В. Д. Корнеев

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В MPI

РОССИЙСКАЯ АКАДЕМИЯ НАУК
СИБИРСКОЕ ОТДЕЛЕНИЕ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ
И МАТЕМАТИЧЕСКОЙ ГЕОФИЗИКИ

В.Д. Корнеев

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В MPI

Ответственный редактор
доктор технических наук
В.Э. Малышкин

Второе, исправленное издание

НОВОСИБИРСК
2002

УДК 518.65
ББК 3 973.2–018
К 67

Корнеев В.Д. Параллельное программирование в MPI. – 2-е изд., испр. – Новосибирск: Изд-во ИВМиМГ СО РАН, 2002. – 215 с.

Книга посвящена параллельному программированию на базе системы с передачей сообщений MPI, которая является основным средством программирования таких современных высокопроизводительных мультикомпьютеров, как Silicon Graphics, Cray T3D, Cray T3E, IBM SP2 и многих других. Рассмотрены многочисленные примеры параллельного программирования алгоритмов решения различных стандартных задач: умножения матрицы на матрицу, задачи Дирихле, решения систем линейных уравнений методом Гаусса и методом простой итерации, разделения множеств и др. Описана модифицированная версия 1.1 MPI стандарта, приведено описание основных функций MPI.

Книга может служить практическим руководством по системе параллельного программирования с передачей сообщений. Изучение строится на практической основе: описываются средства параллельного программирования, предлагается ряд конкретных задач, в ходе которых рассматриваются как средства языка, так и методы программирования.

Книга рассчитана на научных работников и инженеров, использующих параллельные ЭВМ, студентов и аспирантов, изучающих параллельное программирование. Она является учебным пособием для пользователей Сибирского суперкомпьютерного центра.

Книга издана за счет средств Президиума СО РАН, направленных на развитие Сибирского суперкомпьютерного центра.

Korneev V.D. Parallel programming in MPI. – 2 ed., corrected. – Novosibirsk: Inst. of Comp. Math. and Math. Geoph.Publ., 2002. – 215 p.

The book represents of the message passing system of parallel programming based on the MPI (Message Passing Interface). This system is the basis means of programming of such modern high-performance multicomputers, as Silicon Graphics, Cray T3D, Cray T3E, IBM SP2 and many others. Many examples of parallel programming of algorithms of solution of various standard problems are considered: multiplication of a matrix by a matrix, the Dirichlet problem, solution of systems of the linear equations (SSLE) by the Gauss method and solution of the SSLE by the simple iteration method, separation of sets, etc. The modified standard version 1.1 MPI has been described, trial functions of the MPI all being described as well.

The book can serve as a practical guide in the message passing system of parallel programming. Studying is based on the practical grounds: the means of parallel programming are described, the number of concrete problems is offered, in whose solution both the means of the language, and the methods of programming are considered.

The book may be of interest for researchers and engineers, as well as for students and post-graduates, for all those involved in parallel programming.

This book is educational supply for the users of Siberian Supercomputer Center.

The publication of the book was financially supported by Presidium of SB RAS from the funds intended for the development of the Supercomputer Center.

Рецензенты:

доктор технических наук О.Л. Бандман
кандидат технических наук С.В. Пискунов
доктор физико-математических наук В.А. Вшивков

ISBN 5-901548-13-2

Предисловие редактора

После долгого (по компьютерным меркам), примерно сорокалетнего, развития параллельное программирование из экзотического занятия ученых довольно быстро превращается в массовую и популярную профессию. Невероятно быстрое развитие электронных технологий обеспечило появление на рынке большого числа коммерческих суперкомпьютеров. Они успешно применяются для решения многих задач в науке (математическое моделирование) и промышленности (управление сложными техническими устройствами).

Множество проблем параллельного программирования до сих пор не нашли хорошего решения в современных технологиях, языках и системах параллельного программирования. По этой причине параллельная реализация алгоритмов наталкивается на значительные трудности. Разработка, отладка и сопровождение параллельных программ оказываются весьма сложным делом. Нередко встречаются ситуации, когда долго и правильно работающая программа вдруг выдает ошибочные результаты.

Такие случаи известны и в последовательном программировании, однако в параллельном программировании подобные проблемы усугубляются дополнительной необходимостью правильно (содержательно и во времени) запрограммировать межпроцессные коммуникации, что очень трудно сделать человеку (как и всякие действия во времени). Кроме того, параллельные программы часто должны обладать многими необычными динамическими свойствами. Хотя, конечно, есть немало задач, сравнительно просто программируемых для исполнения на мультикомпьютерах, все же многие задачи трудно распараллелить и запрограммировать. Все это делает параллельное программирование занятием для хорошо образованных в математике и хорошо обученных нужным технологиям людей.

Наиболее распространены сейчас системы параллельного программирования на основе MPI (Message Passing Interface). Идея MPI исходно проста и очевидна. Она предполагает представление параллельной программы в виде множества параллельно исполняющихся процессов (программы процессов обычно разработаны на обычном последовательном языке программирования C или Фортран), взаимодействующих друг с другом в ходе исполнения для передачи данных с помощью коммуникационных процедур. Они и составляют библиотеку MPI. Однако надлежащая реализация MPI для обеспечения межпроцессорных коммуникаций оказалась довольно сложной. Достаточно сказать, что библиотека MPI и методы параллельного программирования с MPI изучаются на факультете прикладной математики и информатики Новосибирского государственного технического университета в течении семестра. Такая сложность связана с необходимостью достижения высокой производительности программ, необходимостью использовать многочисленные ресурсы мультикомпьютера, и, как следствие, большим разнообразием в реализации коммуникационных процедур в зависимости от режима обработки данных в процессорных элементах мультикомпьютера. При этом передачи данных между процессами программы должны осуществляться вне зависимости от того, где, на каких процессорах, реально находятся взаимодействующие процессы.

Другой источник сложности – учет в реализации коммуникаций особенностей передачи данных между процессами. Чем больше учет особенностей оборудования и режима обработки данных в процессорных элементах, тем лучше параллельная программа. Чем меньше учет - тем более общие алгоритмы используются для реализации коммуникаций, тем хуже по времени качество передачи данных. Особый бесконечный разговор – об обнаружении и локализации ошибок коммуникаций, потому как такие ошибки происходят, как правило, в одном месте программы, а их последствия обнаруживаются в другом, далёком от источника ошибки, месте.

В результате MPI оказалась большой системой и приходится затрачивать немало усилий как на ее освоение, так и на освоение методов параллельного программирования с MPI. Последнее много важнее. Настоящая книга, видимо, единственная на русском языке, в деталях описывающая MPI и способы ее применения для программирования различных прикладных задач и их решения на мультикомпьютерах.

Книга снабжена множеством примеров программирования решений небольших задач. Эти примеры дадут образцы правильных подходов к программированию прикладных задач. Примеры сами могут использоваться как готовые фрагменты для программирования других задач.

В заключение необходимо отметить, что автоматически параллельная программа не создается с MPI. MPI – это всего лишь сложный инструмент для создания параллельных программ, он может быть использован и правильно и неправильно. И если нет умения им правильно пользоваться, то и на успех рассчитывать не приходится.

В.Э. Малышкин

1. Введение

Книга посвящена параллельному программированию на базе системы с передачей сообщений MPI. Система MPI является основным средством программирования таких современных высокопроизводительных мультикомпьютеров, как Silicon Graphics Origin 2000, Cray T3D, Cray T3E, IBM SP2, МВС-1000/М и многих других. Рассмотрены многочисленные примеры параллельного программирования алгоритмов решения различных стандартных задач: умножения матрицы на матрицу, задача Дирихле, решение систем линейных уравнений (СЛАУ) методом Гаусса и решение СЛАУ методом простой итерации, разделения множеств и других. При написании книги использовались многочисленные материалы по системе MPI, наиболее полной является [1].

Для каждого из приведенных здесь примеров задач рассматриваются несколько алгоритмов ее решения, демонстрирующих, с одной стороны, разнообразие алгоритмов, приемлемых для решения одной и той же задачи в зависимости от наличия имеющихся ресурсов и характеристик этих ресурсов, а с другой, демонстрирующих применение и возможности конструкций MPI.

Книга является практическим руководством по системе параллельного программирования с передачей сообщений. Изучение системы строится на практической основе: предлагается ряд конкретных задач, описываются средства параллельного программирования, в ходе решения задач рассматриваются как средства языка, так и методы программирования. MPI стандарт предназначен для всех, кто пишет переносимые параллельные программы в Fortran'е 77, Fortran'е 90, C и C++. Этот круг пользователей включает в себя индивидуальных прикладных программистов, разработчиков программного обеспечения для параллельных вычислительных машин (или систем), создателей окружающих сред и инструментальных средств. Система MPI удобна также в качестве инструмента для обучения студентов и аспирантов параллельному программированию.

