборки из очереди и вставки в очередь указателя на готовый к обработке блок узлов сетки обеспечивают соответственно функции `GetBlock` и `PutBlock`.

Как следует из приведенной схемы, процессор извлекает блок для обработки из очереди, выполняет необходимые вычисления для блока и отмечает готовность своих границ для соседних справа и снизу блоков. Если при этом оказывается, что у соседних блоков являются подготовленными обе границы, процессор передает эти блоки для запоминания в очередь заданий.

Использование очереди заданий позволяет решить практически все оставшиеся вопросы организации параллельных вычислений для систем с общей памятью. Развитие рассмотренного подхода может предусматривать уточнение правил выделения заданий из очереди для согласования с состояниями процессоров (близкие блоки целесообразно обрабатывать на одних и тех же процессорах), расширение числа имеющихся очередей заданий и т.п. Дополнительная информация по этим вопросам может быть получена, например, в [63, 76].

11.3. Организация параллельных вычислений для систем с распределенной памятью

Использование процессоров с распределенной памятью является другим общим способом построения многопроцессорных вычислительных систем. Актуальность их становится все более высокой в последнее время в связи с широким развитием высокопроизводительных кластерных вычислительных систем (см. лекцию 1).

Многие проблемы параллельного программирования (состязание вычислений, тупики, сериализация) являются общими для систем с общей и распределенной памятью. Основной момент, который отличает параллельные вычисления с распределенной памятью, состоит в том, что взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений* (*message passing*).

Следует отметить, что вычислительный узел системы с распределенной памятью является, как правило, более сложным вычислительным устройством, чем процессор в многопроцессорной системе с общей памятью. Для учета этих различий в дальнейшем процессор с распределенной памятью будет именоваться *вычислительным сервером* (сервером может быть, в частности, многопроцессорная система с общей памятью). При проведении всех ниже рассмотренных экспериментов использовались 4 компьютера с процессорами Pentium IV, 1300 MHz, 256 RAM, 100 Mbit Fast Ethernet.

11.3.1. Общие принципы распределения данных

Первая проблема, которую приходится решать при организации параллельных вычислений на системах с распределенной памятью, обычно состоит в выборе способа разделения обрабатываемых данных между вычислительными серверами. Успешность такого разделения определяется достигнутой степенью локализации вычислений на серверах (в силу больших временных задержек при передаче сообщений интенсивность взаимодействия серверов должна быть минимальной).

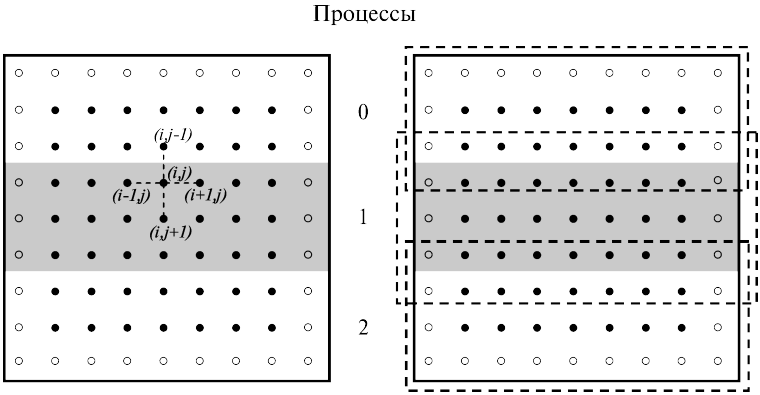


Рис. 11.10. Ленточное разделение области расчетов между процессорами (кружки представляют граничные узлы сетки)

В рассматриваемом учебном примере по решению задачи Дирихле возможны два различных способа разделения данных – *одномерная* или *ленточная* схема (см. рис. 11.10) и *двумерное* или *блочное* разбиение (см. рис. 11.9) вычислительной сетки. Дальнейшее изложение учебного материала будет проводиться на примере первого подхода; блочная схема будет рассмотрена позднее в более кратком виде.

При ленточном разбиении область расчетов делится на горизонтальные или вертикальные полосы (не уменьшая общности, далее будем рассматривать только горизонтальные полосы). Число полос определяется количеством процессоров, размер полос обычно является одинаковым, узлы горизонтальных границ (первая и последняя строки) включаются в первую и последнюю полосы соответственно. Полосы для обработки распределяются между процессорами.

Основной момент при организации вычислений с подобным разделением данных состоит в том, что на процессор, выполняющий об-

работку какой-либо полосы, должны быть продублированы граничные строки предшествующей и следующей полос вычислительной сетки (получаемые в результате расширенные полосы показаны на рис. 11.10 справа пунктирными рамками). Продублированные граничные строки полос используются только при проведении расчетов, пересчет же этих строк происходит в полосах своего исходного месторасположения. Тем самым, дублирование граничных строк должно осуществляться перед началом выполнения каждой очередной итерации метода сеток.

11.3.2. Обмен информацией между процессорами

Параллельный вариант метода сеток при ленточном разделении данных состоит в обработке полос на всех имеющихся серверах одновременно в соответствии со следующей схемой работы:

Алгоритм 11.8. Параллельный алгоритм, реализующий метод сеток при ленточном разделении данных

```
// Алгоритм 11.8
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // <обмен граничных строк полос с соседями>
  // <обработка полосы>
  // <вычисление общей погрешности вычислений dmax>}
 while ( dmax > eps ); // eps - точность решения
```

Для конкретизации представленных в алгоритме действий введем обозначения:

- `ProcNum` – номер процессора, на котором выполняются описываемые действия,
- `PrevProc`, `NextProc` – номера соседних процессоров, содержащих предшествующую и следующую полосы,
- `NP` – количество процессоров,
- `M` – количество строк в полосе (без учета продублированных граничных строк),
- `N` – количество внутренних узлов в строке сетки (т.е. всего в строке N+2 узла).

При нумерации строк полосы будем считать, что строки 0 и $M+1$ есть продублированные из соседних полос граничные строки, а строки собственной полосы процессора имеют номера от 1 до M.

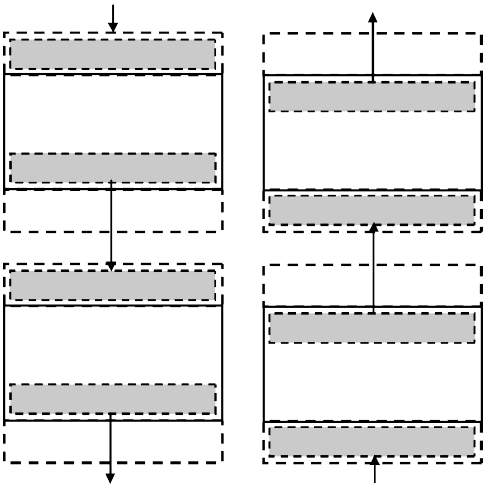


Рис. 11.11. Схема передачи граничных строк между соседними процессорами

Процедура обмена граничных строк между соседними процессорами может быть разделена на две последовательные операции, во время первой из которых каждый процессор передает свою нижнюю граничную строку следующему процессору и принимает такую же строку от предыдущего процессора (см. рис. 11.11). Вторая часть передачи строк выполняется в обратном направлении: процессоры передают свои верхние граничные строки своим предыдущим соседям и принимают переданные строки от следующих процессоров.

Выполнение подобных операций передачи данных в общем виде может быть представлено следующим образом (для краткости рассмотрим только первую часть процедуры обмена):

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
```

(для записи процедур приема-передачи используется близкий к стандарту MPI (см. лекцию 5) формат, где первый и второй параметры представляют пересылаемые данные и их объем, а третий параметр определяет адресата (для операции `Send`) или источник (для операции `Receive`) пересылки данных).

Для передачи данных могут быть задействованы два различных механизма. При первом из них выполнение программ, инициировавших операцию передачи, приостанавливается до полного завершения всех действий по пересылке данных (т.е. до момента получения процессором-адресатом всех передаваемых ему данных). Операции приема-передачи, реализуемые подобным образом, обычно называются *синхронными* или *блокирующими*. Иной подход – *асинхронная* или *неблокирующая* передача — может состоять в том, что операции приема-передачи только инициируют процесс пересылки и на этом завершают свое выполнение. В результате программы, не дожидаясь завершения длительных коммуникационных операций, могут продолжать свои вычислительные действия, проверяя по мере необходимости готовность передаваемых данных. Оба эти варианта операций передачи широко используются при организации параллельных вычислений и имеют свои достоинства и свои недостатки. Синхронные процедуры передачи, как правило, более просты для применения и более надежны; неблокирующие операции могут позволить совместить процессы передачи данных и вычислений, но обычно приводят к повышению сложности программирования. С учетом вышесказанного во всех последующих примерах для организации пересылки данных будут использоваться операции приема-передачи блокирующего типа.

Приведенная выше последовательность блокирующих операций приема-передачи данных (вначале `Send`, затем `Receive`) приводит к строго последовательной схеме выполнения процесса пересылок строк, т.к. все процессоры одновременно обращаются к операции `Send` и переходят в режим ожидания. Первым процессором, который окажется готовым к приему пересылаемых данных, будет сервер с номером *NP*-1. В результате процессор *NP*-2 выполнит операцию передачи своей граничной строки и перейдет к приему строки от процессора *NP*-3 и т.д. Общее количество повторений таких операций равно *NP*-1. Аналогично происходит выполнение и второй части процедуры пересылки граничных строк перед началом обработки строк (см. рис. 11.11).

Последовательный характер рассмотренных операций пересылок данных определяется выбранным способом очередности выполнения. Изменим этот порядок очередности при помощи чередования приема и передачи для процессоров с четными и нечетными номерами.

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
if ( ProcNum % 2 == 1 ) { //  нечетный процессор
```

```
  if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
  if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
}
else { //  процессор с четным номером
  if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
  if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
}
```

Данный прием позволяет выполнить все необходимые операции передачи всего за два последовательных шага. На первом шаге все процессоры с нечетными номерами отправляют данные, а процессоры с четными номерами осуществляют прием этих данных. На втором шаге роли процессоров меняются – четные процессоры выполняют `Send`, нечетные процессоры исполняют операцию приема `Receive`.

Рассмотренные последовательности операций приема-передачи для взаимодействия соседних процессоров широко используются в практике параллельных вычислений. Как результат, во многих базовых библиотеках параллельных программ имеются процедуры для поддержки подобных действий. Так, в стандарте MPI (см. лекцию 5) предусмотрена операция `Sendrecv`, с использованием которой предыдущий фрагмент программного кода может быть записан более кратко:

```
// передача нижней граничной строки следующему
// процессору и прием передаваемой строки от
// предыдущего процессора
Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
```

Реализация подобной объединенной функции `Sendrecv` обычно осуществляется таким образом, чтобы обеспечить и корректную работу на крайних процессорах, когда не нужно выполнять одну из операций передачи или приема, и организацию чередования процедур передачи на процессорах для ухода от тупиковых ситуаций, и возможности параллельного выполнения всех необходимых пересылок данных.

11.3.3. Коллективные операции обмена информацией

Для завершения круга вопросов, связанных с параллельной реализацией метода сеток на системах с распределенной памятью, осталось рассмотреть способы вычисления общей для всех процессоров погрешности вычислений. Возможный очевидный подход состоит в передаче всех локальных оценок погрешности, полученных на отдельных полосах сетки, на один какой-либо процессор, вычислении на нем максимального зна-

чения и последующей рассылке полученного значения всем процессорам системы. Однако такая схема является крайне неэффективной – количество необходимых операций передачи данных определяется числом процессоров и выполнение этих операций может происходить только в последовательном режиме. Между тем, как показывает анализ требуемых коммуникационных действий, выполнение операций сборки и рассылки данных может быть реализовано с использованием рассмотренной в п. 2.5.2 *каскадной схемы* обработки данных. На самом деле, получение максимального значения локальных погрешностей, вычисленных на каждом процессоре, может быть обеспечено, например, путем предварительного нахождения максимальных значений для отдельных пар процессоров (такие вычисления могут выполняться параллельно), затем может быть снова осуществлен попарный поиск максимума среди полученных результатов и т.д. Всего, как полагается по каскадной схеме, необходимо выполнить $\log_2 NP$ параллельных итераций для получения конечного значения (NP – количество процессоров).

С учетом возможности применения каскадной схемы для выполнения коллективных операций передачи данных большинство базовых библиотек параллельных программ содержит процедуры для поддержки подобных действий. Так, в стандарте MPI (см. лекцию 5) предусмотрены операции:

— `Reduce(dm,dmax,op,proc)` – процедура сборки на процессоре *proc* итогового результата *dmax* среди локальных на каждом процессоре значений *dm* с применением операции *op*;

— `Broadcast(dmax,proc)` – процедура рассылки с процессора *proc* значения *dmax* всем имеющимся процессорам системы.

С учетом перечисленных процедур общая схема вычислений на каждом процессоре может быть представлена в следующем виде:

```
// Алгоритм 11.8 – уточненный вариант
// схема Гаусса-Зейделя, ленточное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // обмен граничных строк полос с соседями
  Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
  Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
  // <обработка полосы с оценкой погрешности dm>
  // вычисление общей погрешности вычислений dmax
  Reduce(dm,dmax,MAX,0);
  Broadcast(dmax,0);
} while ( dmax > eps ); // eps – точность решения
```

(в приведенном алгоритме переменная *dm* представляет собой локальную погрешность вычислений на отдельном процессоре, а параметр MAX задает операцию поиска максимального значения для операции сборки). Следует отметить, что в составе MPI имеется процедура `Allreduce`, которая совмещает действия редукции и рассылки данных. Результаты экспериментов для данного варианта параллельных вычислений для метода Гаусса – Зейделя приведены в табл. 11.4.

11.3.4. Организация волны вычислений

Представленные в пп. 11.3.1 – 11.3.3 алгоритмы определяют общую схему параллельных вычислений для метода сеток в многопроцессорных системах с распределенной памятью. Далее эта схема может быть конкретизирована реализацией практически всех вариантов методов, рассмотренных для систем с общей памятью (применение дополнительной памяти для схемы Гаусса – Якоби, чередование обработки полос и т.п.). Проработка таких вариантов не привносит каких-либо новых эффектов с точки зрения параллельных вычислений, и их разбор может использоваться как тема заданий для самостоятельных упражнений.

Таблица 11.4. Результаты экспериментов для систем с распределенной памятью, ленточная схема разделения данных (p=4)

Размер сетки	**Последовательный метод Гаусса – Зейделя (алгоритм 11.1)**		**Параллельный алгоритм 11.8**			**Параллельный алгоритм с волновой схемой расчета (см. п. 11.3.4)**		
	k	t	k	t	S	k	t	S
100	210	0,06	210	0,54	0,11	210	1,27	0,05
200	273	0,35	273	0,86	0,41	273	1,37	0,26
300	305	0,92	305	0,92	1,00	305	1,83	0,50
400	318	1,69	318	1,27	1,33	318	2,53	0,67
500	343	2,88	343	1,72	1,68	343	3,26	0,88
600	336	4,04	336	2,16	1,87	336	3,66	1,10
700	344	5,68	344	2,52	2,25	344	4,64	1,22
800	343	7,37	343	3,32	2,22	343	5,65	1,30
900	358	9,94	358	4,12	2,41	358	7,53	1,32
1000	351	11,87	351	4,43	2,68	351	8,10	1,46
2000	367	50,19	367	15,13	3,32	367	27,00	1,86
3000	364	113,17	364	37,96	2,98	364	55,76	2,03

(k – количество итераций, t – время (сек), S – ускорение)

В завершение рассмотрим возможность организации параллельных вычислений, при которых обеспечивалось бы нахождение таких же решений задачи Дирихле, что и при использовании исходного последовательного метода Гаусса – Зейделя. Как отмечалось ранее, такой результат может быть получен за счет организации волновой схемы расчетов. Для образования волны вычислений представим логически каждую полосу узлов области расчетов в виде набора блоков (размер блоков можно положить, в частности, равным ширине полосы) и организуем обработку полос поблочно в последовательном порядке (см. рис. 11.12). Тогда для полного повторения действий последовательного алгоритма вычисления могут быть начаты только для первого блока первой полосы узлов; после того как этот блок будет обработан, для вычислений будут готовы уже два блока – блок 2 первой полосы и блок 1 второй полосы (для обработки блока полосы 2 необходимо передать граничную строку узлов первого блока полосы 1). После обработки указанных блоков к вычислениям будут готовы уже 3 блока, и мы получаем знакомый уже процесс волновой обработки данных (результаты экспериментов см. в табл. 11.4).

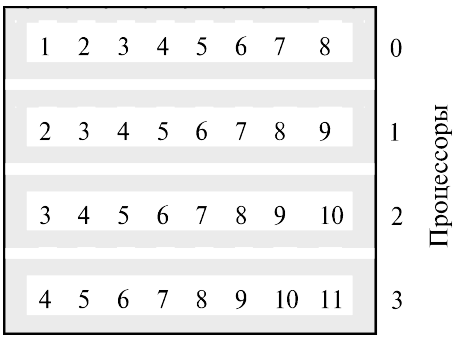


Рис. 11.12. Организация волны вычислений при ленточной схеме разделения данных

Интересный момент при организации подобной схемы параллельных вычислений может состоять в попытке совмещения операций пересылки граничных строк и действий по обработке блоков данных.

11.3.5. Блочная схема разделения данных

Ленточная схема разделения данных может быть естественным образом обобщена на блочный способ представления сетки области расчетов (см. рис. 11.9). При этом столь радикальное изменение способа разбиения сетки практически не потребует каких-либо существенных

корректировок рассмотренной схемы параллельных вычислений. Основной новый момент при блочном представлении данных состоит в увеличении количества граничных строк на каждом процессоре (для блока их количество становится равным 4), что приводит, соответственно, к большему числу операций передачи данных при обмене граничных строк. Сравнивая затраты на организацию передачи граничных строк, можно отметить, что при ленточной схеме для каждого процессора выполняется 4 операции приема-передачи данных, в каждой из которых пересылаются (N+2) значения. Для блочного же способа происходит 8 операций пересылки и объем каждого сообщения равен ($N/\sqrt{NP}$+2) (N – количество внутренних узлов сетки, NP – число процессоров, размер всех блоков предполагается одинаковым). Тем самым, блочная схема представления области расчетов становится оправданной при большом количестве узлов сетки, когда увеличение количества коммуникационных операций приводит к снижению затрат на пересылку данных в силу сокращения размеров передаваемых сообщений. Результаты экспериментов при блочной схеме разделения данных приведены в табл. 11.5.