1.1. В чем особенность и сложность параллельного программирования?

Чтобы представить себе трудности параллельного программирования, нужно ответить на вопрос: в чем одно из важных отличий в написании последовательной и параллельной программ? Здесь имеется ввиду параллельная программа для рассматриваемых MIMD систем. Прежде чем создавать параллельную программу, необходимо знать общую архитектуру параллельной машины и топологию межпроцессорных связей, которая существенно используется при программировании. Это связано с тем, что невозможно создание средства автоматического распараллеливания, которое позволяло бы превращать последовательную программу в параллельную и обеспечивало бы ее высокую производительность. Поэтому в параллельной программе приходится в явном виде задавать операторы создания топологий и операторы обменов данными между процессорами. При написании же последовательной программы знать архитектуру процессора, на котором будет исполняться программа, зачастую нет необходимости, поскольку учет особенностей архитектуры скалярного процессора может быть сделан компилятором с приемлемыми потерями в производительности программы. Поэтому языковый уровень параллельной программы является заведомо ниже уровня последовательной программы, написанной на языке высокого уровня, т. к. пользователю *нужно в явном виде увязывать структуру алгоритма своей задачи со структурой вычислительной системы.* Кроме того, пользователю нужно *обеспечивать правильность взаимодействий множества параллельно выполняющихся независимо друг от друга процессов.*

В качестве примера, демонстрирующего, что распараллеливание алгоритмов простых и даже очень простых задач порой не просто, можно взять параллельную программу, разработанную Э. Дейкстрой для следующей задачи: "Даны два множества натуральных чисел S и T. Сохраняя мощности этих множеств, необходимо собрать в S наименьшие элементы

множества $S \cup T$, а в T – наибольшие". Программа состоит из полутора десятков операторов. Ее частичная корректность была доказана разными авторами. Но в 1996 г. Ю.Г. Карпов (Санкт-Петербургский технический университет) доказал отсутствие свойства ее тотальной корректности [2]. (Программа называется корректной, если при остановке она вырабатывает правильный результат. Программа называется тотально корректной, если она всегда останавливается и всегда вырабатывает правильный результат.) И построенные им простые тестовые примеры показали, что программа работает, в общем, неправильно.

1.2. Метод распараллеливания и модель программы, рассматриваемые здесь

В книге приводятся примеры параллельных алгоритмов решения следующих задач: умножения матрицы на матрицу, итерационный метод Якоби, решение систем линейных уравнений. Здесь рассматривается простой вариант сеточной задачи (метод Якоби), когда шаг сетки в пространстве вычислений одинаков и не меняется в процессе вычислений. При динамически изменяющемся шаге сетки потребовалось бы решать такую задачу параллельного программирования, как перебалансировка вычислительного пространства между компьютерами, для выравнивания вычислительной нагрузки на каждый компьютер, а значит и для повышения эффективности параллельного решения задачи. Перебалансировка вычислительной нагрузки является отдельной проблемой и здесь не рассматривается (интересующиеся могут обратиться к [3]).

Главная цель, преследуемая при решении задач на вычислительных системах, в том числе и на параллельных, – это эффективность. Эффективность параллельной программы существенно зависит от соотношения времени вычислений и времени коммуникаций между компьютерами (при обмене данными). И чем меньше в процентном отношении доля времени, затраченного на коммуникации, в общем времени вычислений, тем больше эффективность. Для параллельных систем с передачей сообщений оптимальное соотношение между вычислениями и коммуникациями обеспечивают методы крупнозернистого распараллеливания, когда параллельные алгоритмы строятся из крупных и редко взаимодействующих блоков [3–11]. Задачи линейной алгебры, задачи, решаемые сеточными методами и многие другие, достаточно эффективно *распараллеливаются крупнозернистыми методами.*

MPMD-модель вычислений. MPI программа представляет собой совокупность автономных процессов, функционирующих под управлением своих собственных программ и взаимодействующих посредством стандартного набора библиотечных процедур для передачи и приема сообщений. Таким образом, в самом общем случае MPI программа реализует MPMD-модель программирования (Multiple program – Multiple Data) [1, 12].

SPMD-модель вычислений. Все процессы исполняют в общем случае различные ветви одной и той же программы. Такой подход обусловлен тем обстоятельством, что задача может быть достаточно естественным образом разбита на подзадачи, решаемые по одному алгоритму. На практике чаще всего встречается именно эта модель программирования (Single program – Multiple Data) [1, 12].

Последнюю модель иначе можно назвать моделью *распараллеливания по данным.* Кратко суть этого способа заключается в следующем. Исходные *данные* задачи распределяются по процессам (ветвям параллельного алгоритма), а алгоритм является одним и тем же во всех процессах, но действия его распределяются в соответствии с имеющимися в этих процессах *данными.* Распределение действий алгоритма заключается, например, в присвоении разных значений переменным одних и тех же циклов в разных ветвях, либо в исполнении в разных ветвях разного количества витков одних и тех же циклов и т. п. Другими словами, процесс в каждой ветви следует различными путями выполнения на той же самой программе.

1.3. Почему MPI?

MPI является на данный момент самой развитой системой параллельного программирования с передачей сообщений. MPI позволяет создавать эффективные, надежные и переносимые параллельные программы высокого уровня.

Эффективность и надежность обеспечиваются:

1) определением MPI операций не процедурно, а логически, т.е. внутренние механизмы выполнения операций скрыты от пользователя;
2) использованием непрозрачных объектов в MPI (*группы, коммуникаторы, типы* и т.д.);
3) хорошей реализацией функций передачи данных, адаптирующихся к структуре физической системы.

Обменные функции разработаны с учетом архитектуры системы, например, для систем с распределенной памятью, систем с общей памятью и некоторых других, что позволяет минимизировать время обмена данными.

Переносимость обеспечивается:

1) тем, что тот же самый исходный текст параллельной программы на MPI может быть выполнен на ряде машин (некоторая настройка необходима, чтобы использовать преимущество элементов каждой системы). Программный код может одинаково эффективно выполняться, как на параллельных компьютерах с распределенной памятью, так и на параллельных компьютерах с общей памятью. Он может выполняться на сети рабочих станций, или на наборе процессоров на отдельной рабочей станции;
2) способностью параллельных программ выполняться на гетерогенных системах, т.е. на системах, состоящих из процессоров с различной архитектурой. MPI обеспечивает вычислительную модель, которая скрывает много архитектурных различий в работе процессоров. MPI автоматически делает любое необходимое преобразование данных и использует правильный протокол связи, посылается ли код сообщения между одинаковыми процессорами или между процессорами с различной архитектурой. MPI может настраиваться как на работу на однородной, так и на работу на гетерогенной системах;
3) такими механизмами, как определение одного вычислительного компьютера в виде *виртуального компьютера* (см. п. 2.1) и возможностью задания произвольного количества таких виртуальных компьютеров в системе независимо от количества физических компьютеров (а только от объема оперативной памяти в системе);
4) заданием *виртуальных топологий* (см. п. 2.1). Отображение виртуальных топологий на физическую систему осуществляется системой MPI. Виртуальные топологии обеспечивают оптимальное приближение архитектуры системы к структурам задач при хорошей переносимости задач;
5) компиляторами для Fortran'а и C.

Уровень языка параллельного программирования определяется языковыми конструкциями, с помощью которых создаются параллельные программы. Как было сказано выше, операторы задания топологий, обменов данными и т.п. нужно задавать в программе явно и поэтому языковый уровень параллельной программы оказывается ниже уровня последовательной программы. Наличие в системе таких средств, как виртуальные топологии, коллективные взаимодействия, создаваемые пользователем типы данных и др., значительно повышают уровень параллельного программирования по сравнению с системами с передачей сообщений, у которых нет таких средств.

1.4. Краткое содержание глав

Вторая глава посвящена общим схемам конкретных параллельных алгоритмов решения прикладных задач. При этом рассматривается несколько вариантов решения одних и тех же задач, с демонстрацией различных подходов параллельных вычислений реальных задач на реальных вычислительных системах.

В **третьей главе** рассмотрены средства MPI, позволяющие разбивать множества компьютеров на подмножества и выполнять операции над ними. Эти операции позволяют строить сложные коммуникационные области, способствующие параллельным процессам эффективно обмениваться данными.

В **четвертой главе** описываются средства задания виртуальных топологий MPI, обеспечивающие очень удобный механизм наименования процессов, связанных коммуникатором, и являющиеся мощным средством отображения процессов на оборудование системы. Виртуальные топологии являются одним из основных средств, обеспечивающих переносимость программ. И, кроме того, они позволяют оптимально отображать структуру параллельной задачи на архитектуру вычислительной системы.

В **пятой главе** подробно описан основной механизм в MPI, обеспечивающий связь между двумя взаимодействующими процессами. Эта связь называется точечной ("point-to-point"). Почти все конструкции из MPI построены на основе point-to-point операций, и ,таким образом, эта глава является основной.

В **шестой главе** описаны операции коллективных взаимодействий. Коллективная связь обеспечивает обмен данными среди всех процессов в группе, указанной аргументом `intracommunicator`. Однако они сделаны более ограниченными чем point-to-point операции. Коллективная операция выполняется при наличии всех процессов в групповом вызове оператора связи, с соответствующими аргументами.

Седьмая глава посвящена средствам конструирования пользователем собственных типов *данных*. Эти средства позволяют создавать из элементов разных типов, к тому же расположенных в несмежных участках памяти, новые типы *данных*. И эти сконструированные данные можно передавать за один вызов коммуникационной функции, что значительно уменьшает время обмена данными, делает программу более компактной.

В **восьмой главе** описываются операции для получения и установки соответствующих различных параметров, которые касаются выполнения программ, написанных с использованием MPI.

В **девятой главе** приведены примеры параллельных программ, демонстрирующих, с одной стороны, методы распараллеливания некоторых классов задач, а с другой стороны, поясняющих применение средств параллельного программирования с использованием MPI.

2. Схемы параллельных алгоритмов задач

В главе приводятся примеры параллельных алгоритмов решения следующих задач: умножения матрицы на матрицу, задача Дирихле, решение систем линейных уравнений методом Гаусса и методом простой итерации. Здесь рассматривается простой вариант сеточной задачи (задача Дирихле), когда шаг сетки в пространстве вычислений одинаков и не меняется в процессе вычислений. При динамически изменяющемся шаге сетки, как было сказано во введении в п. 1.2, потребовалось бы решать такую задачу параллельного программирования, как перебалансировка вычислительного пространства между компьютерами, для выравнивания вычислительной нагрузки компьютеров, а эта задача здесь не рассматривается.

В этой главе приводятся только общие схемы решения указанных задач, а тексты программ приведены в гл. 9, т. к. для понимания общих схем решения знать MPI не обязательно, а для понимания программ необходимо, что бы пользователь предварительно ознакомился с системой программирования MPI. Приведенные здесь параллельные алгоритмы решения задач являются иллюстрационными, демонстрирующими применение и возможности функций MPI, а не универсальными, предназначенными для библиотек алгоритмов.

Рассматриваемые задачи *распараллеливаются крупнозернистыми методами.* Для представления алгоритмов используется SPMD-модель вычислений (*распараллеливание по данным*). Однородное распределение данных по компьютерам основа для хорошего баланса времени, затрачиваемого на вычисления, и времени, затрачиваемого на взаимодействия ветвей параллельной программы. При таком распределении преследуется цель – равенство объёмов распределяемых частей данных и соответствие нумерации распределяемых частей данных нумерации компьютеров в системе. Исходными данными рассматриваемых здесь алгоритмов являются матрицы, векторы и 2D (двумерное) пространство вычислений. В этих алгоритмах применяются следующие способы однородного распределения данных: *горизонтальными полосами, вертикальными полосами и циклическими горизонтальными полосами.* При распределении *горизонтальными полосами* матрица, вектор или 2D пространство "разрезается" на полосы по строкам (далее слово "разрезанная" будем писать без кавычек и матрицу, вектор или 2D пространство обозначать для краткости словом – *данные*). Пусть M количество строк матрицы, количество элементов вектора или количество строк узлов 2D пространства, P количество виртуальных компьютеров в системе, $C_1 = M/P$ целая часть от деления, $C_2 = M\%P$ дробная часть. *Данные* разрезаются на P полос. Первые $(P - C_2)$ полос имеют по C_1 строки, а остальные C_2 полосы имеют по $C_1 + 1$ строки. Полосы *данных* распределяются по компьютерам следующим образом. Первая полоса помещается в компьютер с номером 0, вторая полоса в компьютер 1 и т. д. Такое распределение полос по компьютерам учитывается в параллельном алгоритме. Распределение вертикальными полосами аналогично предыдущему, только в распределении участвуют столбцы матрицы или столбцы узлов 2D пространства. И, наконец, распределение циклическими горизонтальными полосами. При таком распределении *данные* разрезаются на количество полос значительно большее, чем количество компьютеров. И чаще всего полоса состоит из одной строки. Первая полоса загружается в компьютер 0, вторая – в компьютер 1 и т. д., затем $(P-1)$-я полоса – снова в компьютер 0, P-я полоса – в компьютер 1 и т. д.

Приведенные два алгоритма решения СЛАУ методом Гаусса показывают, что однородность распределения *данных* сама по себе еще недостаточна для эффективности алгоритма. Эффективность алгоритмов зависит еще и от способа распределения *данных*. Разный способ представления *данных* влечет соответственно и разную организацию алгоритмов, обрабатывающих их.

В книге эффективность рассматриваемых алгоритмов определяется упрощенными формулами, которые дают грубую оценку эффективности. Это связано с тем, что здесь не рассматриваются конкретные вычислительные системы. Точные замеры эффективности

конкретного параллельного алгоритма могут быть сделаны на конкретной вычислительной системе на некотором наборе *данных*. Эффективность параллельных алгоритмов зависит, во-первых, от вычислительной системы, на которой выполняется задача, а, во-вторых, от структуры самих алгоритмов. Она определяется как отношение времени реализации параллельного алгоритма задачи ко времени реализации последовательного алгоритма этой же задачи. Эффективность можно измерять и соотношением между временем, затраченным на обмен данными между процессами, и общим временем вычислений. Заметим, что эффективность алгоритмов, которые имеют глобальный обмен *данными*, снижается с ростом числа параллельных ветвей, т. к. с ростом числа компьютеров в системе скорость выполнения глобальной операции обмена будет падать. К таким задачам можно отнести, например, решение СЛАУ итерационными методами. Эффективность алгоритмов, у которых обмен *данными* осуществляется только локально, будет неизменной с ростом числа параллельных ветвей. Например, это задачи, решаемые сеточными методами.

2.1. Запуск параллельной программы

В п. 1.3 были употреблены понятия виртуальный компьютер и виртуальная топология. Под *виртуальным компьютером* понимается программно реализуемый компьютер. Виртуальный компьютер работает в режиме интерпретации его физическим процессором. В одном физическом компьютере может находиться и работать одновременно виртуальных компьютеров – столько, сколько позволяет память физического компьютера. Работают виртуальные компьютеры в одном физическом в режиме квантования времени. Под *виртуальной топологией* здесь понимается программно реализуемая топология связей между виртуальными компьютерами на физической системе. Подробное описание виртуальной топологии представлено в гл. 4.

Создаваемая пользователем виртуальная среда позволяет обеспечивать хорошую переносимость параллельных программ, а значит и независимость от конкретных вычислительных систем. Для пользователя очень удобно решать свою задачу в рамках виртуальной среды, использовать столько компьютеров, сколько необходимо для решения его задачи, и задавать такую топологию связей между компьютерами, какая необходима. (При решении задачи на системе с небольшим количеством процессоров в одном физическом компьютере может оказаться много виртуальных компьютеров. При интерпретации виртуальных компьютеров физическим процессором естественно тратится непроизводительное время на переключение с одного виртуального компьютера на другой.)

Запуск параллельной программы зависит от типа ВС. Различаются запуски параллельных программ для сильно-связных ВС и для слабо-связных. Сильно-связными являются ВС как с разделяемой памятью, например, Silicon Graphics Origin 2000, так и с распределенной памятью с быстрыми каналами, компьютеры которых сосредоточены в небольшом пространстве, например, Cray T3D, Cray T3E, IBM SP2, МВС-1000. Слабо-связными являются ВС с компьютерами объединенными (в кластер) обычной сетью связи.

Запуск параллельной программы продемонстрируем на примере. Допустим, требуется решить некоторую задачу. Алгоритм задачи распараллелен на N процессов, независимо выполняющихся и взаимодействующих друг с другом. Вначале рассмотрим запуск программы на сильно-связной ВС и на одном обычном однопроцессорном компьютере. На сильно-связной ВС и на одном компьютере запускаются только параллельные программы, ветви которой реализуются копиями одной и той же программы. Пусть программа имеет имя: program.c. Программа предварительно компилируется:

```
mpicc [ ] -o program.exe program.c
```

В квадратных скобках стоят опции нужной оптимизации.