Таблица 11.5. Результаты экспериментов для систем с распределенной памятью, блочная схема разделения данных (p=4)

Размер сетки	**Последовательный метод Гаусса – Зейделя (алгоритм 11.1)**		**Параллельный алгоритм с с блочной схемой рассчета (см. п. 11.3.5)**			**Параллельный алгоритм 11.9**		
	k	t	k	t	S	k	t	S
100	210	0,06	210	0,71	0,08	210	0,60	0,10
200	273	0,35	273	0,74	0,47	273	1,06	0,33
300	305	0,92	305	1,04	0,88	305	2,01	0,46
400	318	1,69	318	1,44	1,18	318	2,63	0,64
500	343	2,88	343	1,91	1,51	343	3,60	0,80
600	336	4,04	336	2,39	1,69	336	4,63	0,87
700	344	5,68	344	2,96	1,92	344	5,81	0,98
800	343	7,37	343	3,58	2,06	343	7,65	0,96
900	358	9,94	358	4,50	2,21	358	9,57	1,04
1000	351	11,87	351	4,90	2,42	351	11,16	1,06
2000	367	50,19	367	16,07	3,12	367	39,49	1,27
3000	364	113,17	364	39,25	2,88	364	85,72	1,32

(k – количество итераций, t – время (сек), S – ускорение)

При блочном представлении сетки может быть реализован также и волновой метод выполнения расчетов (см. рис. 11.13). Пусть процессоры образуют прямоугольную решетку размером $NB \times NB$ ($NB=\sqrt{NP}$) и процессоры пронумерованы от 0 слева направо по строкам решетки.

Общая схема параллельных вычислений в этом случае имеет вид:

Алгоритм 11.9. Блочная схема разделения данных

```
// Алгоритм 11.9
// схема Гаусса-Зейделя, блочное разделение данных
// действия, выполняемые на каждом процессоре
do {
  // получение граничных узлов
  if ( ProcNum / NB != 0 ) { // строка не нулевая
    // получение данных от верхнего процессора
    Receive(u[0][*],M+2,TopProc);   // верхняя строка
    Receive(dmax,1,TopProc);        // погрешность
  }
  if ( ProcNum % NB != 0 ) { // столбец не нулевой
    // получение данных от левого процессора
    Receive(u[*][0],M+2,LeftProc);  // левый столбец
    Receive(dm,1,LeftProc);         // погрешность
    if ( dm > dmax ) dmax = dm;
  }
  // <обработка блока с оценкой погрешности dmax>
  // пересылка граничных узлов
  if ( ProcNum / NB != NB-1 ) { // строка решетки не последняя
    // пересылка данных нижнему процессору
    Send(u[M+1][*],M+2,DownProc);   // нижняя строка
    Send(dmax,1,DownProc);          // погрешность
  }
  if ( ProcNum % NB != NB-1 ) { // столбец решетки не последний
    // пересылка данных правому процессору
    Send(u[*][M+1],M+2,RightProc);  // правый столбец
    Send(dmax,1, RightProc);        // погрешность
  }
  // синхронизация и рассылка погрешности dmax
  Broadcast(dmax,NP-1);
} while ( dmax > eps ); // eps - точность решения
```

При реализации алгоритма необходимо обеспечить, чтобы в начальный момент времени все процессоры (кроме процессора с нулевым номером) оказались в состоянии передачи своих граничных узлов (верхней строки и левого столбца). Вычисления должен начинать процессор с левым верхним блоком, после завершения обработки которого обновленные значения правого столбца и нижней строки блока нужно переправить правому и нижнему процессорам решетки соответственно. Данные действия обеспечат снятие блокировки процессоров второй диагонали процессорной решетки (ситуация слева на рис. 11.13) и т.д.

Анализ эффективности организации волновых вычислений в системах с распределенной памятью (см. табл. 11.5) показывает значительное снижение полезной вычислительной нагрузки для процессоров, которые занимаются обработкой данных только в моменты, когда их блоки попадают во фронт волны вычислений. При этом балансировка (перераспределение) нагрузки является крайне затруднительной, поскольку связана с пересылкой между процессорами блоков данных большого объема. Возможный интересный способ улучшения ситуации состоит в организации *множественной волны вычислений*, в соответствии с которой процессоры после отработки волны текущей итерации расчетов могут приступить к выполнению волны следующей итерации метода сеток. Так, например, процессор 0 (см. рис. 11.13), передав после обработки своего блока граничные данные и запустив тем самым вычисления на процессорах 1 и 4, оказывается готовым к исполнению следующей итерации метода Гаусса – Зейделя. После обработки блоков первой (процессоры 1 и 4) и второй (процессор 0) волн к вычислениям окажутся готовыми следующие группы процессоров (для первой волны — процессоры 2, 5 и 8, для второй — процессоры 1 и 4). Кроме того, процессор 0 опять окажется готовым к запуску очередной волны обработки данных. После выполнения *NB* подобных шагов в обработке будет находиться одновременно *NB* итераций и все процессоры окажутся задействованными. Подобная схема организации расчетов позволяет рассматривать имеющуюся процессорную решетку как *вычислительный конвейер* поэтапного выполнения итераций метода сеток. Остановка конвейера может осуществляться, как и ранее, по максимальной погрешности вычислений (проверку условия остановки следует начинать только при достижении полной загрузки конвейера после запуска *NB* итераций расчетов). Необходимо отметить также, что получаемое после выполнения условия остановки решение задачи Дирихле будет содержать значения узлов сетки от разных итераций метода и не будет, тем самым, совпадать с решением, получаемым при помощи исходного последовательного алгоритма.

Рис. 11.13. Организация волны вычислений при блочной схеме разделения данных

11.3.6. Оценка трудоемкости операций передачи данных

Время выполнения коммуникационных операций значительно превышает длительность вычислительных команд. Оценка трудоемкости операций приема-передачи может быть осуществлена с использованием двух основных характеристик сети передачи: *латентности* (*latency*), определяющей время подготовки данных к передаче по сети, и *пропускной способности сети* (*bandwidth*), задающей объем передаваемых по сети за 1 секунду данных, – более полное изложение вопроса содержится в лекции 3.

Пропускная способность наиболее распространенной на данный момент сети Fast Ethernet – 100 Мбит/с, для более современной сети Gigabit Ethernet – 1000 Мбит/с. В то же время скорость передачи данных в системах с общей памятью обычно составляет сотни и тысячи миллионов байт в секунду. Тем самым, использование систем с распределенной памятью приводит к снижению скорости передачи данных не менее чем в 100 раз.

Еще хуже дело обстоит с латентностью. Для сети Fast Ethernet эта характеристика имеет значение порядка 150 мкс, для сети Gigabit Ethernet – около 100 мкс. Для современных компьютеров с тактовой частотой свыше 2 ГГц различие в производительности достигает не менее чем 10000 – 100000 раз. При указанных характеристиках вычислительной системы для достижения 90% эффективности в рассматриваемом примере решения задачи Дирихле (т.е. чтобы в ходе расчетов обработка данных занимала не менее 90% времени от общей длительности вычислений и только 10% времени тратилось бы на операции передачи данных) размер блоков вычислительной сетки должен быть не менее N=7500 узлов по вертикали и

горизонтали (объем вычислений в блоке составляет $5N^2$ операций с плавающей запятой).

Как результат, можно заключить, что эффективность параллельных вычислений при использовании распределенной памяти определяется в основном интенсивностью и видом выполняемых коммуникационных операций при взаимодействии процессоров. Необходимый при этом анализ параллельных методов и программ может быть выполнен значительно быстрее за счет выделения типовых операций передачи данных – см. лекцию 3. Так, например, в рассматриваемом учебном примере решения задачи Дирихле практически все пересылки значений сводятся к стандартным коммуникационным действиям, имеющим адекватную поддержку в стандарте MPI (см. рис. 11.14):

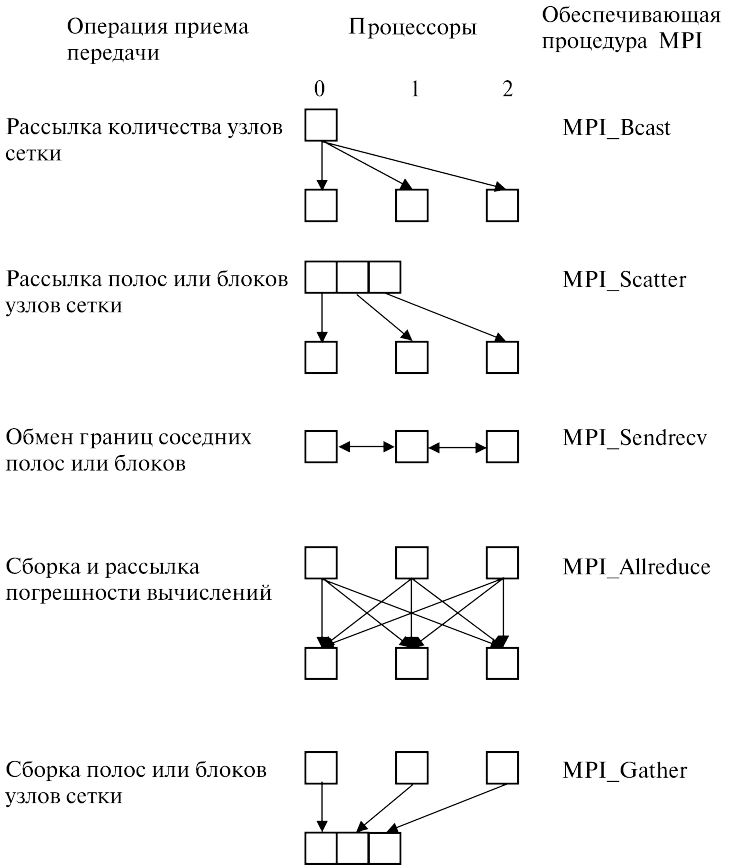


Рис. 11.14. Операции передачи данных при выполнении метода сеток в системе с распределенной памятью

— рассылка количества узлов сетки всем процессорам – типовая операция передачи данных от одного процессора всем процессорам сети (функция `MPI_Bcast`);

— рассылка полос или блоков узлов сетки всем процессорам – типовая операция передачи разных данных от одного процессора всем процессорам сети (функция `MPI_Scatter`);

— обмен граничных строк или столбцов сетки между соседними процессорами – типовая операция передачи данных между соседними процессорами сети (функция `MPI_Sendrecv`);

— сборка и рассылка погрешности вычислений всем процессорам – типовая операция передачи данных от всех процессоров всем процессорам сети (функция `MPI_Allreduce`);

— сборка на одном процессоре решения задачи (всех полос или блоков сетки) – типовая операция передачи данных от всех процессоров сети одному процессору (функция `MPI_Gather`).

11.4. Краткий обзор лекции

В лекции рассматриваются вопросы организации параллельных вычислений для решения задач, в которых при математическом моделировании используются дифференциальные уравнения в частных производных. Для численного решения подобных задач обычно применяется *метод конечных разностей* (*метод сеток*), обладающий высокой вычислительной трудоемкостью. В лекции последовательно разбираются возможные способы распараллеливания сеточных методов на многопроцессорных вычислительных системах с общей и распределенной памятью. При этом большое внимание уделяется проблемам, возникающим при организации параллельных вычислений, анализу причин появления таких проблем и нахождению путей их преодоления. Для наглядной демонстрации излагаемого материала в качестве учебного примера рассматривается проблема численного решения задачи Дирихле для уравнения Пуассона.

В пункте 11.1 приводится краткое описание сеточных методов на примере решения задачи Дирихле.

В пункте 11.2 даются возможные способы организации параллельных вычислений при численном решении дифференциальных уравнений в частных производных для вычислительных систем с общей памятью. В основе излагаемого подхода – технология OpenMP, широко применяемая в настоящее время для разработки параллельных программ. В рамках этой технологии параллельный программный код формируется программистом посредством добавления специальных директив или комментариев в существующие последовательные программы. Как результат, программный код является единым для последовательных и параллельных про-

грамм, что делает более простым развитие и сопровождение программного обеспечения.

Следует отметить, что принятая в лекции последовательность представления учебного материала может быть рассмотрена как наглядная демонстрация поэтапной методики разработки программного обеспечения. Такой подход позволяет достаточно быстро получать начальные варианты параллельных программ, которые далее могут совершенствоваться для достижения максимально возможной эффективности параллельных вычислений. В ходе изложения учебного материала в лекции проводится последовательное развитие параллельной программы для решения задачи Дирихле; для каждого очередного варианта программы проводится анализ порождаемых программой параллельных вычислений, определяются причины имеющихся потерь эффективности расчетов и обосновываются пути дальнейшего совершенствования вычислений. Подобный порядок расположения материала позволяет последовательно показать ряд типовых проблем параллельного программирования – *излишней синхронизации* (*serialization*), *состязания потоков* (*race condition*), *тупиков* (*deadlock*) и др. Особое внимание уделяется проблеме возможной неоднозначности результатов последовательных и параллельных вычислений. Для достижения однозначности получаемых результатов расчетов в приводимом учебном материале оценивается возможность применения нескольких различных подходов, последовательный анализ которых приводит к определению *методов волновой обработки данных* (*wavefront or hyperplane methods*). На примере реализации волновых схем вычислений дается блочная схема представления данных для эффективного использования быстрой кэш-памяти компьютера. В завершение лекции излагается методика организации очередей заданий для равномерной балансировки вычислительной нагрузки процессоров.

В пункте 11.3 вопросы организации параллельных вычислений при численном решении дифференциальных уравнений в частных производных рассматриваются применительно к вычислительным системам с распределенной памятью. Прежде всего отмечается, что многие проблемы параллельного программирования (состязание вычислений, тупики, сериализация) являются общими для систем с общей и распределенной памятью. Основное отличие параллельных вычислений с распределенной памятью состоит в том, что взаимодействие параллельных участков программы на разных процессорах может быть обеспечено только при помощи *передачи сообщений* (*message passing*). При этом эффективность параллельных вычислений во многом определяется равномерностью распределения обрабатываемых данных между процессорами и достигаемой степенью локализации вычислений.

Изложение учебного материала данного раздела лекции начинается с обсуждения общих принципов распределения данных между процессорами, которые применительно к рассматриваемой учебной задаче Дирихле сводятся к *одномерной* (*ленточной*) схеме или *двумерному* (*блочному*) разбиению области расчетов. Последующее рассмотрение вопросов организации параллельных вычислений проводится в основном на примере ленточной схемы; блочный метод разделения данных представлен в более кратком виде. Среди основных тем, выбранных для обсуждения при изложении ленточной схемы: возможные способы выполнения парных и коллективных операций передачи данных между процессорами, особенности реализации волновых схем вычислений в системах с распределенной памятью, возможность совмещения выполняемых вычислений и операций передачи данных. В завершение лекции проводится сравнительная оценка трудоемкости коммуникационных действий и длительности выполнения вычислительных операций.

11.5. Обзор литературы

Дополнительная информация по численным методам решения дифференциальных уравнений в частных производных может быть получена в [6,13]. Рассмотрение вопросов организации при численном решении дифференциальных уравнений в частных производных проводится в [5, 51, 60, 63].

При рассмотрении вопросов организации памяти компьютеров могут оказаться полезными работы [14, 16].

Технология MPI для разработки параллельных программ рассмотрена в лекции 5.

Более подробное рассмотрение вопросов, связанных с использованием очередей заданий при организации параллельных вычислений, проводится в [5, 76].

11.6. Контрольные вопросы

1. Как определяется задача Дирихле для уравнения Пуассона?
2. В чем состоят основные положения метода конечных разностей?
3. Какие способы распараллеливания сеточных методов могут быть использованы для многопроцессорных вычислительных систем с общей памятью?
4. В каких ситуациях необходима синхронизация параллельных вычислений?
5. Как характеризуется поведение параллельных участков программы при наличии условий состязания потоков?

6. В чем состоит проблема взаимоблокировки?
7. Какие методы могут быть использованы для достижения однозначности результатов параллельных вычислений для сеточных методов?
8. Как изменяется объем вычислений при применении методов волновой обработки данных?
9. Как повысить эффективность методов волновой обработки данных?
10. Как очередь заданий позволяет улучшить балансировку вычислительной нагрузки процессоров?
11. Какие проблемы приходится решать при организации параллельных вычислений на системах с распределенной памятью?
12. Какие основные схемы распределения данных между процессорами могут быть использованы для сеточных методов?
13. Какие основные операции передачи данных используются в параллельных методах решения задачи Дирихле?
14. Каким образом организация множественной волны вычислений позволяет повысить эффективность волновых вычислений в системах с распределенной памятью?

11.7. Задачи и упражнения

1. Выполните реализацию первого и второго вариантов параллельного алгоритма Гаусса – Зейделя для систем с общей памятью. Проведите вычислительные эксперименты и сравните время выполнения разработанных программ.
2. Выполните реализации параллельного алгоритма Гаусса – Зейделя при волновой схеме организации вычислений и блочном представлении обрабатываемых данных. Проведите вычислительные эксперименты при разном размере блоков и сравните получаемые характеристики эффективности параллельных вычислений.
3. Выполните реализацию очереди заданий для параллельного алгоритма Гаусса – Зейделя. Подготовьте несколько разных правил выделения заданий из очереди и проведите оценку эффективности для каждого использованного правила.

Лекция 12. Программная система ПараЛаб для изучения и исследования методов параллельных вычислений

В лекции описывается программная система Параллельная Лаборатория (сокращенное наименование – ПараЛаб), которая предназначена для учебного применения студентами и преподавателями вузов в целях исследования и изучения параллельных алгоритмов решения сложных вычислительных задач в рамках лабораторного практикума по различным учебным курсам в области параллельного программирования.