Затем, программа запускается командой:

```
mpirun -np N program.exe
```

N = {1,2,3, } – указывает количество виртуальных компьютеров, необходимых для решения рассматриваемой программы с именем program exe. По этой команде система MPI создает (в оперативной памяти системы из M физических компьютеров) N виртуальных компьютеров, объединенных виртуальными каналами связи со структурой полный граф. И этой группе виртуальных компьютеров присваивается стандартное системное имя MPI_COMM_WORLD. После чего пользовательская программа program exe загружается (копируется) в память каждого из созданных виртуальных компьютеров и стартует. Если M < N, то в некоторых (или всех) физических компьютерах будет создано несколько виртуальных. Виртуальные компьютеры, расположенные в одном физическом, будут работать в режиме интерпретации их физическим процессором с разделением времени. Созданные виртуальные компьютеры имеют линейную нумерацию {0,1,2,3, } и являются базой для создания различных виртуальных топологий, необходимых для реализации конкретных задач, причем со своей внутренней нумерацией виртуальных компьютеров. На некоторых системах, например, МВС-1000, в каждом физическом компьютере создается только один виртуальный компьютер.

Отображение виртуальных компьютеров и структуры их связи на конкретную физическую систему осуществляется системой MPI автоматически, т.е. пользователю не нужно переделывать свою программу для разных физических систем (с другими компьютерами и другой архитектурой). (Рассматриваемая версия MPI не позволяет пользователю осуществлять это отображение, либо осуществлять пересылку виртуальных компьютеров в другие физические компьютеры, т.е. не позволяет перераспределять виртуальные компьютеры по физическим компьютерам.) Везде далее, используя слово компьютер, мы будем иметь в виду виртуальный компьютер, если особо не оговаривается противное.

Теперь рассмотрим запуск программы на слабо-связной ВС. На слабо-связной ВС запускаются параллельные программы обоих указанных выше вариантов. Допустим, что вычислительная система имеет M $\geq$ 2 физических компьютеров с некоторой структурой связей. Далее рассматривается два варианта: 1) ВС – однородна (вычислительная система имеет одинаковые компьютеры, с одинаковыми операционными системами); 2) ВС – неоднородна. Для данного типа ВС имеется выделенный компьютер, с которого осуществляется запуск программы. Этот компьютер назовем host. Для обоих вариантов параллельной программы компиляция делается следующим образом. Для однородной ВС компиляцию программы (программ) достаточно сделать на host, а затем program exe нужно записать во все компьютеры в одноименные директории. Для программ с разными ветвями по компьютерам рассылаются только соответствующие им ветви. Для неоднородных ВС компиляцию программ обоих вариантов нужно делать на каждом компьютере и, затем, так же записать в одноименные директории. Для программ с разными ветвями на компьютерах компилируются только соответствующие им ветви.

Для однородных и неоднородных ВС запуск программы осуществляется следующей командой:

```
mpirun -mashinesfile machines s -np N program exe
```

Опция -mashinesfile указывает системе, что список физических компьютеров нужно взять в файле machines s (этот список представляет собой список IP адресов машин; полагаем, что в нем указаны M компьютеров; и этот список должен находиться в компьютере host, с которого осуществляется запуск параллельной программы, т.е. в котором выполняется команда mpirun). N = {1,2,3, } – указывает количество виртуальных компьютеров, необходимых для решения рассматриваемой программы с именем program exe. Далее работа MPI такая же, как и в описанном выше случае для сильно-связных ВС.

Далее приведем примеры файлов со списком адресов компьютеров (в вышеприведенном примере имя этого файла machines s). Для однородных и неоднородных ВС файлы одинаковы.

1. Допустим, ветви параллельной программы реализуются копиями одной и той же программы. Простой файл со списком компьютеров будет выглядеть следующим образом:

```
klast.sscc.ru
itdc-a.sscc.ru
ssd.sscc.ru
ssd2.sscc.ru
```

Предположим, пользователь заказывает 8 компьютеров, т.е. `N = 8` в команде `mpirun`. Система MPI распределит созданные виртуальные компьютеры по физическим следующим образом: `klast - 0` (слева имя физического, справа номер виртуального компьютеров), `itdc-a - 1, ssd - 2, ssd2 - 3, klast - 4, itdc-a - 5, ssd - 6, ssd2 - 7`. Перед запуском программа - program.exe предварительно должна быть размножена во всех физических компьютерах в одноименной директории, например, `home/name_p/programm exe`.

2. Количество и чередование имен компьютеров в списке может быть самым разнообразным. Допустим, у нас та же вычислительная система и тот же заказ виртуальных компьютеров, что и в предыдущем случае. Файл со списком компьютеров может быть и таким:

```
klast.sscc.ru
itdc-a.sscc.ru
ssd.sscc.ru
ssd2.sscc.ru
ssd2.sscc.ru
ssd.sscc.ru
itdc-a.sscc.ru
klast.sscc.ru
```

В этом случае система MPI распределит созданные виртуальные компьютеры по физическим следующим образом: `klast - 0` (слева имя физического, справа номер виртуального компьютеров), `itdc-a - 1, ssd - 2, ssd2 - 3, ssd2 - 4, ssd - 5, itdc-a - 6, klast - 7`. Мы видим, что распределение виртуальных компьютеров по физическим в этом случае уже другое, чем в первом случае. Таким образом, с помощью составления списка компьютеров пользователь может частично влиять на отображение виртуальных компьютеров на физические.

3. Пользователь может запускать параллельную программу, ветви которой реализуются разными программами и имеют разные имена. Допустим, четыре ветви параллельной программы имеют имена: `program.exe, program1.exe, program2.exe, program3 exe`. И эти программы записаны в разные компьютеры и в разноименные директории (директории могут быть и одноименными). `program.exe` в `klast`, в файл `file`, `program1.exe` в `itdc-a`, в файл `file1`, `program2.exe` в `ssd`, в файл `file2`, `program3.exe` в `ssd2`, в файл `file3`. Список компьютеров может быть таким:

```
klast.sscc.ru  0 home/name/file/
itdc-a.sscc.ru 1 home/name/file1/
ssd.sscc.ru    1 home/name/file2/
ssd2.sscc.ru   1 home/name/file3/
```

В этом случае в командной строке запуска программ из компьютера `host` должно стоять имя ветви, стоящей в этом же компьютере. Например, если программа в приведенном примере запускается с компьютера `klast`, то имя программы в команде `mpirun` должно быть `program.exe`.

2.2. Умножение матрицы на матрицу

Умножение матрицы на вектор и матрицы на матрицу являются базовыми макрооперациями для многих задач линейной алгебры, например итерационных методов решения систем линейных уравнений и т. п. Поэтому приведенные алгоритмы можно рассматривать как фрагменты в алгоритмах этих методов. В этом пункте приведено три алгоритма умножения матрицы на матрицу. Разнообразие вариантов алгоритмов проистекает от разнообразия вычислительных систем и размеров задач. Рассматриваются и разные варианты загрузки *данных* в систему: загрузка данных через один компьютер; и загрузка данных непосредственно каждым компьютером с дисковой памяти. Если загрузка *данных* осуществляется через один компьютер, то *данные* считываются этим компьютером с дисковой памяти, разрезаются и части рассылаются по остальным компьютерам. Но данные могут быть подготовлены и заранее, т. е. заранее разрезаны по частям и каждая часть записана на диск в виде отдельного файла со своим именем; затем каждый компьютер непосредственно считывает с диска, предназначенный для него файл.

2.2.1. Алгоритм 1

Заданы две исходные матрицы A и B. Вычисляется произведение $C = A \times B$, где A – матрица $n_1 \times n_2$, и B – матрица $n_2 \times n_3$. Матрица результатов C имеет размер $n_1 \times n_3$. Исходные матрицы предварительно разрезаны на полосы, полосы записаны на дисковую память отдельными файлами со своими именами и доступны всем компьютерам. Матрица результатов возвращается в нулевой процесс.

Реализация алгоритма выполняется на кольце из p_1 компьютеров. Матрицы разрезаны как показано на рис. 2.1: матрица A разрезана на p_1 горизонтальных полос, матрица B – на p_1 вертикальных полос и матрица результата C разрезана на p_1 полосы. Здесь предполагается, что в память каждого компьютера загружается и может находиться только одна полоса матрицы A и одна полоса матрицы B.

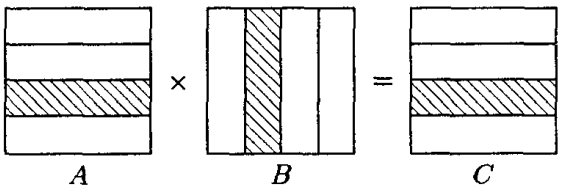


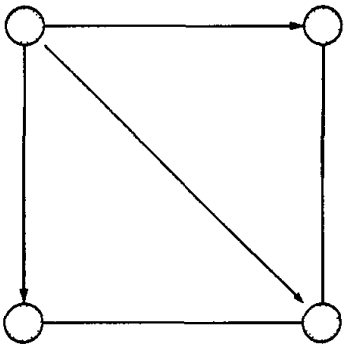
Рис. 2.1. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении на кольце компьютеров. Выделенные полосы расположены в одном компьютере

Поскольку по условию в компьютерах находится по одной полосе матриц, то полосы матрицы B (либо полосы матрицы A) необходимо "прокрутить" по кольцу компьютеров мимо полос матрицы A (матрицы B). Каждый сдвиг полос вдоль кольца и соответствующее умножение представлено на рис. 2.2 в виде отдельного шага. На каждом таком шаге вычисляется только часть полосы. Процесс i вычисляет на j-м шаге произведение i-й горизонтальной полосы матрицы A и j-й вертикальной полосы матрицы B, произведение получено в подматрице (i, j) матрицы C. Текст программы, реализующий алгоритм, приведен в гл. 9.

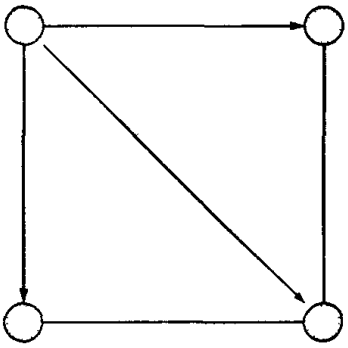
Последовательные стадии вычислений иллюстрируются на рис. 2.2.

1. Каждый компьютер считывает с диска соответствующую ему полосу матрицы A. Нулевая полоса должна считываться нулевым компьютером, первая полоса – первым компьютером и т.д., последняя полоса считывается последним компьютером. На рис. 2.2 полосы матрицы A и B пронумерованы.

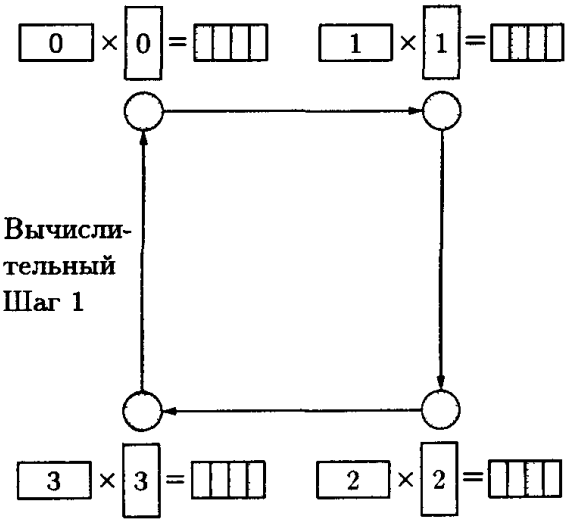
2. Каждый компьютер считывает с диска соответствующую ему полосу матрицы B. В данном случае нулевая полоса должна считываться нулевым компьютером, первая полоса – первым компьютером и т. д., последняя полоса считывается последним компьютером.

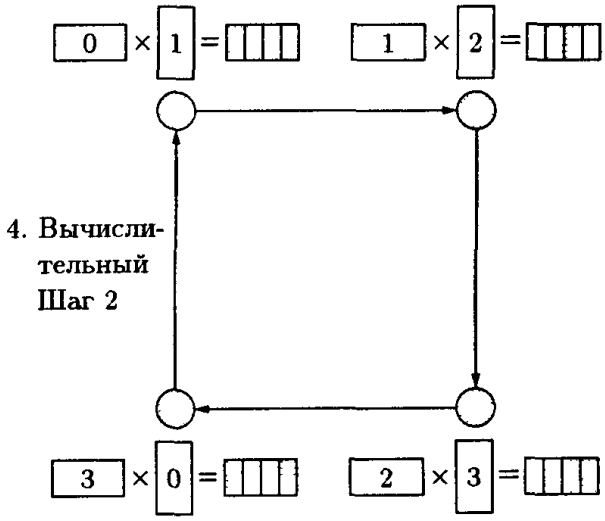


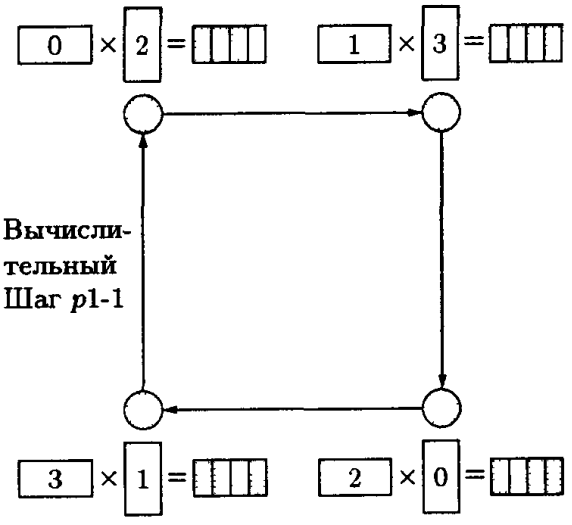
1. Scatter A

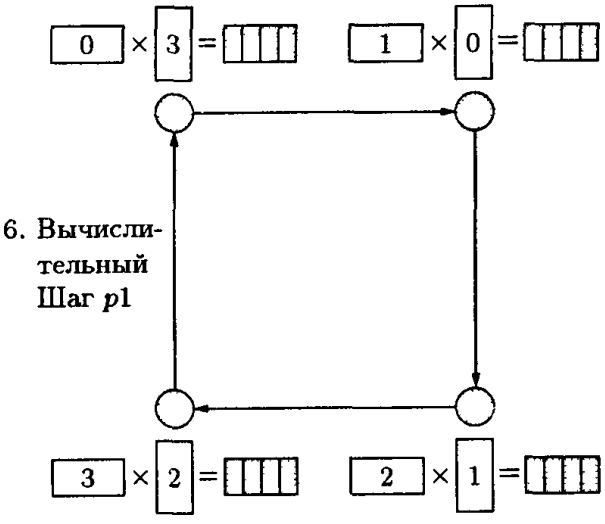


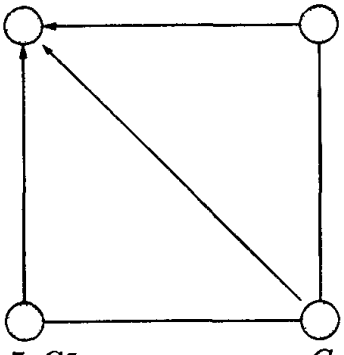
2. Scatter B











7. Сбор результатов в C

Рис. 2.2. Стадии вычислений произведения матриц в кольце компьютеров

3. Вычислительный шаг 1. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.
4. Вычислительный шаг 2. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров и т. д.
5. Вычислительный шаг $p_1 - 1$. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.
6. Вычислительный шаг p_1. Каждый процесс вычисляет одну подматрицу произведения. Вертикальные полосы матрицы B сдвигаются вдоль кольца компьютеров.
7. Матрица C собирается в нулевом компьютере.

Если "прокручивать" вертикальные полосы матрицы B, то матрица C будет распределена горизонтальными полосами, а если "прокручивать" горизонтальные полосы матрицы A, то матрица C будет распределена вертикальными полосами.

Алгоритм характерен тем, что после каждого шага вычислений осуществляется обмен данными. Пусть t_u, t_s и t_p – время операций соответственно умножения, сложения и пересылки одного числа в соседний компьютер. Согласно приведенным в начале пункта обозначениям суммарное время операций умножений равно

$$U = (n_1 * n_2 * n_3) * t_u,$$

суммарное время операций сложений равно

$$S = ((n_1 - 1) * n_2 * n_3) * t_s,$$

суммарное время операций пересылок данных по всем компьютерам равно

$$P = (n_3 * n_2) * (p_1 - 1) * t_p.$$

Тогда общее время вычислений будет равно

$$T = (U + S + P)/p_1.$$

И отношение времени "вычислений без обменов" к общему времени вычислений есть величина

$$K = (U + S)/(U + S + P).$$

Если время передачи *данных* велико по сравнению с временем вычислений, либо каналы передачи данных медленные, то эффективность алгоритма будет не высока. Здесь не учитывается время начальной загрузки и выгрузки *данных* в память системы. В полосах матриц может быть разное количество строк (различие в одну строку). При больших матрицах этим можно пренебречь. При достаточных ресурсах памяти в системе, конечно же, лучше использовать алгоритм, в котором минимизированы обмены между компьютерами в процессе вычислений. Это достигается за счет дублирования некоторых *данных* в памяти компьютеров. В следующих двух алгоритмах используется этот подход.