Ключевые слова: вычислительная задача, метод решения, вычислительная система, топология и характеристики сети передачи данных, передача сообщений и передача пакетов, латентность, пропускная способность, визуализация процесса вычислений, вычислительный эксперимент, показатели эффективности параллельных вычислений, матричные вычисления, сортировка, обработка графов.

12.1. Введение

Программная система Параллельная Лаборатория (сокращенное наименование – ПараЛаб) обеспечивает возможность проведения вычислительных экспериментов с целью изучения и исследования параллельных алгоритмов решения сложных вычислительных задач. Система может быть использована для организации лабораторного практикума по различным учебным курсам в области параллельного программирования, в рамках которого обеспечивается возможность:

- *моделирования многопроцессорных вычислительных систем* с различной топологией сети передачи данных;
- *получения визуального представления* о вычислительных процессах и операциях передачи данных, происходящих при параллельном решении разных вычислительных задач;
- *построения оценок эффективности* изучаемых методов параллельных вычислений.

Проведение такого практикума может быть организовано на «обычных» однопроцессорных компьютерах, работающих под управлением операционных систем MS Windows 2000 или MS Windows XP (режим многозадачной имитации параллельных вычислений). Кроме режима имитации, в системе ПараЛаб может быть обеспечен удаленный доступ к многопроцессорной вычислительной системе для выполнения эксперимен-

тов в режиме «настоящих» параллельных вычислений для сопоставления результатов имитации и реальных расчетов.

В целом система ПараЛаб представляет собой *интегрированную среду для изучения и исследования параллельных алгоритмов* решения сложных вычислительных задач. Широкий набор имеющихся средств визуализации процесса выполнения эксперимента и анализа полученных результатов позволяет изучить эффективность использования тех или иных алгоритмов на разных вычислительных системах, сделать выводы о масштабируемости алгоритмов и определить возможное ускорение процесса параллельных вычислений.

Реализуемые системой ПараЛаб процессы изучения и исследований ориентированы на активное усвоение основных теоретических положений и способствуют формированию у обучаемых собственных представлений о моделях и методах параллельных вычислений путем наблюдения, сравнения и сопоставления широкого набора различных визуальных графических форм, демонстрируемых в ходе выполнения вычислительного эксперимента.

Основной сферой использования системы ПараЛаб является *учебное применение* студентами и преподавателями вузов для исследования и изучения параллельных алгоритмов решения сложных вычислительных задач в рамках лабораторного практикума по различным учебным курсам в области параллельного программирования. Система ПараЛаб может применяться также и при проведении *научных исследований* для оценки эффективности параллельных вычислений.

Пользователи, начинающие знакомиться с проблематикой параллельных вычислений, найдут систему ПараЛаб полезной для освоения методов параллельного программирования, опытные вычислители могут использовать систему для оценки эффективности новых разрабатываемых параллельных алгоритмов.

12.2. Общая характеристика системы

ПараЛаб — программная система, которая позволяет как проводить реальные параллельные вычисления на многопроцессорной вычислительной системе, так и имитировать такие эксперименты на одном последовательном компьютере с визуализацией процесса параллельного решения сложной вычислительной задачи.

При проведении имитационных экспериментов ПараЛаб предоставляет для пользователя возможность:

- *определить топологию* параллельной вычислительной системы для проведения экспериментов, *задать число процессоров* в этой топологии, *установить производительность процессоров*, вы-

брать *характеристики коммуникационной среды* и способ *коммуникации*;

- *осуществить постановку вычислительной задачи*, для которой в составе системы ПараЛаб имеются реализованные параллельные алгоритмы решения, *выполнить задание параметров* задачи;
- *выбрать параллельный метод* для решения выбранной задачи;
- *установить параметры визуализации* для выбора желаемого темпа демонстрации, способа отображения пересылаемых между процессорами данных, степени детальности визуализации выполняемых параллельных вычислений;
- *выполнить эксперимент* для параллельного решения выбранной задачи; при этом в системе ПараЛаб может быть сформировано несколько различных *заданий для проведения экспериментов* с отличающимися типами многопроцессорных систем, задач или методов параллельных вычислений, для которых выполнение эксперимента может происходить одновременно (в режиме разделения времени). Одновременное выполнение эксперимента для нескольких заданий позволяет наглядно сравнивать динамику решения задачи различными методами, на разных топологиях, с разными параметрами исходной задачи. При выполнении *серии экспериментов*, требующих длительных вычислений, в системе имеется возможность их проведения в автоматическом режиме с запоминанием результатов для организации последующего анализа полученных данных;
- *накапливать и анализировать результаты выполненных экспериментов*; по запомненным результатам в системе имеется возможность построения графиков зависимостей характеристик параллельных вычислений (*времени решения, ускорения, эффективности*) от параметров задачи и вычислительной системы.

Одной из важнейших характеристик системы является возможность выбора способов проведения экспериментов. Эксперимент может быть выполнен в *режиме имитации*, т.е. проведен на одном процессоре без использования каких-либо специальных программных средств типа библиотек передачи сообщений. Кроме того, в рамках системы ПараЛаб обеспечивается возможность проведения реального *вычислительного эксперимента*.

При построении зависимостей временных характеристик от параметров задачи и вычислительной системы для экспериментов, выполненных в режиме имитации, используются теоретические оценки в соответствии с имеющимися моделями параллельных вычислений (см., например, [2, 94]). Для реальных экспериментов на многопроцессорных вычислительных системах зависимости строятся по набору результатов проведенных вычислительных экспериментов.

Важно отметить, что в системе ПараЛаб обеспечена возможность запоминания результатов проведенных экспериментов в специальной области памяти. Запомненные результаты позволяют выполнить анализ полученных данных; по имеющейся информации любой из проведенных ранее экспериментов может быть восстановлен для повторного выполнения или продолжения расчетов.

Реализованные таким образом процессы изучения и исследований позволят усвоить теоретические положения и помогут формированию представлений о методах построения параллельных алгоритмов, ориентированных на решение конкретных прикладных задач.

Демонстрационный пример

Для выполнения примера, имеющегося в комплекте поставки системы:
- выберите пункт меню **Начало** и выполните команду **Загрузить**;
- выберите строку **first.prl** в списке имен файлов и нажмите кнопку **Открыть**;
- выберите пункт меню **Выполнение** и выполните команду **В активном окне**.

В результате этих действий на экране дисплея будет представлено окно для выполнения вычислительного эксперимента (рис. 12.1). В этом окне демонстрируется решение задачи умножения матриц при помощи ленточного алгоритма.

В области «Выполнение эксперимента» представлены процессоры вычислительной системы и структура линий коммутации. Рядом с каждым процессором изображены те данные, которые он обрабатывает в каждый момент выполнения алгоритма (для ленточного алгоритма умножения матриц это несколько последовательных строк матрицы A и несколько последовательных столбцов матрицы B). При помощи перемещающихся прямоугольников (пакетов) желтого цвета изображается обмен данными, который осуществляют процессоры.

В области «Результат умножения матриц» изображается текущее состояние матрицы – результата умножения. Поскольку результатом перемножения полос исходных матриц A и B является блок матрицы C, получаемая результирующая матрица имеет блочную структуру. Темно-синим цветом обозначены уже вычисленные блоки, голубым цветом выделены блоки, еще подлежащие определению.

В списке «Выполняемая задача» представлены параметры решаемой задачи: название, метод решения, объем исходных данных. В списке «Вычислительная система» приводятся атрибуты выбранной вычислительной системы: топология, количество и производительность процессоров, характеристики сети.

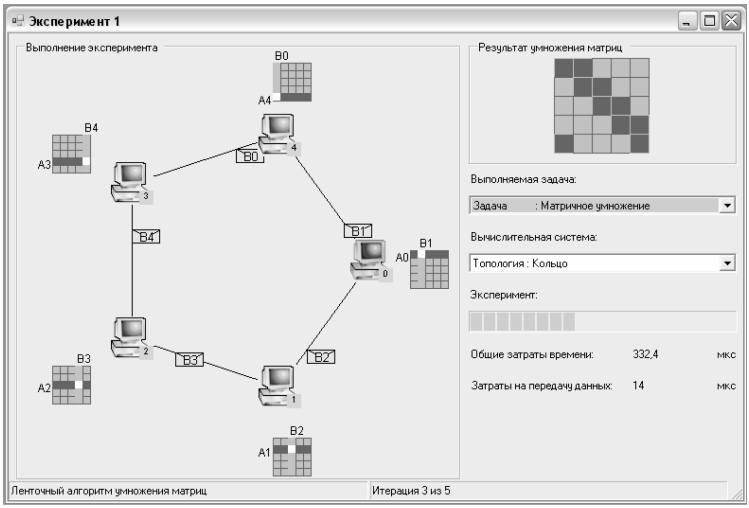


Рис. 12.1. Окно вычислительного эксперимента

Ленточный индикатор «Эксперимент» отображает текущую стадию выполнения алгоритма. В строках «Общие затраты времени» и «Затраты на передачу данных» представлены временные характеристики алгоритма.

После выполнения эксперимента (восстанавливается главное меню системы) можно завершить работу системы. Для этого выберите пункт меню **Архив** и выполните команду **Завершить**.

12.3. Формирование модели вычислительной системы

Для формирования модели вычислительной системы необходимо определить топологию сети, количество процессоров, производительность каждого процессора и характеристики коммуникационной среды (латентность, пропускную способность и метод передачи данных). Следует отметить, что в рамках системы ПараЛаб вычислительная система полагается однородной, т.е. все процессоры обладают одинаковой производительностью, а все каналы связи имеют одинаковые характеристики.

12.3.1. Выбор топологии сети

Топология сети передачи данных определяется структурой линий коммутации между процессорами вычислительной системы. В системе ПараЛаб обеспечивается поддержка следующих типовых топологий:

- **полный граф** (*completely-connected graph* или *clique*) – система, в которой между любой парой процессоров существует прямая линия связи; как результат, данная топология обеспечивает минимальные затраты при передаче данных, однако является сложно реализуемой при большом количестве процессоров;
- **линейка** (*linear array* или *farm*) – система, в которой каждый процессор имеет линии связи только с двумя соседними (с предыдущим и последующим) процессорами; такая схема является, с одной стороны, просто реализуемой, с другой стороны, соответствует структуре передачи данных при решении многих вычислительных задач (например, при организации конвейерных вычислений);
- **кольцо** (*ring*) – данная топология получается из линейки процессоров соединением первого и последнего процессоров линейки;
- **решетка** (*mesh*) – система, в которой граф линий связи образует прямоугольную двумерную сетку; подобная топология может быть достаточно просто реализована и, кроме того, может эффективно использоваться при параллельном выполнении многих численных алгоритмов (например, при реализации методов блочного умножения матриц);
- **гиперкуб** (*hypercube*) – данная топология представляет частный случай структуры N-мерной решетки, когда по каждой размерности сетки имеется только два процессора (т.е. гиперкуб содержит 2^N процессоров при размерности N); данный вариант организации сети передачи данных достаточно широко распространен на практике и характеризуется следующим рядом отличительных признаков:
  - два процессора имеют соединение, если двоичное представление их номеров имеет 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 различных таких разбиений);
  - кратчайший путь между двумя любыми процессорами имеет длину, совпадающую с количеством различающихся битовых значений в номерах процессоров (данная величина известна как расстояние Хэмминга).

Правила использования системы ПараЛаб

1. Запуск системы. Для запуска системы ПараЛаб выделите пиктограмму системы и выполните двойной щелчок левой кнопкой мыши (или нажмите клавишу **Enter**). Далее выполните команду **Выполнить новый эксперимент** (пункт меню **Начало**) и нажмите в диалоговом окне **Название эксперимента** кнопку **OK** (при желании до нажатия кнопки **OK**

может быть изменено название создаваемого окна для проведения экспериментов).

2. Выбор топологии вычислительной системы. Для выбора топологии вычислительной системы следует выполнить команду **Топология** пункта меню **Система**. В появившемся диалоговом окне (рис. 12.2) щелкните левой клавишей мыши на пиктограмме нужной топологии или внизу в области соответствующей круглой кнопки выбора (радиокнопки). При нажатии кнопки **Помощь** можно получить справочную информацию о реализованных топологиях. Нажмите кнопку **OK** для подтверждения выбора и кнопку **Отмена** для возврата в основное меню системы ПараЛаб.

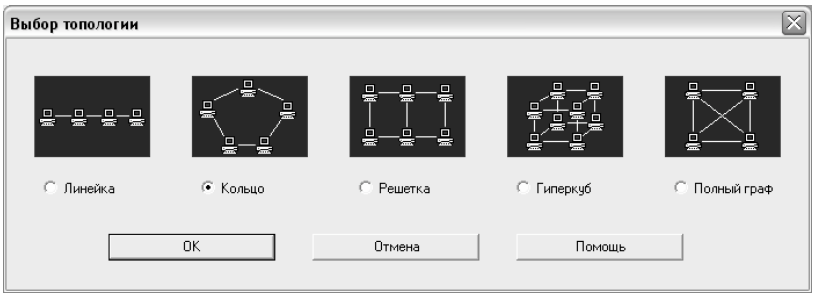


Рис. 12.2. Диалоговое окно для выбора топологии

12.3.2. Задание количества процессоров

Для выбранной топологии система ПараЛаб позволяет установить необходимое количество процессоров. Выполняемый при этом выбор конфигурации системы осуществляется в соответствии с типом используемой топологии. Так, например, число процессоров в двумерной решетке должно являться полным квадратом (размеры решетки по горизонтали и вертикали совпадают), а число процессоров в гиперкубе – степенью числа 2.

Под *производительностью процессора* в системе ПараЛаб понимается количество операций с плавающей запятой, которое процессор может выполнить за секунду (floating point operations per second – flops). Важно отметить, что при построении оценок времени выполнения эксперимента предполагается, что все машинные команды являются одинаковыми и соответствуют одной и той же операции с плавающей точкой.

Правила использования системы ПараЛаб

1. Задание количества процессоров. Для выбора числа процессоров необходимо выполнить команду **Количество Процессоров** пункта меню **Система**. В появившемся диалоговом окне (рис. 12.3) вам предоставляет-

ся несколько пиктограмм со схематическим изображением числа процессоров в активной топологии. Для выбора щелкните левой клавишей мыши на нужной пиктограмме. Пиктограмма, соответствующая текущему числу процессоров, выделена ярко-синим цветом.

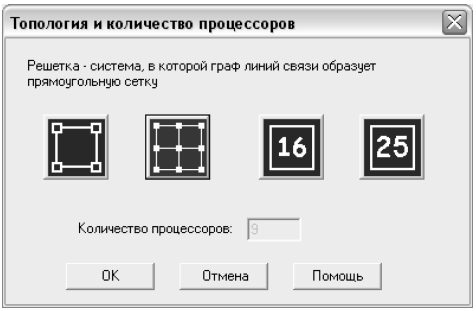


Рис. 12.3. Диалоговое окно для задания числа процессоров

Нажмите кнопку **OK** для подтверждения выбора или кнопку **Отмена** для возврата в основное меню системы ПараЛаб без изменения числа процессоров.

2. Определение производительности процессора. Для задания производительности процессоров, составляющих многопроцессорную вычислительную систему, следует выполнить команду **Производительность Процессора** пункта меню **Система**. Далее в появившемся диалоговом окне (рис. 12.4) при помощи бегунка нужно задать величину производительности. Для подтверждения выбора нажмите кнопку **OK** (или клавишу **Enter**). Для возврата в основное меню системы ПараЛаб без изменений нажмите кнопку **Отмена** (или клавишу **Escape**).

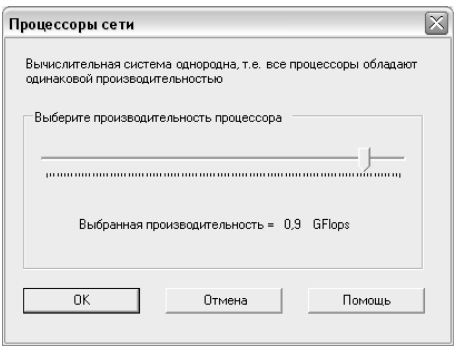


Рис. 12.4. Диалоговое окно для задания производительности процессора

12.3.3. Задание характеристик сети

Время передачи данных между процессорами определяет коммуникационную составляющую (*communication overhead*) длительности выполнения параллельного алгоритма в многопроцессорной вычислительной системе. Основной набор параметров, описывающих время передачи данных, состоит из следующего ряда величин:

— *латентность* ($t_н$) — время начальной подготовки, которое характеризует длительность подготовки сообщения для передачи, поиска маршрута в сети и т.п.;

— *пропускная способность сети* (R) – определяется как максимальный объем данных, который может быть передан за некоторую единицу времени по одному каналу передачи данных. Данная характеристика измеряется, например, количеством переданных бит в секунду.

К числу реализованных в системе ПараЛаб методов передачи данных относятся два следующих широко известных способа коммуникации (см. [51]). Первый из них ориентирован на *передачу сообщений* (МПС) как неделимых (атомарных) блоков информации (*store-and-forward routing* или *SFR*). При таком подходе процессор, содержащий исходное сообщение, готовит весь объем данных для передачи, определяет транзитный процессор, через который данные могут быть доставлены целевому процессору, и запускает операцию пересылки данных. Процессор, которому направлено сообщение, в первую очередь осуществляет прием полностью всех пересылаемых данных и только затем приступает к пересылке принятого сообщения далее по маршруту. Время пересылки данных T для метода передачи сообщения размером m по маршруту длиной l определяется выражением:

$$T = (t_н + (m / R)) \cdot l.$$

Второй способ коммуникации основывается на представлении пересылаемых сообщений в виде блоков информации меньшего размера (*пакетов*), в результате чего передача данных может быть сведена к *передаче пакетов* (МПП). При таком методе коммуникации (*cut-through routing* или *CTR*) транзитный процессор может осуществлять пересылку данных по дальнейшему маршруту непосредственно сразу после приема очередного пакета, не дожидаясь завершения приема данных всего сообщения. Количество передаваемых при этом пакетов равно

$$n = \left\lceil \frac{m}{V - V_0} \right\rceil + 1,$$

где V есть размер пакета, а величина V_0 определяет объем служебных данных, передаваемых в каждом пакете (*заголовок пакета*). Как результат, время передачи сообщения в этом случае составит

$$T = t_н + \frac{V}{R}\left(l + \left\lceil \frac{m}{V - V_0} \right\rceil\right) = t_н + \frac{V}{R}(l + n - 1)$$

(скобки $\lceil\,\rceil$ обозначают операцию приведения к целому с избытком).