2.2.2. Алгоритм 2

Вычисляется произведение $C = A \times B$, где A – матрица $n_1 \times n_2$ и B – матрица $n_2 \times n_3$. Матрица результатов C имеет размер $n_1 \times n_3$. Исходные матрицы первоначально доступны на нулевом процессе, и матрица результатов возвращена в нулевой процесс. Параллельное выполнение алгоритма осуществляется на двумерной (2D) решетке компьютеров размером $p_1 \times p_2$. Матрицы разрезаны, как показано на рис. 2.3: матрица A – на p_1 горизонтальных полос, матрица B – на p_2 вертикальных полос и матрица результата C – на $p_1 \times p_2$ подматрицы (или субматрицы).

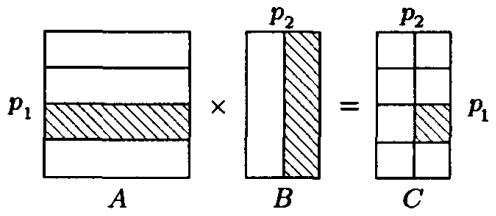


Рис. 2.3. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении в 2D решетке компьютеров. Выделенные данные расположены в одном компьютере

Каждый компьютер (i, j) вычисляет произведение i-й горизонтальной полосы матрицы A и j-й вертикальной полосы матрицы B, произведение получено в подматрице (i, j) матрицы C.

Последовательные стадии вычисления иллюстрируются на рис. 2.4:

1. Матрица A распределяется по горизонтальным полосам вдоль координаты $(x, 0)$.
2. Матрица B распределяется по вертикальным полосам вдоль координаты $(0, y)$.
3. Полосы A распространяются в измерении y.
4. Полосы B распространяются в измерении x.

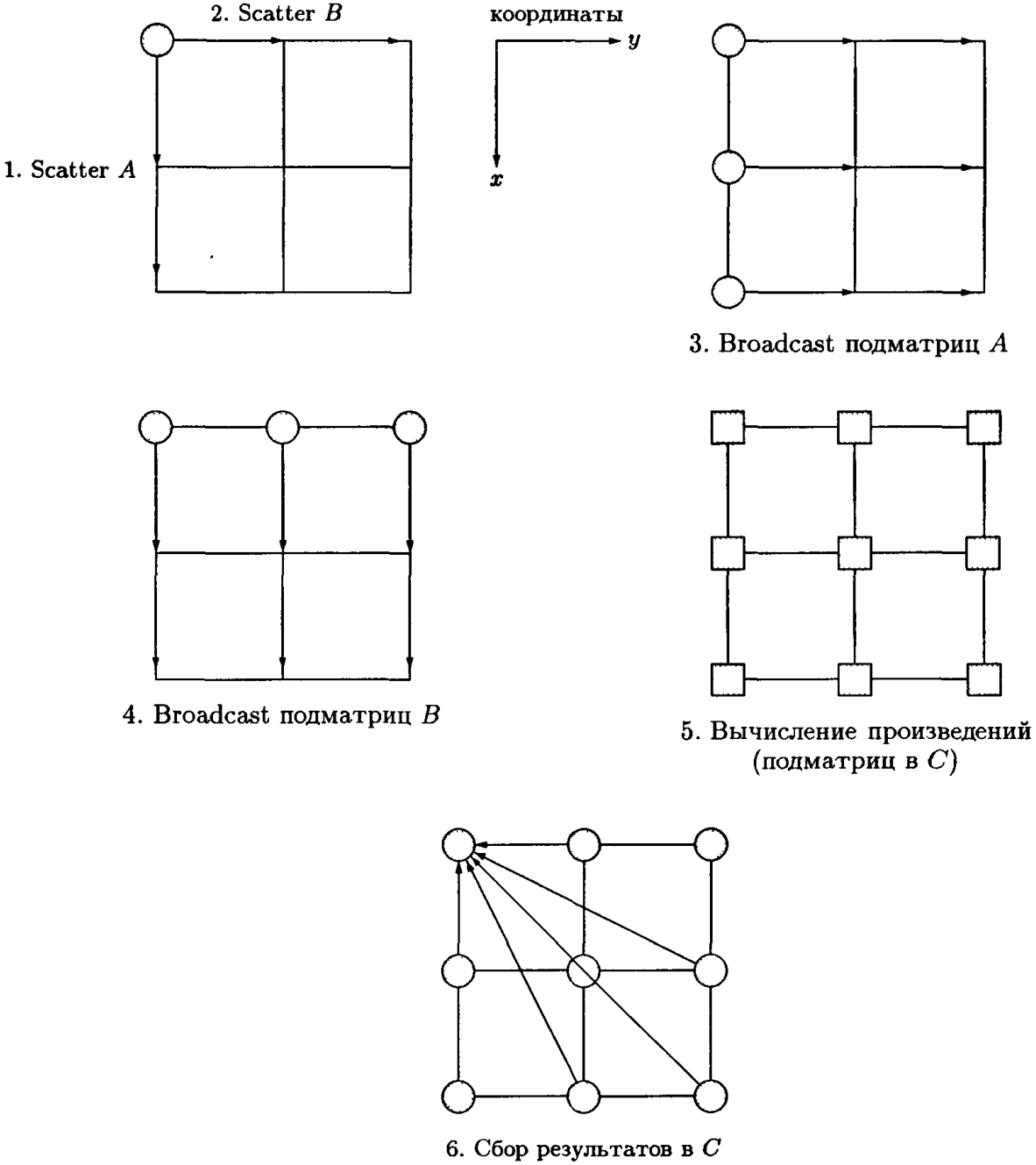


Рис. 2.4. Стадии вычисления произведения матриц в 2D параллельном алгоритме

5. Каждый процесс вычисляет одну подматрицу произведения.
6. Матрица C собирается из (x, y) плоскости.

Осуществлять пересылки между компьютерами во время вычислений не нужно, т. к. все полосы матрицы A пересекаются со всеми полосами матрицы B в памяти компьютеров системы.

Этот алгоритм эффективней предыдущего, т. к. непроизводительное время пересылок данных осуществляется только при загрузке исходных данных в память компьютеров и их выгрузке и нет обменов данными в процессе вычислений. Поскольку время обменов равно нулю, а время загрузки и выгрузки здесь не учитывается, то общее время вычислений будет равно

$$T = (U + S)/(p_1 * p_2).$$

И отношение времени "вычислений без обменов" к общему времени вычислений есть величина

$$K = (U + S)/(U + S) = 1.$$

Здесь значения T, K, U и S – см. в п. 2.2.1.

2.2.3. Алгоритм 3

Для больших матриц время вычисления произведения может быть уменьшено применением алгоритма, который осуществляет вычисление на трехмерной (пространственной) сетке компьютеров. В приведенном ниже алгоритме осуществляется отображение основных данных объемом $n_1 \times n_2 + n_2 \times n_3 + n_1 \times n_3$ на объемную сетку компьютеров размером $p_1 \times p_2 \times p_3$. Матрицы разрезаны, как показано на рис. 2.5: матрица A – на $p_1 \times p_2$ субматрицы, матрица B – на $p_2 \times p_3$ субматрицы и матрица C разрезана на $p_1 \times p_3$ субматрицы. Компьютер (i, j, k) вычисляет произведение субматрицы (i, j) матрицы A и субматрицы (j, k) матрицы B. Субматрица (i, k) матрицы C получается суммированием промежуточных результатов, вычисленных в компьютерах (i, j, k), $j = 0, \ldots, p_2 - 1$.

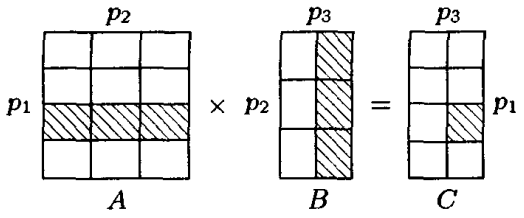
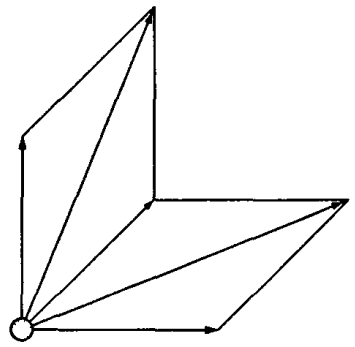


Рис. 2.5. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении в 3D решетке компьютеров

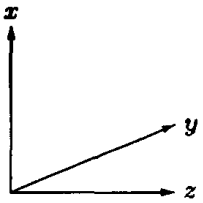
Последовательные стадии вычисления иллюстрируются на рис. 2.6.