Сравнивая полученные выражения, можно заметить, что в случае, когда длина маршрута больше единицы, метод передачи пакетов приводит к более быстрой пересылке данных; кроме того, данный подход снижает потребность в памяти для хранения пересылаемых данных для организации приема-передачи сообщений, а для передачи пакетов могут использоваться одновременно разные коммуникационные каналы.

Правила использования системы ПараЛаб

1. Определение характеристик коммуникационной среды. Для определения характеристик сети выполните команду **Характеристики сети** пункта меню **Система**. В открывшемся диалоговом окне (рис. 12.5) при помощи бегунков можно задать время начальной подготовки данных (латентность) в микросекундах и пропускную способность каналов сети (Мбит/с). Для подтверждения выбора нажмите кнопку **ОК**. Для возврата в основное меню системы ПараЛаб без изменения этих параметров нажмите кнопку **Отмена**.

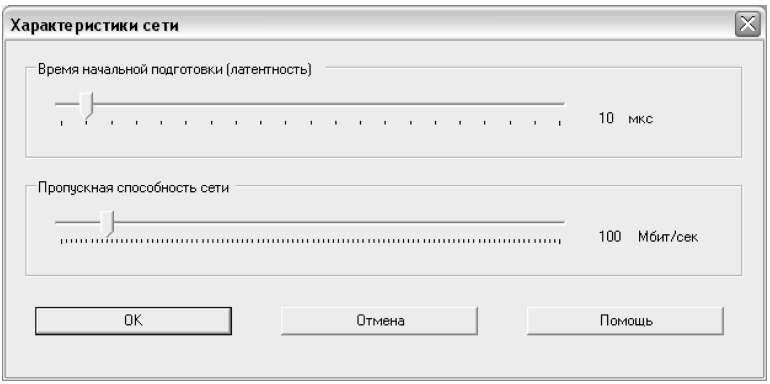


Рис. 12.5. Диалоговое окно для задания характеристик сети

2. Определение метода передачи данных. Для определения метода передачи данных, который будет использоваться при проведении вычислительного эксперимента и при построении временных характеристик, не-

обходимо выполнить команду **Метод передачи данных** пункта меню **Система**. В открывшемся диалоговом окне (рис. 12.6) следует щелкнуть левой клавишей мыши в области радиокнопки, которая соответствует желаемому методу передачи данных. Если выбран метод передачи пакетов, при помощи бегунков возможно задать длину пакета и длину заголовка пакета в байтах. Для подтверждения выбора метода передачи данных и его параметров нажмите кнопку **OK**.

Рис. 12.6. Диалоговое окно для задания метода передачи данных

3. Завершение работы системы. Для завершения работы системы ПараЛаб следует выполнить команду **Завершить** (пункт меню **Архив**).

12.4. Постановка вычислительной задачи и выбор параллельного метода решения

Для параллельного решения тех или иных вычислительных задач процесс вычислений должен быть представлен в виде набора независимых вычислительных процедур, допускающих выполнение на независимых процессорах.

Общая схема организации таких вычислений может быть представлена следующим образом:

- разделение процесса вычислений на части, которые могут быть выполнены одновременно;
- распределение вычислений по процессорам;
- обеспечение взаимодействия параллельно выполняемых вычислений.

Возможные способы получения методов параллельных вычислений:

- разработка новых параллельных алгоритмов;
- распараллеливание последовательных алгоритмов.

Условия эффективности параллельных алгоритмов:

- равномерная загрузка процессоров (отсутствие простоев);
- низкая интенсивность взаимодействия процессоров (независимость).

В системе ПараЛаб реализованы широко применяемые параллельные алгоритмы для решения ряда сложных вычислительных задач из разных областей научно-технических приложений: алгоритмы сортировки данных, матричных операций, решения систем линейных уравнений и обработки графов.

Правила использования системы ПараЛаб

1. Выбор задачи. Для выбора задачи из числа реализованных в системе выберите пункт меню **Задача** и выделите левой клавишей мыши одну из строк: **Сортировка**, **Умножение матрицы на вектор**, **Матричное умножение**, **Решение системы линейных уравнений**, **Обработка графов**. Выбранная задача станет текущей в активном окне.

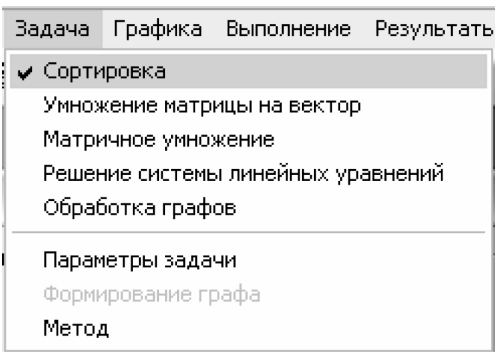


Рис. 12.7. Выбор задачи

2. Определение параметров задачи. Основным параметром задачи в системе ПараЛаб является объем исходных данных. Для задачи сортировки это размер упорядочиваемого массива данных, для матричных операций и задачи решения системы линейных уравнений – размерность исходных матриц, для задачи обработки графов – число вершин в графе. Для выбора параметров задачи необходимо выполнить команду **Параметры задачи** пункта меню **Задача**. В появившемся диалоговом окне (рис. 12.8) следует при помощи бегунка задать необходимый объем исходных данных. Нажмите **OK (Enter)** для подтверждения задания параметра. Для возврата в основное меню системы ПараЛаб без сохранения изменений нажмите **Отмена (Escape)**.

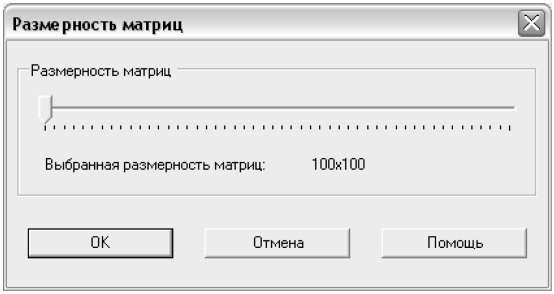


Рис. 12.8. Диалоговое окно задания параметров задачи в случае решения задачи матричного умножения

3. Определение метода решения задачи. Для выбора метода решения задачи выполните команду **Метод** пункта меню **Задача**. В появившемся диалоговом окне (рис. 12.9) выделите мышью нужный метод, нажмите **OK** для подтверждения выбора, нажмите **Отмена** для возврата в основное меню системы ПараЛаб.

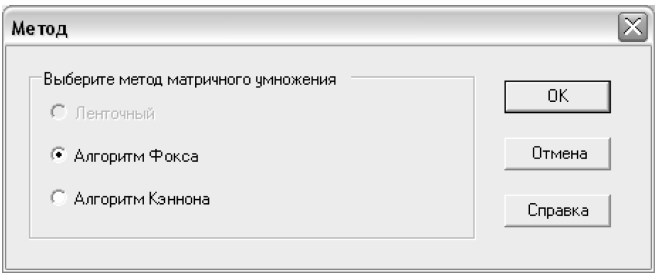


Рис. 12.9. Диалоговое окно выбора метода в случае решения задачи матричного умножения

12.4.1. Сортировка данных

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$S=\{a_1, a_2,\ldots,a_n\}$

в порядке монотонного возрастания или убывания

$$S \sim S'=\{(a_1', a_2',\ldots, a_n') : a_1' \le a_2' \le \ldots \le a_n'\}.$$

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (пузырьковая сортировка, сортировка включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных

$$T_1 \sim n^2.$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T_1 \sim n\log_2 n.$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

Ускорение сортировки может быть обеспечено при использовании нескольких (p, $p>1$) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами, при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении сортировки значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

В системе ПараЛаб в качестве методов упорядочения данных представлены *пузырьковая сортировка, сортировка Шелла, быстрая сортировка*. Исходные (последовательные) варианты этих методов изложены во многих изданиях (см., например, [26]), способы их параллельного выполнения приводятся в лекции 9.

12.4.1.1. Пузырьковая сортировка

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания — сравнение пар соседних элементов происходит строго последовательно. Параллельный вариант алгоритма основывается на *методе чет-нечетной перестановки*, для которого на нечетной итерации выполнения сравниваются элементы пар

$$(a_1,a_2),(a_3,a_4),\ldots,(a_{n-1},a_n).$$

Если пара не упорядочена, то ее элементы переставляются. На четной итерации упорядочиваются пары

$(a_2,a_3),(a_4,a_5),\ldots,(a_{n-2},a_{n-1})$.

После n-кратного повторения подобных итераций массив оказывается отсортированным. Более подробная информация о параллельном варианте алгоритма приводится в лекции 9.

Задания и упражнения

1. Запустите систему ПараЛаб и создайте новый эксперимент. Выберите пункт меню **Задача** и убедитесь, что в активном окне текущей задачей является задача сортировки. Откройте диалоговое окно выбора метода и посмотрите, какие алгоритмы сортировки могут быть выполнены на текущей топологии. Так как при создании эксперимента по умолчанию текущей топологией становится кольцо, то единственный возможный алгоритм – алгоритм сортировки пузырьком. Закройте диалоговое окно и вернитесь в основное меню системы ПараЛаб.

2. Выполните несколько экспериментов, изменяя размер исходных данных. Для выполнения эксперимента выполните команду **В активном окне** пункта меню **Выполнить**. Проанализируйте временные характеристики экспериментов, которые отображаются в правой нижней части окна.

3. Проведите несколько вычислительных экспериментов, изменяя количество процессоров вычислительной системы. Проанализируйте полученные временные характеристики.

12.4.1.2. Сортировка Шелла

Параллельный алгоритм сортировки Шелла может быть получен как обобщение метода параллельной пузырьковой сортировки. Основное различие состоит в том, что на первых итерациях алгоритма Шелла происходит сравнение пар элементов, которые в исходном наборе данных находятся далеко друг от друга (для упорядочивания таких пар в пузырьковой сортировке может понадобиться достаточно большое количество итераций). Подробное описание параллельного обобщения алгоритма сортировки Шелла можно найти в лекции 9.

Задания и упражнения

1. Запустите систему ПараЛаб. Выберите топологию *кольцо* и установите количество процессоров, равное восьми. Проведите эксперимент по выполнению алгоритма пузырьковой сортировки.

2. Установите в окне вычислительного эксперимента топологию *гиперкуб* и число процессоров, равное восьми.

3. Откройте диалоговое окно выбора метода и посмотрите, какие алгоритмы сортировки могут быть выполнены на этой топологии. Выберите метод сортировки Шелла. Закройте окно.

4. Проведите вычислительный эксперимент. Сравните количество итераций, выполненных при решении задачи при помощи метода Шелла, с количеством итераций алгоритма пузырьковой сортировки (количество итераций отображается справа в строке состояния). Убедитесь в том, что при выполнении эксперимента с использованием алгоритма Шелла для сортировки массива необходимо выполнить меньшее количество итераций.

5. Проведите эксперимент с использованием метода Шелла несколько раз. Убедитесь, что количество итераций не является постоянной величиной и зависит от исходного массива.

12.4.1.3. Быстрая сортировка

Алгоритм быстрой сортировки основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности. При параллельном обобщении алгоритма обеспечивается отношение упорядоченности между элементами сортируемого набора, расположенными на процессорах, соседних в структуре гиперкуба. В материалах лекции 9 вы можете найти подробное описание алгоритма быстрой сортировки.

Задания и упражнения

1. Запустите систему ПараЛаб. В появившемся окне вычислительного эксперимента установите топологию *гиперкуб* и количество процессоров, равное восьми.

2. Выполните три последовательных эксперимента с использованием трех различных алгоритмов сортировки: сортировки пузырьком, сортировки Шелла и быстрой сортировки. Сравните временные характеристики алгоритмов, которые отображаются в правой нижней части окна. Убедитесь в том, что у быстрой сортировки наименьшее время выполнения алгоритма и время передачи данных.

3. Измените объем исходных данных (выполните команду **Параметры задачи** пункта меню **Задача**). Снова проведите эксперименты. Сравните временные характеристики алгоритмов.

4. Измените количество процессоров (выполните команду **Количество процессоров** пункта меню **Система**). Проведите вычислительные эксперименты и сравните временные характеристики.

12.4.2. Умножение матрицы на вектор

В результате умножения матрицы A размерности $m \times n$ и вектора b, состоящего из n элементов, получается вектор c размера m, каждый i-й элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора b:

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j \,, 1 \le i \le m \,.$$

Тем самым получение результирующего вектора c предполагает повторение m однотипных операций по умножению строк матрицы A и вектора b. Каждая такая операция включает умножение элементов строки матрицы и вектора b (n операций) и последующее суммирование полученных произведений (n-1 операций). Общее количество необходимых скалярных операций есть величина

$$T_1 = m \cdot (2n - 1).$$

Для операции умножения матрицы на вектор характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Это свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов, и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*).

Более полная информация об алгоритмах матрично-векторного умножения, реализованных в системе ПараЛаб, содержится в лекции 6.

12.4.2.1. Умножение матрицы на вектор при разделении данных по строкам

Данный алгоритм основан на представлении матрицы непрерывными наборами (горизонтальными полосами) строк. Полученные полосы распределяются по процессорам вычислительной системы. Вектор b копируется на все процессоры. Перемножение полосы матрицы на вектор (а данная операция может быть выполнена процессорами параллельно)

приводит к получению блока элементов результирующего вектора *c*. Для объединения результатов расчета и получения полного вектора *c* на каждом из процессоров вычислительной системы необходимо выполнить операцию обобщенного сбора данных.

Задания и упражнения

1. Создайте в системе ПараЛаб новое окно вычислительного эксперимента. Для этого окна выберите задачу умножения матрицы на вектор (щелкните левой кнопкой мыши на строке **Умножение матрицы на вектор** пункта меню **Задача**).

2. Откройте диалоговое окно выбора метода и убедитесь в том, что выбран метод, основанный на горизонтальном разбиении матрицы.

3. Проведите несколько вычислительных экспериментов. Изучите зависимость времени выполнения алгоритма от объема исходных данных и от количества процессоров.

12.4.2.2. Умножение матрицы на вектор при разделении данных по столбцам

Другой подход к параллельному умножению матрицы на вектор основан на разделении исходной матрицы на непрерывные наборы (вертикальные полосы) столбцов. Вектор *b* при таком подходе разделен на блоки. Вертикальные полосы исходной матрицы и блоки вектора распределены между процессорами вычислительной системы.

Параллельный алгоритм умножения матрицы на вектор начинается с того, что каждый процессор *i* выполняет умножение своей вертикальной полосы матрицы *A* на блок элементов вектора *b*, в итоге на каждом процессоре получается вектор промежуточных результатов $c'(i)$. Далее для получения элементов результирующего вектора *c* процессоры должны обменяться своими промежуточными данными между собой.

12.4.2.3. Умножение матрицы на вектор при блочном разделении данных

Рассмотрим теперь параллельный алгоритм умножения матрицы на вектор, который основан на ином способе разделения данных – на разбиении матрицы на прямоугольные фрагменты (*блоки*). При таком способе разделения данных исходная матрица *A* представляется в виде набора прямоугольных блоков. Вектор *b* также должен быть разделен на блоки. Блоки матрицы и блоки вектора распределены между процессорами вычислительной системы. Логическая (виртуальная) топология вычислительной системы в данном случае имеет вид прямоугольной двумерной

решетки. Размеры процессорной решетки соответствуют количеству прямоугольных блоков, на которые разбита матрица A. На процессоре $p_{i,j}$, находящемся на пересечении i-й строки и j-го столбца процессорной решетки, располагается блок $A_{i,j}$ матрицы A и блок b_j вектора b.

После перемножения блоков матрицы A и вектора b каждый процессор $p_{i,j}$ будет содержать вектор частичных результатов $c'(i,j)$. Поэлементное суммирование векторов частичных результатов для каждой горизонтальной строки процессорной решетки позволяет получить результирующий вектор c.

Задания и упражнения

1. Запустите систему ПараЛаб. Установите количество процессоров, равное четырем.

2. Выполните три последовательных эксперимента с использованием трех различных алгоритмов умножения матрицы на вектор — алгоритмов, основанных на горизонтальном, вертикальном и блочном разбиении матрицы. Сравните временные характеристики алгоритмов, которые отображаются в правой нижней части окна. Убедитесь в том, что у алгоритмов, основанных на ленточном разбиении матрицы, время выполнения практически совпадает, а также в том, что время выполнения алгоритма, основанного на блочном разбиении, несколько больше.

3. Измените объем исходных данных (выполните команду **Параметры задачи** пункта меню **Задача**). Снова проведите эксперименты. Сравните временные характеристики алгоритмов.

4. Измените количество процессоров, установите количество процессоров, равное 16 (выполните команду **Количество процессоров** пункта меню **Система**). Проведите вычислительные эксперименты и сравните временные характеристики.

12.4.3. Матричное умножение

Задача умножения матрицы на матрицу определяется соотношениями:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj},\ 1 \le i, j \le n$$

(для простоты изложения материала будем предполагать, что перемножаемые матрицы A и B являются квадратными и имеют порядок $n \times n$). Как следует из приведенных соотношений, вычислительная сложность задачи является достаточно высокой (оценка количества выполняемых операций имеет порядок n^3).

Основу возможности параллельных вычислений для матричного умножения составляет независимость расчетов для получения элементов c_{ij} ре-

зультирующей матрицы *C*. Тем самым, все элементы матрицы *C* могут быть вычислены параллельно при наличии n^2 процессоров, при этом на каждом процессоре будет располагаться по одной строке матрицы *A* и одному столбцу матрицы *B*. При меньшем количестве процессоров подобный подход приводит к *ленточной схеме* разбиения данных, когда на процессорах располагаются по несколько строк и столбцов (полос) исходных матриц.

Другой широко используемый подход для построения параллельных способов выполнения матричного умножения состоит в применении *блочного представления* матриц, при котором исходные матрицы *A, B* и результирующая матрица *C* рассматриваются в виде наборов блоков (как правило, квадратного вида некоторого размера $m \times m$). Тогда операцию матричного умножения матриц *A* и *B* в блочном виде можно представить следующим образом:

$$\begin{pmatrix} A_{11}A_{12}...A_{1k} \\ \cdots \\ A_{k1}A_{k2}...A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11}B_{12}...B_{1k} \\ \cdots \\ B_{k1}B_{k2}...B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11}C_{12}...C_{1k} \\ \cdots \\ C_{k1}C_{k2}...C_{kk} \end{pmatrix},$$

где каждый блок C_{ij} матрицы *C* определяется в соответствии с выражением:

$$C_{ij} = \sum_{l=1}^{k} A_{il} B_{lj}.$$

Полученные блоки C_{ij} также являются независимыми, и, как результат, возможный подход для параллельного выполнения вычислений может состоять в расчетах, связанных с получением отдельных блоков C_{ij}, на разных процессорах. Применение подобного подхода позволяет получить многие эффективные *параллельные методы умножения блочно-представленных матриц*.

В системе ПараЛаб реализованы *параллельный алгоритм умножения матриц при ленточной схеме разделения* данных и два метода (*алгоритмы Фокса и Кэннона*) для блочно представленных матриц. Более полная информация об алгоритмах умножения матриц, реализованных в системе ПараЛаб, содержится в лекции 7.

12.4.3.1. Ленточный алгоритм

При ленточной схеме разделения данных исходные матрицы разбиваются на горизонтальные (для матрицы *A*) и вертикальные (для матрицы *B*) полосы. Получаемые полосы распределяются по процессорам, при этом на каждом из имеющегося набора процессоров располагается только по одной полосе матриц *A* и *B*. Перемножение полос (а данная операция может быть выполнена процессорами параллельно) приводит к получению части бло-

ков результирующей матрицы *C*. Для вычисления оставшихся блоков матрицы *C* сочетания полос матриц *A* и *B* на процессорах должны быть изменены. В наиболее простом виде это может быть обеспечено, например, при кольцевой топологии вычислительной сети (при числе процессоров, равном количеству полос) – в этом случае необходимое для матричного умножения изменение положения данных может быть реализовано циклическим сдвигом полос матрицы *B* по кольцу. После многократного выполнения описанных действий (количество необходимых повторений является равным числу процессоров) на каждом процессоре получается набор блоков, образующий горизонтальную полосу матрицы *C*.

Рассмотренная схема вычислений позволяет определить параллельный алгоритм матричного умножения при ленточной схеме разделения данных как итерационную процедуру, на каждом шаге которой происходит параллельное выполнение операции перемножения полос и последующего циклического сдвига полос одной из матриц по кольцу. Подробное описание ленточного алгоритма приводится в лекции 7.

Задания и упражнения

1. Создайте в системе ПараЛаб новое окно вычислительного эксперимента. Для этого окна выберите задачу матричного умножения (щелкните левой кнопкой мыши на строке **Матричное умножение** пункта меню **Задача**).

2. Откройте диалоговое окно выбора метода и убедитесь в том, что выбран метод ленточного умножения матриц.

3. Проведите несколько вычислительных экспериментов. Изучите зависимость времени выполнения алгоритма от объема исходных данных и от количества процессоров.

12.4.3.2. Блочные алгоритмы Фокса и Кэннона

При блочном представлении данных параллельная вычислительная схема матричного умножения в наиболее простом виде может быть построена, если топология вычислительной сети имеет вид прямоугольной решетки (если реальная топология сети имеет иной вид, представление сети в виде решетки можно обеспечить на логическом уровне). Основные положения параллельных методов для блочно представленных матриц состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы *C*;
- в ходе вычислений на каждом из процессоров располагается по одному блоку исходных матриц *A* и *B*;
- при выполнении итераций алгоритмов блоки матрицы *A* последовательно сдвигаются вдоль строк процессорной решетки, а блоки матрицы *B* — вдоль столбцов решетки;

- в результате вычислений на каждом из процессоров получается блок матрицы C, при этом общее количество итераций алгоритма равно $\sqrt{p}$ (где p – число процессоров).

В лекции 7 приводится полное описание параллельных методов Фокса и Кэннона для умножения блочно-представленных матриц.

Задания и упражнения

1. В активном окне вычислительного эксперимента системы ПараЛаб установите топологию **Решетка**. Выберите число процессоров, равное девяти. Сделайте текущей задачей этого окна задачу матричного умножения.

2. Выберите метод Фокса умножения матриц и проведите вычислительный эксперимент.

3. Выберите алгоритм Кэннона матричного умножения и выполните вычислительный эксперимент. Понаблюдайте различные маршруты передачи данных при выполнении алгоритмов. Сравните временные характеристики алгоритмов.

4. Измените число процессоров в топологии **Решетка** на шестнадцать. Последовательно выполните вычислительные эксперименты с использованием метода Фокса и метода Кэннона. Сравните временные характеристики этих экспериментов.

12.4.4. Решение систем линейных уравнений

Системы линейных уравнений возникают при решении ряда прикладных задач, описываемых системами нелинейных (трансцендентных), дифференциальных или интегральных уравнений. Они могут появляться также в задачах математического программирования, статистической обработки данных, аппроксимации функций, при дискретизации краевых дифференциальных задач методом конечных разностей или методом конечных элементов и т.д.

Линейное уравнение с n неизвестными $x_0, x_1, ..., x_{n-1}$ может быть определено при помощи выражения

$$a_0x_0 + a_1x_1 + ... + a_{n-1}x_{n-1} = b,$$

где величины $a_0, a_1, ..., a_{n-1}$ и b представляют собой постоянные значения.

Множество n линейных уравнений

$$\begin{aligned}
a_{0,0}x_0 \quad &+a_{0,1}x_1 \quad &+...+a_{0,n-1}x_{n-1} \quad &= b_0 \\
a_{1,0}x_0 \quad &+a_{1,1}x_1 \quad &+...+a_{1,n-1}x_{n-1} \quad &= b_1 \\
&... \\
a_{n-1,0}x_0 \quad &+a_{n-1,1}x_1 \quad &+...+a_{n-1,n-1}x_{n-1} \quad &= b_{n-1}
\end{aligned}$$

называется *системой линейных уравнений* или *линейной системой*. В более кратком (матричном) виде система может быть представлена как

$$Ax=b,$$

где $A=(a_{i,j})$ есть вещественная матрица размера $n \times n$, а векторы b и x состоят из n элементов.

Под *задачей решения системы линейных уравнений* для заданных матрицы A и вектора b понимается нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

В системе ПараЛаб реализованы два алгоритма решения систем линейных уравнений: широко известный *метод Гаусса* (прямой метод решения систем линейных уравнений, находит точное решение системы с невырожденной матрицей за конечное число шагов) и *метод сопряженных градиентов* – один из широкого класса итерационных методов решения систем линейных уравнений с матрицей специального вида. Более полная информация об алгоритмах решения систем линейных уравнений, реализованных в системе ПараЛаб, содержится в лекции 8.

12.4.4.1. Алгоритм Гаусса

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе – *прямой ход метода Гаусса* – исходная система линейный уравнений при помощи последовательного исключения неизвестных (при помощи эквивалентных преобразований) приводится к верхнему треугольному виду. На *обратном ходе метода Гаусса* (второй этап алгоритма) осуществляется определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_{n-1}, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-2} и т.д.

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В этом случае матрицу A можно разделить на горизонтальные полосы, а вектор правых частей b – на блоки. Полосы матрицы и блоки вектора правых частей распределяются между процессорами вычислительной системы.

Для выполнения прямого хода метода Гаусса необходимо осуществить $(n-1)$ итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду.

Выполнение итерации i, $0 \le i < n-1$, прямого хода метода Гаусса включает ряд последовательных действий. Процессор, на котором расположена строка i матрицы, должен разослать ее и соответствующий элемент вектора b всем процессорам, которые содержат строки с номерами k, $k > i$. Получив ведущую строку, процессоры выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении обратного хода метода Гаусса процессоры выполняют необходимые вычисления для нахождения значения неизвестных. Как только какой-либо процессор определяет значение своей переменной x_i, это значение должно быть разослано всем процессорам, которые содержат строки с номерами k, $k < i$. Далее процессоры подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

Задания и упражнения

1. В активном окне вычислительного эксперимента системы ПараЛаб установите топологию **Полный граф**. Выберите число процессоров, равное десяти. Сделайте текущей задачей этого окна задачу решения системы линейных уравнений.

2. Выберите метод Гаусса решения задачи и выполните вычислительный эксперимент. Пронаблюдайте маршруты передачи данных при выполнении алгоритма.

12.4.5. Обработка графов

Математические модели в виде графов широко используются при моделировании самых разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов.

В системе ПараЛаб реализованы параллельные алгоритмы решения двух типовых задач на графах: *алгоритм Прима для поиска минимального охватывающего дерева* и *алгоритм Дейкстры для поиска кратчайших путей*. Более полная информация об алгоритмах обработки графов, реализованных в системе ПараЛаб, содержится в лекции 10.

Правила использования системы ПараЛаб

1. Переход в режим редактирования графа. При выборе задачи **Обработка графов** в системе ПараЛаб предусмотрена возможность создания и редактирования графов, запоминания графов в файле и чтения графа из файла. Для того чтобы перейти в режим редактирования графа, выполни-

те команду **Формирование графа** пункта меню **Задача**. Заметим, что команда доступна только в том случае, когда текущей задачей является задача обработки графов.

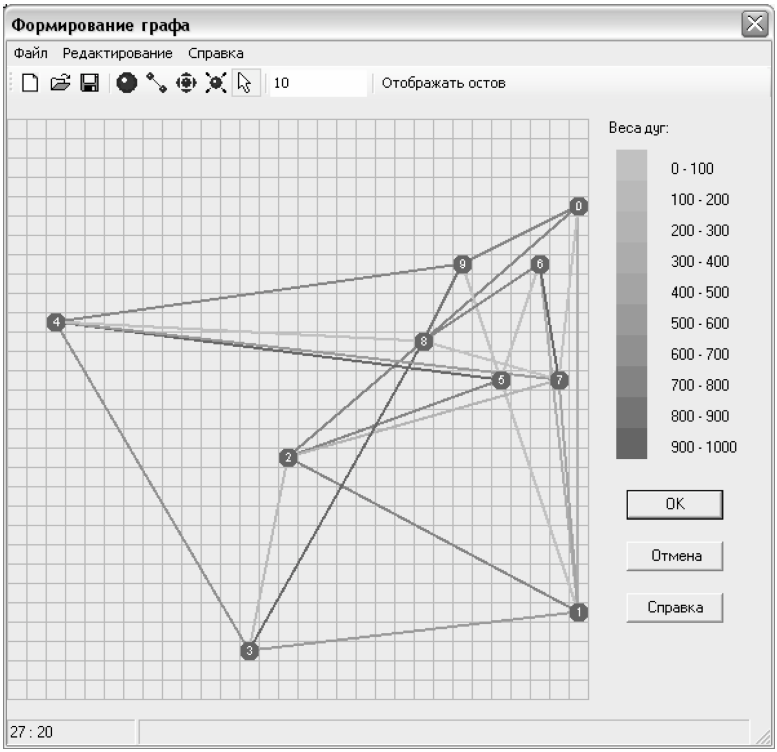


Рис. 12.10. Окно встроенного редактора графов

После выполнения команды **Формирование графа** на экране дисплея появляется новое окно (рис. 12.10), в рабочей области которого отображается граф активного эксперимента. Если граф в эксперимент не был загружен, то рабочая область окна пуста.

Вам предоставляется возможность создавать новые графы, редактировать уже существующие, сохранять новые графы в файлах и загружать графы из файлов в активный эксперимент.

2. Создание нового графа. Для создания нового пустого графа выберите пункт меню **Файл** и выполните команду **Новый** (или щелкните левой кнопкой мыши на иконке 🗋 панели инструментов). Если граф, который отображается в рабочей области, был изменен, то вам будет предложено сохранить измененный граф в файл.

3. Открытие существующего графа. Для загрузки графа из файла выберите пункт меню **Файл** и выполните команду **Загрузить** (или щелкните левой кнопкой мыши на иконке панели инструментов). В появившемся диалоговом окне выберите имя файла (файлы графов ПараЛаб имеют расширение .plg) и нажмите кнопку **Открыть**.

4. Сохранение графа. Для сохранения графа в файл выберите пункт меню **Файл** и выполните команду **Сохранить** (или щелкните левой кнопкой мыши на иконке панели инструментов). В появившемся диалоговом окне введите имя нового файла или выберите какой-либо из существующих файлов для того, чтобы сохранить граф в этом файле. Нажмите кнопку **Сохранить**.

5. Редактирование графа. С графом, расположенным в рабочей области окна, можно производить различные операции.

- Для того чтобы добавить к графу одну или несколько новых вершин, выберите пункт меню **Редактирование** и выполните команду **Добавить вершину** (или щелкните левой кнопкой мыши на иконке панели инструментов). При этом вид курсора изменится, над указателем появится символическое изображение вершины. Рабочая область окна представляет собой сетку. Щелкая левой кнопкой мыши в различных клетках сетки, вы можете добавлять в граф новые вершины. Если вы щелкнули в клетке, где уже есть вершина, то добавления вершины не произойдет.
- Для того чтобы соединить две вершины графа ребром, выберите пункт меню **Редактирование** и выполните команду **Добавить ребро** (или щелкните левой кнопкой мыши на иконке панели инструментов). После этого курсор изменит форму, под указателем появится изображение двух вершин, соединенных ребром. Выделите одну из вершин графа, щелкнув на ней левой кнопкой мыши. Ее цвет изменится на темно-красный. Выделите другую вершину графа – в результате между первой и второй указанными вершинами появится ребро. Вес ребра определяется случайным образом. Если между первой и второй вершинами до редактирования существовало ребро, то оно будет удалено.
- Для того чтобы переместить вершину графа, выберите пункт меню **Редактирование** и выполните команду **Переместить вершину** (или щелкните левой кнопкой мыши на иконке панели инструментов). После этого курсор изменит форму, примет вид, изображенный на кнопке панели инструментов. Выделите одну из вершин графа, щелкнув на ней левой кнопкой мыши. Цвет вершины изменится на темно-красный. Перемещайте курсор мыши по рабочей области окна – вы увидите, что вершина перемещается вслед за курсором. Щелкните на любой пустой

клетке рабочей области, и выделенная вершина переместится в эту точку.

- Для того чтобы удалить вершину графа, выберите пункт меню **Редактирование** и выполните команду **Удалить вершину** (или щелкните левой кнопкой мыши на иконке панели инструментов). После этого курсор изменит вид, под указателем появится пиктограмма перечеркнутой вершины. Щелкните левой кнопкой мыши на любой вершине графа, чтобы удалить ее.

Для выхода из любого из режимов (Добавление вершины, Удаление вершины, Перемещение вершины, Добавление ребра) щелкните левой кнопкой мыши на иконке панели инструментов.

6. Формирование графа при помощи случайного механизма. Граф можно задавать при помощи случайного генератора. Для этого в редакторе, расположенном на панели инструментов, укажите число вершин графа, далее выберите пункт меню **Редактирование** и выполните команду **Случайное формирование**.

7. Редактирование веса ребра графа. Цвет ребер графа имеет разную интенсивность. Чем темнее цвет, тем больше вес ребра. Для того чтобы приблизительно определить вес ребра, нужно сравнить его цвет со шкалой, расположенной справа. Для того чтобы изменить вес ребра, щелкните на нем правой кнопкой мыши. Рядом с ребром появится ползунок. Перемещая его вправо, вы увеличиваете вес ребра, перемещая влево – уменьшаете. Для закрепления изменений щелкните мышкой в любой точке рабочей области или нажмите любую клавишу.

8. Выход из режима редактирования. Для загрузки текущего графа в активный эксперимент нажмите кнопку **OK**. Для выхода без сохранения изменений нажмите кнопку **Отмена**.

12.4.5.1. Алгоритм Прима поиска минимального охватывающего дерева

Охватывающим деревом (или *остовом*) неориентированного графа G называется подграф T графа G, который является деревом и содержит все вершины из G. Определим вес подграфа для взвешенного графа как сумму весов входящих в подграф дуг, тогда под *минимально охватывающим деревом* (*МОД*) T будем понимать охватывающее дерево минимального веса. Содержательная интерпретация задачи нахождения МОД может состоять, например, в практическом примере построения локальной сети персональных компьютеров с прокладыванием наименьшего количества соединительных линий связи.

Дадим краткое описание алгоритма решения поставленной задачи, известного под названием *метода Прима* (*Prim*). Алгоритм начинает рабо-

ту с произвольной вершины графа, выбираемой в качестве корня дерева, и в ходе последовательно выполняемых итераций расширяет конструируемое дерево до МОД. Распределение данных между процессорами вычислительной системы должно обеспечивать независимость перечисленных операций алгоритма Прима. В частности, это может быть реализовано, если каждая вершина графа располагается на процессоре вместе со всей связанной с вершиной информацией.

С учетом такого разделения данных итерация параллельного варианта алгоритма Прима состоит в следующем:

- определяется вершина, имеющая наименьшее расстояние до построенного к этому моменту МОД (операции вычисления расстояния для вершин графа, не включенных в МОД, независимы и, следовательно, могут быть выполнены параллельно);
- найденная вершина включается в состав МОД.

12.4.5.2. Алгоритм Дейкстры поиска кратчайших путей

Задача поиска кратчайших путей на графе состоит в нахождении путей минимального веса от некоторой заданной вершины S до всех имеющихся вершин графа. Постановка подобной проблемы имеет важное практическое значение в различных приложениях, когда веса дуг означают время, стоимость, расстояние, затраты и т.п.