1. Субматрицы A распределяются в $(x, y, 0)$ плоскости.
2. Субматрицы B распределяются в $(0, y, z)$ плоскости.
3. Субматрицы A распространяются в измерении z.
4. Субматрицы B распространяются в измерении x.
5. Каждый процесс вычисляет одну субматрицу.
6. Промежуточные результаты редуцируется в измерении y.
7. Матрица C собирается из $(x, 0, z)$ плоскости.

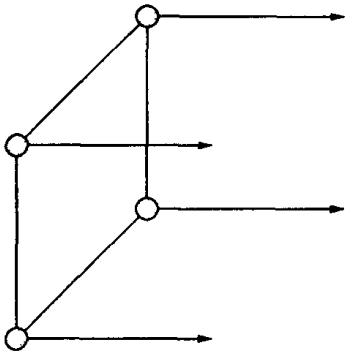
Этот алгоритм похож на предыдущий, но дополнительно разрезаются еще полосы матриц, и эти разрезанные полосы распределяются в третьем измерении y. В данном случае в каждом компьютере будут перемножаться только части векторов строк матрицы A и части столбцов матрицы B и в результате будет только частичная сумма для каждого



1. Scatter A

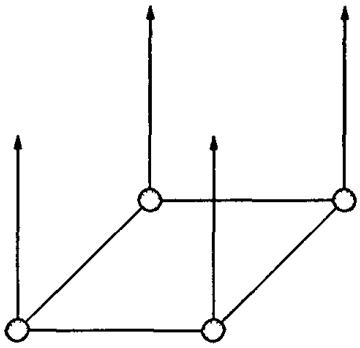


координаты

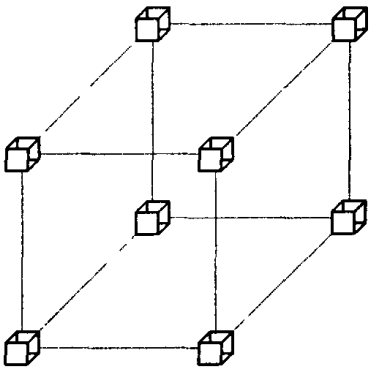


3. Broadcast подматриц из A

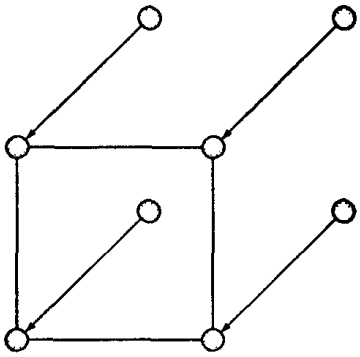
2. Scatter B



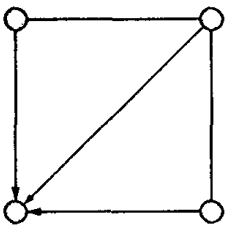
4. Broadcast подматриц из B



5. Вычисление произведений (подматриц из C)



6. Reduce произведений (суммирование произведений)



7. Gather C (сбор результатов)

Рис. 2.6. Стадии вычислений в 3D параллельном алгоритме произведения матриц

элемента результирующей матрицы C. Операция суммирования вдоль координаты y этих полученных частичных сумм для результирующих элементов и завершает вычисления матрицы C.

Этот алгоритм для больших матриц является еще более эффективным, чем предыдущий. Соотношения для общего времени вычислений в этом алгоритме будет равно

$$T = (U + S)/(p_1 * p_2 * p_3).$$

А отношение времени "вычислений без обменов" к общему времени вычислений есть величина

$$K = (U + S)/(U + S) = 1.$$

Значения T, K, U и S – см. ранее.

2.3. Задача Дирихле. Явная разностная схема для уравнения Пуассона

В прямоугольной области $0 \leq x \leq a$, $0 \leq y$ требуется найти решение уравнения:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = g(x, y)$$

при заданных значениях функции u на границе.

Явная разностная схема решения этого уравнения имеет вид:

$$u_{i,m}^{k+1} = 0.25 * (u_{i-1,m}^{k} + u_{i+1,m}^{k} + u_{i,m-1}^{k} + u_{i,m+1}^{k} - h^2 g_{im}),$$

где u_{im} и g_{im} – значения функций и в точке разностной сетки. Ниже представлен главный фрагмент программы итерационного решения задачи.

```
double A[n+2][m+2], B[n][m];
...
/* Главный цикл */
while(! converged) {
/* выполнение схемы "крест" */
for(j = 1; j <= m; j++)
  for(i = 1; i <= n; i++)
    B[i-1][j-1] = 0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
/* копирование результата обратно в массив A */
for(j = 1; j <= m; j++)
  for(i = 1; i <= n; i++)
    A[i][j] = B[i-1][j-1];
...
 }
```

Этот фрагмент программы описывает главный цикл итерационного процесса решения, где на каждой итерации значение в окрестности точки заменяется средним значением сумм значений ее четырех соседних точек на предыдущем временном шаге итерации (рис. 2.7).

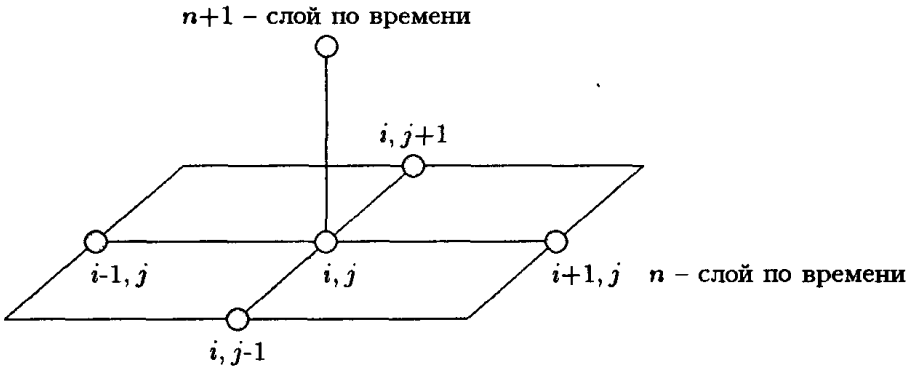


Рис. 2.7. Вычисление точки через значения точек на предыдущем шаге итерации

Граничные значения не изменяются. В массиве B вычисляются значения следующей итерации, а в массиве A находятся значения предыдущей итерации. Здесь приведен внутренний цикл, где выполнено большинство вычислений. В первой и последней строке, а также в первом и последнем столбце двумерного массива A записаны граничные значения.

Алгоритм этой задачи имеет простую структуру, идентичную для всех точек пространства вычисления, поэтому здесь при построении параллельной программы целесообразно использовать метод распараллеливания по данным. Массивы "разрезаются" на части, затем в каждый процессор загружается программа, аналогичная последовательной программе, но с модифицированными значениями индексов циклов и соответствующая часть массива данных, т. е. каждый процессор обрабатывает только часть данных. Части данных соответственно упорядочены.

Способы разрезания данных могут быть разные, и в зависимости от способа разрезания данных будут и разные схемы взаимодействий между параллельными процессами. Распределение данных по процессорам должно быть сбалансировано. Связь между процессорами должна быть минимизирована.

Двумерный массив может быть разрезан на части как в одном измерении, так и в двух измерениях. Рис. 2.8 поясняет эти способы разрезания данных: 1D разрезание – матрица разрезана в одном измерении полосами и 2D разрезание – матрица разрезана в двух измерениях.

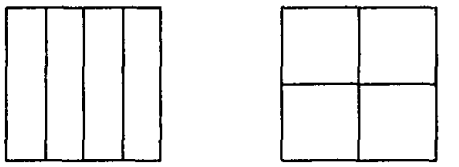


Рис. 2.8. Два разных способа разрезания двумерного массива на блоки

На рисунке матрица разрезана на четыре блока; каждый блок обрабатывается в отдельном процессоре. Так как связь между процессорами осуществляется границами блоков, то объем связи минимизирован в 2D разрезании, которое имеет меньший периметр области связи. В этом разбиении каждый процессор взаимодействует, в общем случае, с четырьмя соседями быстрее, чем два соседа в 1D разрезании. Когда отношение n/P (P – число процессоров) маленькое, время связи будет зависеть от внешних коммуникаций, и первое разбиение будет лучше по взаимодействиям. Когда это отношение большое, второе разбиение лучше по взаимодействиям. Здесь используется первое разбиение.