Возможный способ решения поставленной задачи, известный как алгоритм Дейкстры, практически совпадает с методом Прима. Различие состоит лишь в интерпретации и в правиле оценки расстояний. В алгоритме Дейкстры эти величины означают суммарный вес пути от начальной вершины до всех остальных вершин графа. Как результат, на каждой итерации алгоритма выбирается очередная вершина, расстояние от которой до корня дерева минимально, и происходит включение этой вершины в дерево кратчайших путей.

Задания и упражнения

1. Запустите систему ПараЛаб. В активном окне вычислительного эксперимента установите топологию **Полный граф**. Текущей задачей этого окна сделайте задачу обработки графов.

2. Выполните команду **Формирование графа** пункта меню **Задача**. В появившемся редакторе графов сформируйте случайным образом граф с десятью вершинами.

3. Выполните вычислительный эксперимент по поиску минимального охватывающего дерева с помощью алгоритма Прима (выполните команду **Метод** пункта меню **Задача**, в появившемся диалоговом окне выберите **Метод Прима**).

4. Проведите несколько экспериментов, изменяя количество процессоров. Изучите зависимость временных характеристик алгоритма Прима от количества процессоров.

5. Проведите аналогичную последовательность экспериментов для изучения временных характеристик метода Дейкстры.

12.5. Определение графических форм наблюдения за процессом параллельных вычислений

Для наблюдения за процессом выполнения вычислительного эксперимента по параллельному решению сложных вычислительно трудоемких задач в рамках системы ПараЛаб предусмотрены различные формы графического представления результатов параллельных вычислений.

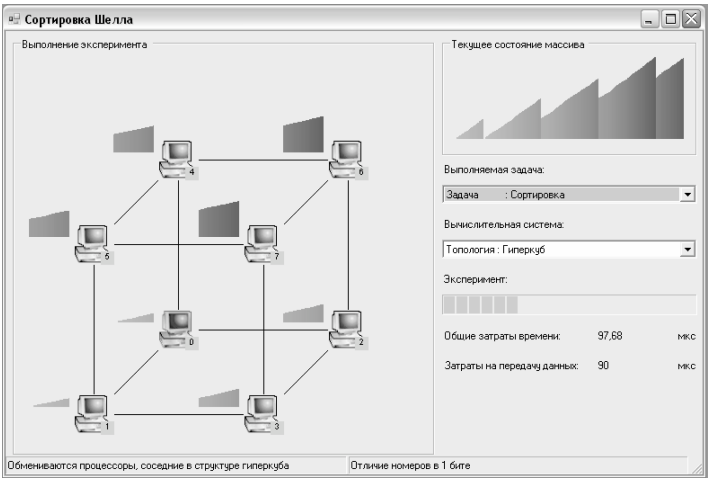


Рис. 12.11. Вид окна вычислительного эксперимента

Для представления сведений о ходе выполнения эксперимента в рабочей области системы ПараЛаб для каждого эксперимента выделяется прямоугольный участок экрана – *окно вычислительного эксперимента*. В левой части окна выделена область «Выполнение эксперимента», где изображаются процессоры многопроцессорной вычислительной системы, объединенные в ту или иную топологию, данные, расположенные на каждом процессоре, и показывается взаимообмен данными между процессорами системы. В правой верхней части окна отображается область с информацией о текущем состоянии объекта, являющегося результатом выполняемого эксперимента. В зависимости от того, какой эксперимент выполняется, эта область носит название:

- «Текущее состояние массива» при выполнении алгоритма сортировки;
- «Результат умножения матриц» при выполнении матричного умножения;
- «Результат обработки графа» при выполнении алгоритмов на графах.

В средней части правой половины окна эксперимента приводятся сведения о выполняемой задаче. Здесь же в списке «Вычислительная система» указаны характеристики вычислительной системы, такие, как топология, количество процессоров, производительность процессоров и характеристики коммуникационной среды.

В правом нижнем углу располагается ленточный индикатор выполнения эксперимента и его текущие временные характеристики.

Дополнительно в отдельном окне могут быть более подробно визуально представлены вычисления, которые производит один из имеющегося набора процессор (задание режима высветки этого окна и выбор наблюдаемого процессора осуществляется пользователем системы).

12.5.1. Область «Выполнение эксперимента»

В этой области окна изображены процессоры многопроцессорной вычислительной системы, соединенные линиями коммутации в ту или иную топологию. Процессоры в топологии пронумерованы. Для того чтобы узнать номер процессора, достаточно навести на него указатель мыши. Вид указателя изменится, и появится подсказка с номером процессора. Если при этом дважды щелкнуть левой клавишей мыши на изображении процессора, то появится окно «Демонстрация работы процессора», где будет детально отображаться деятельность этого процессора.

Около каждого процессора схематически изображаются данные, которые находятся на нем в данный момент выполнения эксперимента. В процессе эксперимента в области «Выполнение эксперимента» также отображается обмен данными между процессорами многопроцессорной вычислительной системы. Это может происходить в двух режимах:

- режим **«Выделение каналов»** — выделяются красным цветом те линии коммутации, по которым происходит обмен;
- режим **«Движение пакетов»** — визуализация обмена при помощи движущихся между процессорами прямоугольников (конвертов). Если изучаются параллельные алгоритмы матричного умножения, то на конверте изображается номер блока, который пересылается.

При выполнении алгоритмов на графах все итерации параллельного алгоритма однотипны и число их велико (равно числу вершин в графе). Для отображения всех итераций понадобится достаточно много времени.

Поэтому в системе ПараЛаб реализована возможность отображать не все итерации, а лишь некоторые из них.

Правила использования системы ПараЛаб

1. Изменение способа отображения пересылки данных. Для задания способа отображения коммуникации процессоров выполните команду **Пересылка данных** пункта меню **Графика**. В появившемся списке выделите название желаемого способа отображения.

Рис. 12.12. Выбор способа отображения пересылки данных

2. Выбор темпа демонстрации. Для выбора темпа демонстрации необходимо выполнить команду **Темп демонстрации** пункта меню **Графика**. В появившемся диалоговом окне (рис. 12.13) предоставляется возможность выбора величины задержки между итерациями алгоритма и скорости движения пакетов (времени цветового выделения канала) при отображении коммуникации процессоров.

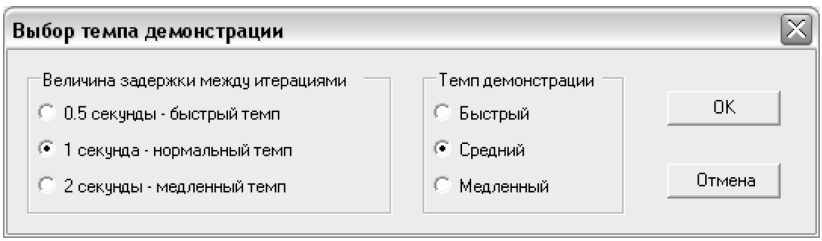


Рис. 12.13. Диалоговое окно выбора темпа демонстрации

Нажмите **OK** (**Enter**) для подтверждения выбора темпа демонстрации. Для возврата в основное меню системы ПараЛаб без сохранения изменений нажмите **Отмена** (**Escape**).

3. Настройка цветовой палитры. Для изменения цветов, которые используются в системе ПараЛаб для визуализации процесса решения задач, выполните команду **Настройка цвета** пункта меню **Графика**. В появившемся диалоговом окне (рис. 12.14) вы увидите прямоугольник

с линейным перетеканием более светлого цвета в более темный и квадрат, залитый цветом выделения. При выполнении алгоритмов сортировки более светлый (левый) цвет используется для отображения минимальных элементов сортируемого массива, а более темный – для отображения максимальных элементов. При выполнении алгоритмов на графах более светлый цвет используется для изображения дуг графа, имеющих минимальный вес, а темный – для дуг максимального веса. Цвет выделения используется при выполнении алгоритмов умножения матриц для отображения в области «Выполнение эксперимента» блоков матриц, расположенных на процессорах, и в алгоритмах на графах для отображения минимального охватывающего дерева (*алгоритм Прима*) и дерева кратчайших путей (*алгоритм Дейкстры*).

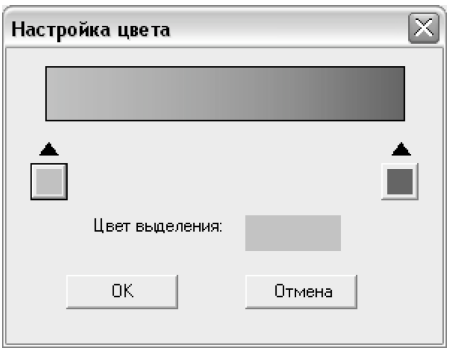


Рис. 12.14. Диалоговое окно настройки цветовой палитры

Чтобы изменить эти цвета, щелкните левой клавишей мыши на кнопке, расположенной слева или справа под шкалой перетекания. В результате появится стандартное диалоговое окно выбора цвета операционной системы Windows. В этом окне выберите цвет и нажмите кнопку **OK**. Цвет будет изменен. Для того чтобы изменить цвет выделения, щелкните левой клавишей мыши на отображающем этот цвет квадрате. При помощи стандартного диалогового окна задайте необходимый цвет.

Для изменения цветовой палитры нажмите кнопку **OK** (**Enter**) в диалоговом окне «Настройка цвета». Для возврата в основной режим работы системы ПараЛаб нажмите **Отмена** (**Escape**).

12.5.2. Область «Текущее состояние массива»

Эта область расположена в правой верхней части окна и отображает последовательность элементов сортируемого массива. Каждый элемент,

как и в области «Выполнение эксперимента», отображается вертикальной линией. Высота и интенсивность цвета линии дают представление о величине элемента: чем выше и темнее линия, тем больше элемент.

Все параллельные алгоритмы сортировки используют идею разделения исходного массива между процессорами. Блоки, выстроенные один за другим в порядке возрастания номеров процессоров, на которых они располагаются, образуют результирующий массив. После выполнения сортировки блоки массива на каждом процессоре должны быть отсортированы и, кроме того, элементы, находящиеся на процессоре с меньшим номером, не должны превосходить элементы, находящиеся на процессоре с большим номером.

Изначально массив – случайный набор элементов. После выполнения сортировки массив (при достаточно большом объеме исходных данных) изображается в виде прямоугольного треугольника с плавным перетеканием цвета из голубого в темно-синий.

12.5.3. Область «Результат умножения матрицы на вектор»

Эта область находится в правой верхней части окна и отображает состояние вектора-результата в процессе выполнения параллельного алгоритма умножения матрицы на вектор.

При выполнении алгоритма, основанного на ленточном горизонтальном разбиении матрицы, каждый процессор вычисляет один блок результирующего вектора путем умножения полосы матрицы A на вектор-аргумент b. Вычисленный на активном процессоре (подсвечен синим цветом) блок изображается темно-синим цветом. После выполнения коммуникации на каждом процессоре располагается весь результирующий вектор. Таким образом, все блоки вектора становятся темно-синими.

При выполнении алгоритма, основанного на ленточном вертикальном разбиении матрицы, каждый процессор вычисляет вектор частичных результатов путем умножения полосы матрицы на блок вектора-аргумента b; все блоки результирующего вектора в области «Результат умножения матрицы на вектор» подсвечиваются светло-синим цветом. После выполнения коммуникационного шага на каждом процессоре располагается блок результирующего вектора, блок активного процессора отображается в области «Результат умножения матрицы на вектор» темно-синим цветом.

При выполнении алгоритма, основанного на блочном разбиении матрицы, вектор b распределен между процессорами, составляющими столбцы процессорной решетки. После умножения блока матрицы A на блок вектора b процессор вычисляет блок вектора частичных результатов – он подсвечивается светло-синим цветом. После обмена блоками в рамках одной строки процессорной решетки каждый процессор этой строки содержит блок

результирующего вектора, блок активного процессора отображается в области «Результат умножения матрицы на вектор» темно-синим цветом.

12.5.4. Область «Результат умножения матриц»

Эта область находится в правой верхней части окна и отображает состояние матрицы-результата в процессе выполнения параллельного алгоритма матричного умножения.

Матрица *C* представляется разбитой на квадратные блоки. Каждый процессор многопроцессорной вычислительной системы отвечает за вычисление одного (алгоритмы Фокса и Кэннона) или нескольких (ленточный алгоритм) блоков результирующей матрицы *C*.

При выполнении ленточного алгоритма умножения темно-синим цветом закрашиваются те блоки, которые уже вычислены к данному моменту.

Если же выполняется алгоритм Фокса или алгоритм Кэннона, то все блоки матрицы *C* вычисляются одновременно, ни один из блоков не может быть вычислен раньше, чем будут выполнены все итерации алгоритма. Поэтому в области «Результат умножения матриц» отображается динамика вычисления того блока результирующей матрицы, который расположен на активном процессоре (этот процессор в области «Выполнение эксперимента» выделен синим цветом). Вычисленные к этому моменту слагаемые написаны темно-синим цветом, вычисляемое на данной итерации – цветом выделения.

Рис. 12.15. Область «Результат умножения матриц» при выполнении алгоритма Фокса

12.5.5. Область «Результат решения системы уравнений»

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает текущее состояние матрицы линейной системы уравнений в ходе выполнения алгоритма Гаусса. Темно-синим

цветом изображаются ненулевые элементы, а голубым – нулевые. После выполнения прямого хода алгоритма Гаусса ниже главной диагонали расположены только нулевые элементы. После выполнения обратного хода все ненулевые элементы расположены на главной диагонали.

12.5.6. Область «Результат обработки графа»

Эта область расположена в правой верхней части окна вычислительного эксперимента и отображает текущее состояние графа. Вершины графа имеют такое же взаимное расположение, как и в режиме редактирования графа. Дуги графа изображаются разными цветами: чем темнее цвет, тем больший вес имеет дуга.

В процессе выполнения алгоритмов на графах цветом выделения помечаются вершины и ребра, включенные к данному моменту в состав минимального охватывающего дерева (алгоритм Прима) или в дерево кратчайших путей (алгоритм Дейкстры).

12.5.7. Выбор процессора

Для более детального наблюдения за процессом выполнения эксперимента в системе ПараЛаб предусмотрена возможность отображения вычислений одного из процессоров системы в отдельном окне (рис. 12.16).

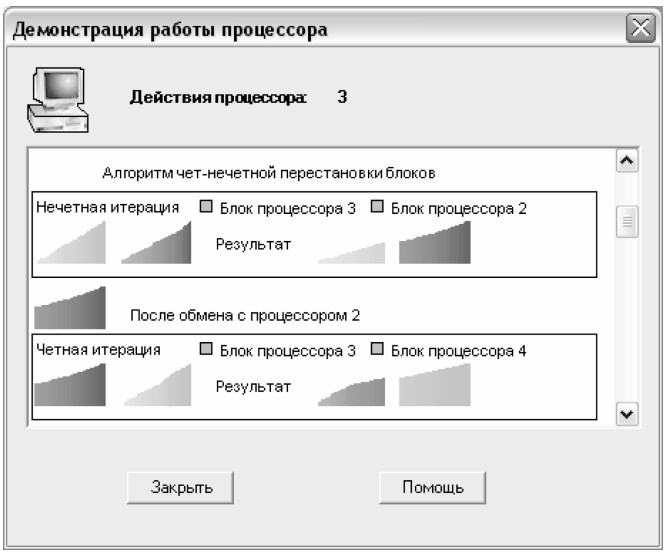


Рис. 12.16. Вид окна демонстрации работы процессора

Чтобы выбрать процессор для более детального наблюдения за теми вычислениями, которые он выполняет, необходимо в рабочей области навести на него указатель мыши (при этом его форма изменится и появится подсказка с номером выбранного процессора) и дважды щелкнуть левой клавишей.

Далее в появившемся окне «Демонстрация работы процессора» будет детально поясняться ход выполняемых на процессоре вычислений.

12.6. Накопление и анализ результатов экспериментов

Выполнение численных экспериментов для изучения различных параллельных алгоритмов решения сложных вычислительных задач во многих случаях может потребовать проведения длительных вычислений. Для обоснования выдвигаемых предположений необходимо выполнить достаточно широкий набор экспериментов. Эти эксперименты могут быть выполнены на различных многопроцессорных вычислительных системах, разными методами, для различных исходных данных.

Накопление итогов экспериментов производится системой ПараЛаб автоматически. О каждом проведенном эксперименте хранится исчерпывающая информация: дата и время проведения, детальное описание вычислительной системы и решаемой задачи, метод вычислений, время, потребовавшееся для выполнения эксперимента. Следует отметить, что повторение эксперимента с идентичными исходными установками приведет к повторному учету результатов.

При просмотре результатов предоставляется возможность восстановления эксперимента по сохраненной записи. Можно выполнять операции удаления отдельной записи и очистки всего списка результатов.

При сохранении текущего эксперимента в файле происходит сохранение всех записанных результатов.

12.6.1. Просмотр результатов

Для демонстрации результатов экспериментов в системе ПараЛаб существует окно, содержащее **Таблицу итогов** и **Лист графиков**.

Каждая строка таблицы итогов (см. рис. 12.17) представляет один выполненный эксперимент. По умолчанию, в таблице итогов выделена первая строка и по ней построен график зависимости времени выполнения эксперимента от объема исходных данных на листе графиков. Для того чтобы изменить вид отображаемой зависимости, нужно выбрать соответствующие пункты в списках, расположенных в левом верхнем и нижнем правом углу листа графиков. Можно построить зависимости времени

выполнения эксперимента и ускорения от объема исходных данных, количества процессоров, производительности процессора и характеристик сети. Выделяя различные строки таблицы, вы можете просматривать графики, соответствующие различным экспериментам.

Следует отметить, что при построении графиков для экспериментов, проведенных в режиме имитации, используются необходимые аналитические зависимости (см. лекции 6 – 10). Для экспериментов, проведенных на вычислительном кластере, применяется набор полученных к данному моменту результатов реальных экспериментов.

При выделении нескольких строк в таблице результатов на листе графиков отображается несколько зависимостей.

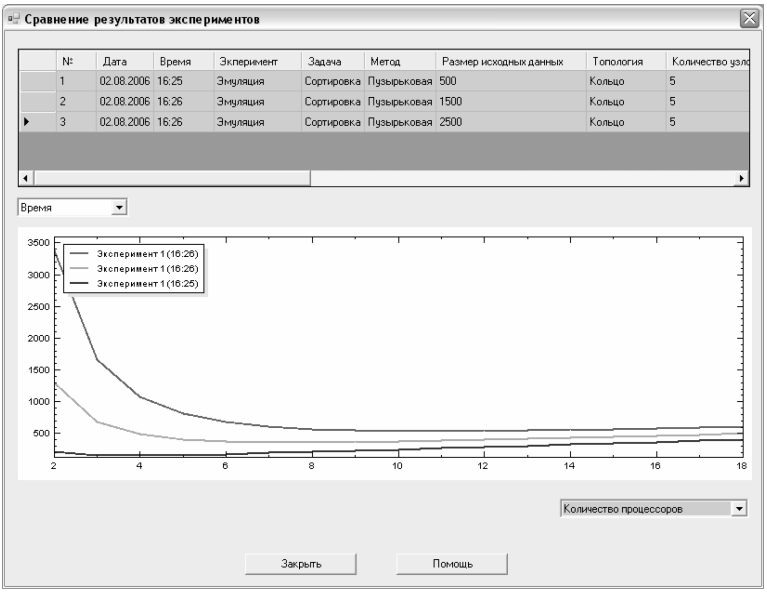


Рис. 12.17. Анализ результатов экспериментов

Правила использования системы ПараЛаб

1. Общие результаты. Для демонстрации накопленных результатов экспериментов следует выбрать пункт меню **Результаты**, выделить команду **Показать** и выполнить одну из двух команд: **Из активного окна** или **Из всех окон**. При выполнении первой команды будут отображены результаты, накопленные в активном окне вычислительного эксперимента. При выполнении второй команды – результаты из всех открытых окон экспериментов. Вид окна с результатами экспериментов представлен на рис. 12.17. Это окно содержит таблицу результатов и лист графиков.

2. Выделение строки в таблице результатов. Каждая строка таблицы представляет один выполненный эксперимент. Для выделения строки наведите указатель мыши на нужную строчку и нажмите левую кнопку мыши. Также можно воспользоваться курсорными стрелками вверх и вниз (выделенной строкой станет соответственно предыдущая или следующая строка). Если выделенная строчка одна, то на листе графиков отображается только зависимость, соответствующая выделенной строке.

3. Выделение нескольких строк в таблице результатов. Чтобы выделить несколько подряд идущих строк в таблице итогов, нажмите **Shift** и выделите мышью первую и последнюю строчку желаемого диапазона. Для выделения нескольких строк, не образующих непрерывную последовательность, нажмите **Ctrl** и выделяйте строки в произвольном порядке. Для того чтобы выделить несколько строк при помощи курсорных клавиш, нажмите **Shift** и перемещайтесь по таблице при помощи клавиш вверх и вниз. При выделении нескольких строк в таблице результатов на листе графиков отображается несколько зависимостей.

4. Восстановление эксперимента по записи в таблице итогов. Как уже отмечалось выше, запись в таблице итогов содержит исчерпывающую информацию о вычислительном эксперименте. Для восстановления эксперимента по записи необходимо выделить эту запись одним из перечисленных способов, щелкнуть правой кнопкой мыши в области списка итогов и выполнить команду **Восстановить эксперимент** появившегося контекстного меню. Эксперимент будет восстановлен в активном окне.

5. Удаление записи. Для удаления выделенной записи выполните команду **Удалить** контекстного меню списка итогов.

6. Изменение вида зависимости на листе графиков. Для того чтобы изменить вид зависимости, изображенной на листе графиков, выберите нужные значения в списках, расположенных слева вверху и справа внизу от листа графиков. Нижний правый список позволяет выбрать аргумент зависимости, а левый верхний – вид зависимости.

Задания и упражнения

Выполните несколько экспериментов с одним и тем же методом умножения матриц, изменяя объем исходных данных и количество процессоров. Используя окно итогов экспериментов, проанализируйте полученные результаты. Постройте одновременно несколько графиков на листе графиков и сравните их.

12.7. Выполнение вычислительных экспериментов

В рамках системы ПараЛаб допускаются разные схемы организации вычислений при проведении экспериментов по изучению и исследова-

нию параллельных алгоритмов решения сложных вычислительных задач. Решение задач может происходить в режиме последовательного исполнения или в режиме разделения времени с возможностью одновременного наблюдения итераций алгоритмов во всех окнах вычислительных экспериментов. Проведение серийных экспериментов, требующих длительных вычислений, может происходить в автоматическом режиме с возможностью запоминания результатов решения для организации последующего анализа полученных данных. Выполнение экспериментов может осуществляться и в пошаговом режиме.

12.7.1. Последовательное выполнение экспериментов

В общем случае цель проведения вычислительных экспериментов состоит в оценке эффективности параллельного метода при решении сложных вычислительных задач в зависимости от параметров многопроцессорной вычислительной системы и (или) от объема исходных данных. Выполнение таких экспериментов может сводиться к многократному повторению этапов постановки и решения задач. При решении задач в рамках системы ПараЛаб процесс может быть приостановлен в любой момент времени (например, для смены графических форм наблюдения за процессом решения) и продолжен далее до получения результата. Результаты решения вычислительных задач записываются в базу результатов экспериментов и представляются далее в виде, удобном для проведения анализа.

Правила использования системы ПараЛаб

1. Проведение вычислительного эксперимента. Для выполнения вычислительного эксперимента выберите пункт меню **Выполнение** и выполните команду **В активном окне**. Решение задачи осуществляется без останова до получения результата. В ходе выполнения эксперимента основное меню системы заменяется на меню с командой **Остановить**; после завершения решения задачи основное меню системы восстанавливается.

2. Приостановка решения. Для приостановки процесса выполнения эксперимента следует выбрать в строке меню команду **Остановить** (команда доступна только до момента завершения решения).

3. Продолжение решения. Для продолжения ранее приостановленного процесса выполнения эксперимента следует выбрать команду **Продолжить** пункта меню **Выполнение** (команда может быть выполнена только в случае, если после приостановки процесса поиска не изменялись постановка задачи и параметры вычислительной системы; при невозможности продолжения ранее приостановленного процесса выполнения эксперимента имя данной команды высвечивается серым цветом).

Задания и упражнения

1. В активном окне вычислительного эксперимента установите топологию **Кольцо** и число процессоров, равное десяти. Сделайте текущей задачей задачу сортировки с использованием пузырькового алгоритма.

2. Выполните первые две итерации алгоритма и приостановите процесс вычислений.

3. Измените темп демонстрации и способ отображения пересылки данных.

4. Продолжите выполнение эксперимента до получения результата.

12.7.2. Выполнение экспериментов по шагам

Для более детального анализа итераций параллельного алгоритма в системе ПараЛаб предусмотрена возможность пошагового выполнения вычислительных экспериментов. В данном режиме после выполнения каждой итерации происходит приостановка параллельного алгоритма. Это дает пользователю системы возможность подробнее изучить результаты проведенной итерации.

Правила использования системы ПараЛаб

1. Пошаговый режим. Для задания режима приостановки вычислительного эксперимента после выполнения каждой итерации следует выбрать команду **Пошаговый режим** пункта меню **Выполнение**. После выполнения этой команды основное меню системы ПараЛаб заменяется на меню пошагового выполнения эксперимента с командами:

- команда **Шаг** — выполнить очередную итерацию поиска;
- команда **Без остановки** — продолжить выполнение эксперимента без остановки;
- команда **Закрыть** — приостановить эксперимент и вернуться к выполнению команд основного меню.

12.7.3. Выполнение нескольких экспериментов

Последовательное выполнение экспериментов затрудняет сравнение результатов итераций параллельных алгоритмов. Для удобства более детального сравнения таких данных система ПараЛаб позволяет демонстрировать на экране дисплея одновременно результаты всех сравниваемых экспериментов. Для этого экран дисплея может разделяться на несколько прямоугольных областей (*окон экспериментов*), в каждой из которых могут высвечиваться результаты отдельно проводимого эксперимента. В любой момент пользователь системы ПараЛаб может создать новое окно для выполнения нового эксперимента. При этом итоги экс-

периментов формируются раздельно для каждого имеющегося окна. При визуализации окна экспериментов могут разделять экран (в этом случае содержимое всех окон является видимым) или могут перекрываться. Пользователь может сделать любое окно активным для выполнения очередного эксперимента. Но вычисления могут быть выполнены и во всех окнах одновременно в режиме разделения времени, когда каждая новая итерация выполняется последовательно во всех имеющихся окнах. Используя этот режим, исследователь может наблюдать за динамикой нескольких экспериментов, результаты вычислений могут быть визуально различимы, и их сравнение может быть выполнено на простой наглядной основе.

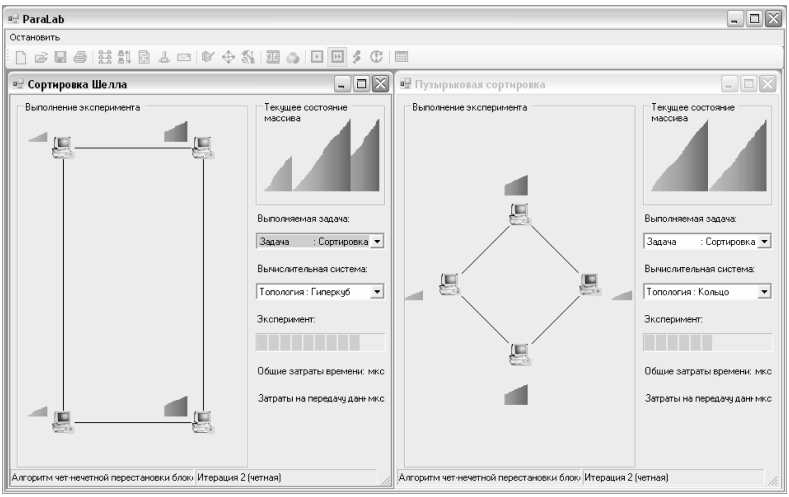


Рис. 12.18. Пример демонстрации нескольких окон экспериментов

Следует отметить, что итоги экспериментов, проведенных в разных окнах, могут высвечиваться совместно в одной и той же таблице итогов (см. п. 12.6.1).

Правила использования системы ПараЛаб

1. Создание окна. Для создания окна для проведения экспериментов следует выполнить команду **Создать новый** пункта меню **Эксперимент**. Закрытие окна эксперимента производится принятыми в операционной системе Windows способами (например, нажатием кнопки закрытия окна в правом верхнем углу окна). Для одновременного закрытия всех имеющихся окон следует выполнить команду **Закрыть все** пункта меню **Эксперимент**.

2. Управление окнами. Управление размерами окон экспериментов осуществляется принятыми в системе Windows способами (максимизация, минимизация, изменение размеров при помощи мыши). Для одновременного показа всех имеющихся окон без перекрытия можно использовать команду **Показать все** пункта меню **Эксперимент**; для выделения большей части экрана для активного окна (но при сохранении возможности быстрого доступа ко всем имеющимся окнам) следует применить команду **Расположить каскадом** пункта меню **Эксперимент**.

3. Проведение экспериментов во всех окнах. Для выполнения вычислительных экспериментов во всех имеющихся окнах в режиме разделения времени (т.е. при переходе к выполнению следующей итерации только после завершения текущей во всех имеющихся окнах) следует применить команду **Во всех окнах** пункта меню **Выполнение**. Управление процессом вычислений осуществляется так же, как и при использовании единственного окна (приостановка выполнения алгоритмов по команде **Остановить**, продолжение вычислений по команде **Продолжить** пункта меню **Выполнение**).

4. Сравнение итогов экспериментов. Для того чтобы свести в одну таблицу итогов результаты, полученные во всех окнах экспериментов, выполните последовательность команд **Результаты→Показать→Из всех окон**.

Задания и упражнения

1. Откройте второе окно вычислительного эксперимента, установите режим показа окон без перекрытия.

2. В первом окне выберите метод пузырьковой сортировки, установите топологию **Гиперкуб**. Во втором окне установите топологию **Гиперкуб** и выберите метод сортировки Шелла.

3. Выполните вычислительные эксперименты одновременно в обоих окнах; отрегулируйте скорость демонстрации установкой подходящего темпа показа.

4. Получите сводную таблицу итогов экспериментов. Сравните временные характеристики алгоритмов пузырьковой сортировки и сортировки Шелла.

12.7.4. Выполнение серии экспериментов

ПараЛаб обеспечивает возможность автоматического (без участия пользователя) выполнения серий экспериментов, требующих проведения длительных вычислений. При задании этого режима работы системы пользователь должен выбрать окно, в котором будут выполняться эксперименты, установить количество экспериментов и выбрать тот параметр, который будет изменяться от эксперимента к эксперименту (объем исходных данных или количество процессоров). Результаты экспериментов

можно запомнить в списке итогов и журнале экспериментов, а в последующем проанализировать.

Правила использования системы ПараЛаб

1. Выполнить серию. Переход в режим выполнения последовательности экспериментов осуществляется при помощи команды **Выполнить серию** пункта меню **Выполнение**. При выполнении команды может быть задано число экспериментов в серии, а также выбран тип серии: исследуется ли зависимость времени и ускорения решения поставленной задачи от объема исходных данных или от количества используемых процессоров.

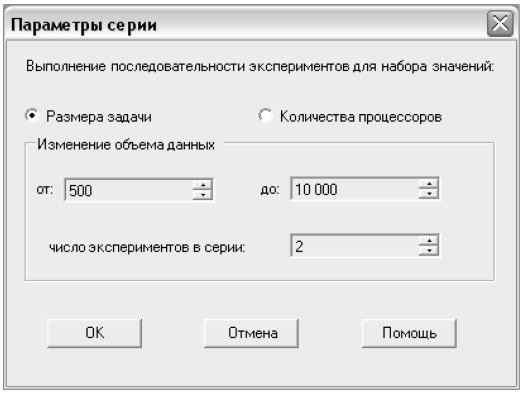


Рис. 12.19. Диалоговое окно задания параметров серии

При выполнении серии экспериментов основное меню системы ПараЛаб заменяется на меню управления данным режимом вычислений, команды которого позволяют:

- команда **Пуск** — выполнить последовательность экспериментов;
- команда **Закрыть** — приостановить выполнение данного режима и вернуться к выполнению команд основного меню;
- команда **Справка** — получить дополнительную справочную информацию.

При решении серии поставленных задач (после выполнения команды **Пуск**) эксперимент может быть приостановлен в любой момент времени при помощи команды **Остановить**.

12.7.5. Выполнение реальных вычислительных экспериментов

Помимо выполнения экспериментов в режиме имитации, в системе ПараЛаб предусмотрена возможность проведения реальных эксперимен-

тов в режиме удаленного доступа к вычислительному кластеру (настройка возможности удаленного доступа описывается в инструкции комплекта поставки системы ПараЛаб). При выборе этого режима выполнения эксперимента необходимо поставить задачу и выбрать нужное количество процессоров для ее решения. После выполнения имитационных и реальных экспериментов пользователь ПараЛаб может сравнить результаты и оценить точность используемых в системе теоретических моделей времени выполнения параллельных алгоритмов. Результаты реальных экспериментов автоматически заносятся в таблицу итогов.

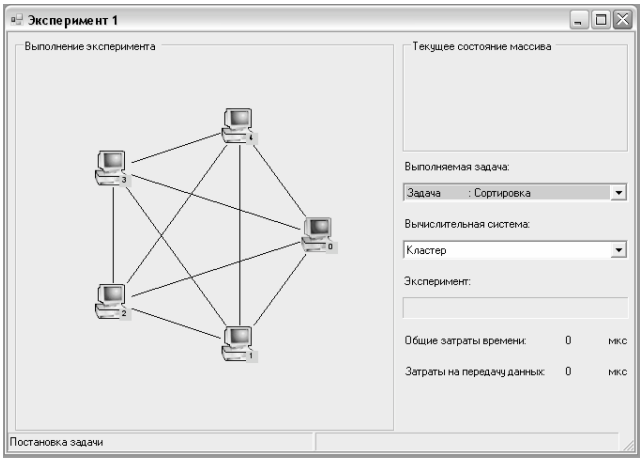


Рис. 12.20. Окно для выполнения реального вычислительного эксперимента

При выполнении реального эксперимента ход вычислений и обмен данными между процессорами не отображаются. В списке параметров вычислительной системы присутствует только строка, указывающая на то, что выполняется эксперимент в режиме удаленного доступа к кластеру, и указывается число процессоров. Режим пошагового выполнения эксперимента недоступен.

Правила использования системы ПараЛаб

1. Переход в режим реального выполнения эксперимента. Для перехода в режим выполнения реальных вычислительных экспериментов в режиме удаленного доступа к вычислительному кластеру выберите пункт меню **Система** и выделите мышью команду **Кластер**. Подтверждением того, что данный режим активен, является значок ✔ слева от надписи. После выбора этого режима топология вычислительной системы автомати-

чески заменяется на топологию **Полный граф**, так как последняя соответствует топологии кластера. Постановка задачи осуществляется так же, как и при выполнении экспериментов в режиме имитации.

2. Проведение реального эксперимента. Для проведения реального вычислительного эксперимента выполните команду **В активном окне** пункта меню **Выполнение**.

12.8. Использование результатов экспериментов

12.8.1. Запоминание результатов

В любой момент результаты выполненных в активном окне вычислительных экспериментов могут быть сохранены в архиве системы ПараЛаб. Данные, сохраняемые для окна проведения эксперимента, включают:

- параметры активной вычислительной системы (топология, количество процессоров, производительность процессора, время начальной подготовки данных, пропускная способность сети, метод передачи данных);
- постановку задачи (тип задачи, размер исходных данных, метод решения);
- таблицу результатов, ранее полученных в этом окне.

Данные, сохраненные в архиве системы, в любой момент могут быть восстановлены из архива, и, тем самым, пользователь может продолжать выполнение своих экспериментов в течение нескольких сеансов работы с системой ПараЛаб.

Кроме того, в рамках системы ПараЛаб исследователю предоставляется возможность сохранения в архиве и чтения из архива сформированных графов (см. п. 12.4.5).

Правила использования системы ПараЛаб

1. Запись данных. Для сохранения результатов выполненных экспериментов следует выполнить команду **Сохранить** пункта меню **Архив**. При выполнении записи в диалоговом окне **Сохранить файл как** следует задать имя файла, в котором будут сохранены данные. Расширение имени файла может не указываться. Файлы с параметрами вычислительных экспериментов имеют расширение **.prl**.

2. Чтение данных. Для чтения параметров экспериментов, записанных ранее в архив системы ПараЛаб, следует выбрать пункт меню **Архив** и указать команду **Загрузить**. После выполнения этой команды в активное окно будут загружены параметры вычислительного эксперимента и таблица результатов, сохраненные в выбранном файле.

Задания и упражнения

Выполните вычислительные эксперименты, план проведения которых состоит в следующем:

1. Выполните какой-либо эксперимент и сохраните параметры выполненного эксперимента в архиве системы.

2. Завершите выполнение системы.

3. Выполните повторный запуск системы и загрузите запомненные параметры эксперимента из архива.

12.9. Краткий обзор лекции

В лекции описывается программная система Параллельная Лаборатория, которая обеспечивает возможность проведения вычислительных экспериментов с целью изучения и исследования параллельных алгоритмов решения сложных вычислительных задач. Система может быть использована для организации лабораторного практикума по различным учебным курсам в области параллельного программирования, в рамках которого обеспечивается возможность:

- моделирования многопроцессорных вычислительных систем с различной топологией сети передачи данных;
- получения визуального представления о вычислительных процессах и операциях передачи данных, происходящих при параллельном решении разных вычислительных задач;
- построения оценок эффективности изучаемых методов параллельных вычислений.

В лекции описывается методика работы с системой ПараЛаб. Подробно рассмотрены действия, которые необходимо выполнить для формирования модели вычислительной системы, постановки вычислительной задачи, получения визуального представления о вычислительных и коммуникационных процессах, которые необходимы для решения задачи. В лекции описываются основные алгоритмы, реализованные в системе ПараЛаб. Приведены примеры анализа результатов экспериментов на основе той информации, которая сохраняется в системе.

В целом система ПараЛаб представляет собой интегрированную среду для изучения и исследования параллельных алгоритмов решения сложных вычислительных задач. Широкий набор имеющихся средств визуализации процесса выполнения эксперимента и анализа полученных результатов позволяет изучить эффективность использования тех или иных алгоритмов на разных вычислительных системах, сделать выводы о масштабируемости алгоритмов и определить возможное ускорение процесса параллельных вычислений.

Реализуемые системой ПараЛаб процессы изучения и исследований ориентированы на активное усвоение основных теоретических положений и способствуют формированию у пользователей собственных представлений о моделях и методах параллельных вычислений путем наблюдения, сравнения и сопоставления широкого набора различных визуальных графических форм, демонстрируемых в ходе выполнения вычислительного эксперимента.

12.10. Обзор литературы

Дополнительная информация по моделированию и анализу параллельных вычислений может быть получена, например, в [2, 22], полезная информация содержится также в [51, 63].

Подробное рассмотрение параллельных алгоритмов, реализованных в системе ПараЛаб, выполнено в [26, 51, 63], а также в [3].

Впервые модель Хокни параллельных вычислений была изложена в работе [46].

Систематическое изложение (на момент издания работы) вопросов моделирования и анализа параллельных вычислений приводится в [77].

Литература

Основная литература

1. *Богачев К.Ю.* Основы параллельного программирования. М.: БИНОМ. Лаборатория знаний, 2003.
2. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
3. *Гергель В.П., Стронгин Р.Г.* Основы параллельных вычислений для многопроцессорных вычислительных систем. Н. Новгород: Изд-во ННГУ, 2001.
4. *Немнюгин С., Стесик О.* Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.
5. *Andrews G.R.* Foundations of Multithreading, Parallel and Distributed Programming. Addison-Wesley, 2000 (русский перевод Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: Издательский дом «Вильямс», 2003).

Дополнительная литература

6. *Бахвалов Н.С., Жидков Н.П., Кобельков Г.М.* Численные методы. М.: Наука, 1987.
7. *Воеводин В.В.* Модели и методы в параллельных процессах. М.: Наука, 1986.
8. *Воеводин В.В.* Математические основы параллельных вычислений. М.: МГУ, 1991.
9. *Гергель В.П., Стронгин Л.Г., Стронгин Р.Г.* Метод окрестностей в задачах распознавания // Изв. АН СССР. Техническая кибернетика. 1987. №4. С.14-22.
10. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНТО, 1999.
11. *Корнеев В.В.* Параллельные вычислительные системы. М.: Нолидж, 1999.
12. *Корнеев В.В.* Параллельное программирование в MPI. М. – Ижевск: Институт компьютерных исследований, 2003.
13. *Самарский А.А., Гулин А.В.* Численные методы. М.: Наука, 1989.
14. *Таненбаум Э.* Архитектура компьютера. СПб.: Питер, 2002.
15. *Тихонов А.Н., Самарский А.А.* Уравнения математической физики. М.: Наука, 1977.
16. *Хамахер К., Вранешич З., Заки С.* Организация ЭВМ. СПб.: Питер, 2003.
17. *Akl S.G.* Parallel Sorting Algorithms. – Orlando, FL: Academic Press, 1985.

18. *Amdahl G.* Validity of the single processor approach to achieving large scale computing capabilities // AFIPS Conference Proceedings, 1967. Vol. 30. P. 483 – 485, Washington, D.C.: Thompson Books.
19. *Barker M.* (Ed.) Cluster Computing Whitepaper at http://www.dcs.port.ac.uk/ ~mab/tfcc/WhitePaper/. 2000.
20. *Barnard S.* PMRSB: Parallel multilevel recursive spectral bisection//Proc. Supercomputing '95. 1995.
21. *Berger M., Bokhari S.* Partitioning strategy for nonuniform problems on multiprocessors // IEEE Transactions on Computers. 1987. C-36(5). P. 570–580.
22. *Bertsekas D.P., Tsitsiklis J.N.* Parallel and Distributed Computation. Numerical Methods. — Prentice Hall, Englewood Cliffs, New Jersey, 1989.
23. *Blackford L.S., Choi J., Cleary A., D'Azevedo E., Demmel J., Dhillon I., Dongarra J.J., Hammarling S., Henry G., Petitet A., Stanley D. Walker R.C. Whaley K.* Scalapack Users' Guide (Software, Environments, Tools). Soc. for Industrial & Applied Math., 1997.
24. *Buyya R.* (Ed.) High Performance Cluster Computing. Volume 1: Architectures and Systems. Volume 2: Programming and Applications. — Prentice Hall PTR, Prentice-Hall Inc., 1999.
25. *Clark D.* Breaking the Terraflops Barrier // Computer. 1997. V. 30. N 2. P. 12–14.
26. *Cormen T.H., Leiserson C.E. , Rivest R.L., Stein C.* Introduction to Algorithms. 2nd Edition. — The MIT Press, 2001.
27. *Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., and Melon R.* Parallel Programming in OpenMP. Morgan Kaufmann Publishers, 2000.
28. *Culler D., Singh J.P., Gupta A.* Parallel Computer Architecture: A Hardware/Software Approach. — Morgan Kaufmann, 1998.
29. *Dally W.J., Towles B.P.* Principles and Practices of Interconnection Networks. — Morgan Kaufmann, 2003.
30. *Dongarra J.J., Duff L.S., Sorensen D.C., Vorst H.A.V.* Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools). Soc. for Industrial & Applied Math., 1999.
31. *Flynn M.J.* Very high-speed computing systems // Proceedings of the IEEE 1966. 54(12): P. 1901–1909.
32. *Foster I.* Designing and Building Parallel Programs: Concepts and Tools for Software Engineering. Reading, MA: Addison-Wesley, 1995.
33. *Fox G.C.* et al. Solving Problems on Concurrent Processors. — Prentice Hall, Englewood Cliffs, NJ, 1988.
34. *Fox G.C., Otto S.W. and Hey A.J.G.* Matrix Algorithms on a Hypercube I: Matrix Multiplication. Parallel Computing. 1987. 4 H. 17-31.

35. *Geist G.A., Beguelin A., Dongarra J., Jiang W., Manchek B., Sunderam V.* PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Network Parallel Computing. MIT Press, 1994.
36. *George A., Liu J.* Computer Solution of Large Sparse Positive Definite Systems. — Prentice-Hall, Englewood Cliffs NJ, 1981.
37. *Gilbert J., Miller G., Teng S.* Geometric mesh partitioning: Implementation and experiments // Proceedings of International Parallel Processing Symposium. 1995.
38. *Gilbert J., Zmijewski E.* A parallel graph partitioning algorithm for a message-passing multiprocessor // International Journal of Parallel Programming. 1987. P. 498 – 513.
39. *Grama A.Y., Gupta A. and Kumar V.* Isoefficiency: Measuring the scalability of parallel algorithms and architectures // IEEE Parallel and Distributed technology. 1993. 1 (3). P. 12 – 21.
40. *Group W., Lusk E., Skjellum A.* Using MPI. Portable Parallel Programming with the Message-Passing Interface. – MIT Press, 1994.
41. *Group W., Lusk E., Skjellum A.* Using MPI — 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). — MIT Press, 1999.
42. *Group W., Lusk E., Thakur R.* Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). — MIT Press, 1999.
43. *Gustavson J.L.* Reevaluating Amdahl's law // Communications of the ACM. 1988. 31 (5). P. 532 – 533.
44. *Heath M., Raghavan P. A* Cartesian parallel nested dissection algorithm. SIAM Journal of Matrix Analysis and Applications, 1995. 16(1). P. 235-253.
45. *Hockney R.W., Jesshope C.R.* Parallel Computers 2. Architecture, Programming and Algorithms. — Adam Hilger, Bristol and Philadelphia, 1988 (русский перевод 1-го издания: Хокни Р., Джессхоуп К. Параллельные ЭВМ. Архитектура, программирование и алгоритмы. М.: Радио и связь, 1986).
46. *Hockney R.* The communication challenge for MPP: Intel Paragon and Meiko CS-2 // Parallel Computing. 1994. 20 (3). P. 389 – 398.
47. *Kahaner D., Moler C., Nash S.* Numerical Methods and Software. – Prentice Hall, 1988.
48. *Karypis G., Kumar V.* A parallel algorithm for multilevel graph partitioning and sparse matrix ordering // Journal of Parallel and Distributed Computing. 1988. 48(1).
49. *Karypis G., Kumar V.* Parallel multilevel k-way partitioning scheme for irregular graphs // Siam Review, 1999. 41(2). P. 278 – 300.
50. *Knuth D.E.* The Art of Computer Programming. Volume 3: Sorting and Searching. Second edition. — Reading, MA: Addison-Wesley, 1997 (рус-

ский перевод: Кнут Д. Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. М.: Мир, 1981).
51. *Kumar V., Grama A., Gupta A., Karypis G.* Introduction to Parallel Computing. — The Benjamin/Cummings Publishing Company, Inc.,1994 (2nd edn., 2003).
52. *Kung H.T.* Why Systolic Architecture? // Computer. 1982. 15. № 1. P. 37–46.
53. *Miller G., Teng S., Thurston W., Vavasis S.* Automatic mesh partitioning // A. George, John R. Gilbert, and J. Liu, editors. Sparse Matrix Computations: Graph Theory Issues and Algorithms. IMA Volumes in Mathematics and its applications. Springer-Verlag, 1993.
54. NSF. 1992. Grand Challenge: High-Performance Computing and Communications, Report, Committee on Physical, Mathematical and Engineering Sciences, D.C.: U.S. Office of Science and Technology Policy, National Science Foundation.
55. *Nour-Omid B., Raefsky A., Lyzenga G.* Solving finite element equations on concurrent computers // A. K. Noor, editor. American Soc. Mech. 1986. P. 291-307.
56. *Ou C., Ranka S., Fox G.* Fast and parallel mapping algorithms for irregular and adaptive problems // Journal of Supercomputing. 1996. 10. P. 119 – 140.
57. *Pacheco P.* Parallel Programming with MPI. — Morgan Kaufmann, 1996.
58. *Patra A., Kim D.* Efficient mesh partitioning for adaptive hp finite element methods // International Conference on Domain Decomposition Methods. 1998.
59. *Patterson D.A., Hennessy J.L.* Computer Architecture: A Quantitative Approach. 2d ed. — San Francisco: Morgan Kaufmann, 1996.
60. *Pfister G. P.* In Search of Clusters. — Prentice Hall PTR, Upper Saddle River, NJ, 1995 (2nd edn., 1998).
61. *Pilkington J., Baden S.* Partitioning with space filling curves. Technical Report CS94-349, Dept. of Computer Science and Engineering, Univ. of California. 1994.
62. *Pothen A.* Graph partitioning algorithms with applications to scientific computing // D. Keyes, A. Sameh, and V. Venkatakrishnan, editors, Parallel Numerical Algorithms. Kluwer Academic Press, 1996.
63. *Quinn M.J.* Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill, 2004.
64. *Raghavan P.* Line and plane separators. — Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61901. 1993.
65. *Raghavan P.* Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee. 1995.

66. *Roosta S.H.* Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag,NY, 2000.
67. *Schloegel K., Karypis G., Kumar V.* Graph Partitioning for High Performance Scientific Simulations. 2000.
68. *Skillicorn D.B., Talia D.* Models and languages for parallel computation // ACM Computing surveys, 1998. 30, 2.
69. *Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J.* MPI: The Complete Reference. — MIT Press, Boston, 1996.
70. *Sterling T.* (Ed.) Beowulf Cluster Computing with Windows. — Cambridge, MA: The MIT Press, 2001.
71. *Sterling T.* (Ed.) Beowulf Cluster Computing with Linux. — Cambridge, MA: The MIT Press, 2002.
72. *Strongin R.G., Sergeev Ya.D.* Global optimization with non-convex constraints: Sequential and parallel algorithms. Kluwer Academic Publisher, Dordrecht. 2000.
73. *Tanenbaum A.* Modern Operating System. 2nd edn. – Prentice Hall, 2001.
74. *Walshaw C., Cross M.* Parallel optimization algorithms for multilevel mesh partitioning. Techical Report 99/IM/44, Uviversity of Greenwich, London, UK. 1999.
75. *Wilkinson B., Allen M.* Parallel programming. — Prentice Hall, 1999.
76. *Xu Z., Hwang K.* Scalable Parallel Computing Technology, Architecture, Programming. – Boston: McGraw-Hill, 1998.
77. *Zomaya A.Y.* (Ed.) Parallel and Distributed Computing Handbook. — McGraw-Hill, 1996.

Учебно-методические пособия

78. *Афанасьев К.Е. и др.* Многопроцессорные вычислительные системы и параллельное программирование. Кемерово: Кузбассвузиздат, 2003.
79. *Головашкин Д.Л.* Методы параллельных вычислений. Ч. 1. Самара: Самар. гос. аэрокосм, ун-т, 2002.
80. *Головашкин Д., Головашкина С.П.* Методы параллельных вычислений. Ч. 2. Самара: Самар. гос. аэрокосм. ун-т, 2003.
81. *Деменев А.Г.* Параллельные вычислительные системы: основы программирования и компьютерного моделирования. Пермь: ПГПУ, 2001.
82. *Дацюк В.Н. и др.* Методическое пособие по курсу «Многопроцессорные системы и параллельное программирование». Ростов-на-Дону: РГУ, 2000.
83. *Дорошенко А.Е.* Математические модели и методы организаций высокопроизводительных вычислений. Киев: Наукова думка, 2000.
84. *Комолкин А.В., Немнюгин С.А.* Программирование для высокопроизводительных ЭВМ. СПб.: Изд-во НИИ химии СПбГУ, 1998.

85. *Сергеев Я.Д., Стронгин Р.Г., Гришагин В.А.* Введение в параллельную глобальную оптимизацию. Н. Новгород: ННГУ, 1998.
86. *Старченко А.В., Есаулов А.О.* Параллельные вычисления на многопроцессорных вычислительных системах. Томск: ТГУ, 2002.
87. *Шпаковский Г.И., Серикова Н.В.* Программирование для многопроцессорных систем в стандарте MPI: Пособие. Мн.: БГУ, 2002.
88. *Фурсов В.А. и др.* Введение в программирование для параллельных ЭВМ и кластеров. Самара: СНЦ РАН, СГАУ, 2000.
89. *Якобовский М.В.* Распределенные системы и сети. М.: МГТУ «Станкин», 2000.

Информационные ресурсы сети Интернет

90. Информационно-аналитические материалы по параллельным вычислениям (http://www.parallel.ru).
91. Информационные материалы Центра компьютерного моделирования Нижегородского университета (http://www.software.unn.ac.ru/ccam).
92. Информационные материалы рабочей группы IEEE по кластерным вычислениям (http://www.ieeetfcc.org).
93. Introduction to Parallel Computing (Teaching Course) (http://www.ece.nwu.edu/~choudhar/C58/).
94. Foster I. Designing and Building Parallel Programs. — Addison Wesley, 1994 (http://www.mcs.anl.gov/dbpp).

Учебное издание

Гергель Виктор Павлович
ТЕОРИЯ И ПРАКТИКА ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ
Учебное пособие

Литературный редактор *С. Перепелкина*
Корректор *Ю. Голомазова*
Компьютерная верстка *Н. Овчинникова*
Обложка *М. Автономова*

Подписано в печать 25.05.2007. Формат 60x90 $^1/_{16}$.
Гарнитура Таймс. Бумага офсетная. Печать офсетная.
Усл. печ. л. 26,5. Тираж 2000 экз. Заказ №

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
Москва, Электрический пер., 8, стр. 3
E-mail: admin@intuit.ru, http://www.intuit.ru

ООО «БИНОМ. Лаборатория знаний»
Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-1902, (495) 157-5272
E-mail: Lbz@aha.ru, http://www.Lbz.ru