Значение каждой точки в массиве B вычисляется через значения четырех ее соседей в массиве A. Связь между процессорами необходима границами блоков, чтобы вычислять граничные точки блоков через свои соседние точки, которые принадлежат другому процессору. Так как точки вычисляются через только свои соседние точки, вычисленные на предыдущей итерации, то для вычисления граничных точек блока необходимо присутствие копии соответствующего столбца предыдущей итерации соседнего блока, находящегося в соседнем компьютере. Следовательно, при 1D разрезании необходимо распределение блоков массива A по компьютерам с перекрытием в один столбец. Таких перекрытий столбцами для массива B не нужно – это следует из алгоритма.

На рис. 2.9 иллюстрируется перекрытие соседних полос в один столбец: римскими цифрами обозначены крайние столбцы соседних полос, пунктирной линией – граница разреза массива данных.

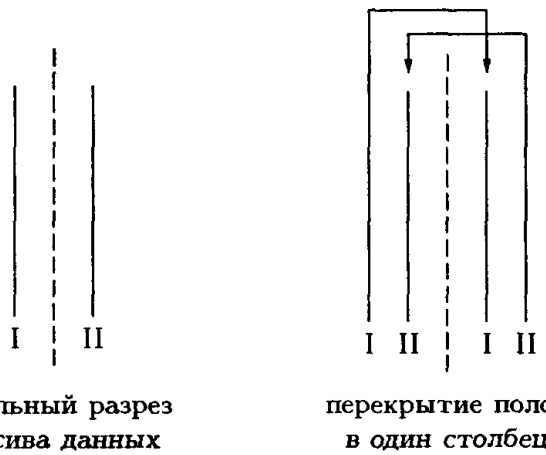


Рис. 2.9. Перекрытие двух полос в один столбец

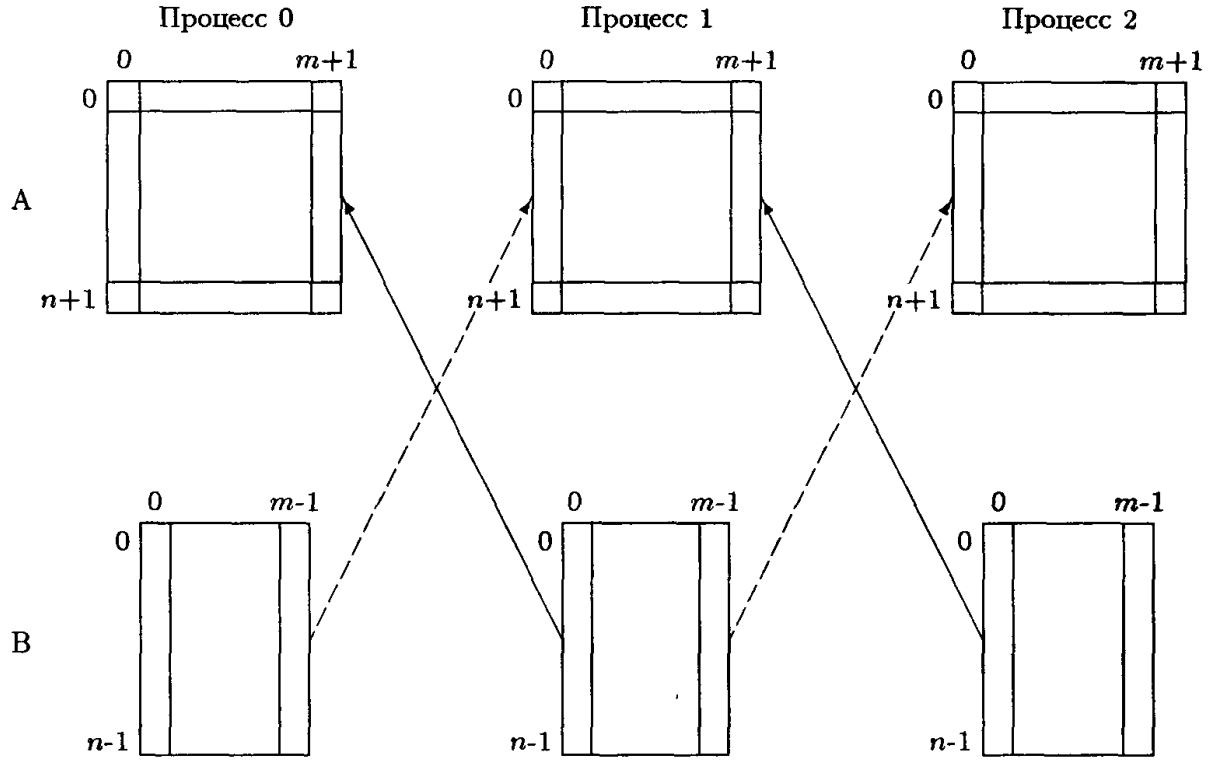


Рис. 2.10. Перекрытие 1D блоков в массиве A и пересылка столбцов в конце каждой итерации

Алгоритм и схема обменов данными аналогичны и при 2D разрезании массивов. В этом случае обмен данными будет осуществляться с четырьмя соседними компьютерами. Подобно этой задаче аналогичным методом распараллеливания по данным решаются итерационные задачи размерности больше трех.

На рис. 2.10 показаны дополнительные столбцы массива A и то, как данные пересылаются после каждой итерации. Перекрытие одним столбцом обычно используется для разностной схемы, представляемой пятиточечным шаблоном сетки. Для разностной схемы, использующей девятиточечный шаблон сетки, необходимо делать двух столбцовое перекрытие полос массивов, находящихся в смежных компьютерах. В общем случае, если рассматривается окрестность с диаметром в k точек от вычисляемой точки на предыдущем шаге вычислений, то перекрытие массивов нужно делать в k столбцов.

Фрагменты программ рассмотренного здесь параллельного алгоритма приведены в гл. 9. Выше приведена общая схема параллельного алгоритма решения задачи. При реализации этой схемы могут использоваться разные языковые средства MPI, приспосабливающие программу к вычислительной системе. В гл. 9 приведено несколько разных фрагментов программ решения рассмотренной задачи, демонстрирующих возможности конструкций MPI, в частности, возможность совмещения пересылок данных с вычислениями. При этом способе в каждом процессоре вначале вычисляются граничные точки массивов, после чего запускается процесс передачи этих граничных точек соседним компьютерам и затем продолжается вычисление внутренних точек массивов. При этом передача данных и вычисления будут частично перекрываться во времени.

Эффективность этого алгоритма зависит от соотношения количества обрабатываемых точек в каждом компьютере к количеству передаваемых компьютером граничных точек соседним компьютерам. Эффективность, конечно же, зависит и от самого алгоритма, например, вычисления организуются с перекрытием обменов *данными* или без перекрытия. Практика показывает, что подобные задачи решаются параллельными методами очень эффективно, т.е. временные затраты на обмен данными в процентном соотношении ко всему времени решения очень незначительны.

2.4. Параллельные алгоритмы решения систем линейных алгебраических уравнений методом Гаусса

Требуется найти решение системы линейных алгебраических уравнений:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = f_1, \\ &a_{21}x_2 + a_{22}x_2 + \ldots + a_{2n}x_n = f_2, \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &a_{n_1}x_1 + a_{n_2}x_2 + \ldots + a_{nn}x_n = f_n. \end{aligned} \quad l$$

Метод Гаусса основан на последовательном исключении неизвестных. Здесь рассматриваются два алгоритма решения СЛАУ методом Гаусса. Они связаны с разными способами представления *данных* (матрицы коэффициентов и правых частей) в распределенной памяти мультикомпьютера.

2.4.1. Первый алгоритм решения СЛАУ методом Гаусса

В алгоритме, представленном здесь, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами, как показано на рис. 2.11. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая – в первый компьютер и т. д., последняя полоса – в p_1 компьютер.

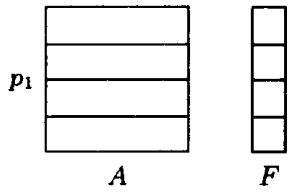


Рис. 2.11. Разрезание данных для параллельного алгоритма 1 решения СЛАУ методом Гаусса

При прямом ходе матрица приводится к треугольному виду последовательно по компьютерам. Вначале к треугольному виду приводятся строки в нулевом компьютере, при этом нулевой компьютер последовательно, строка за строкой, передает свои сроки остальным компьютерам, начиная с первого. Затем к треугольному виду приводятся строки в первом компьютере, передавая свои строки остальным компьютерам, начиная со второго, т. е. компьютерам с большими номерами, и т. д. Процесс деления строк на коэффициенты при x_i не требует информации от других компьютеров.

После прямого хода полосы матрицы A в каждом узле будут иметь вид (рис. 2.12)

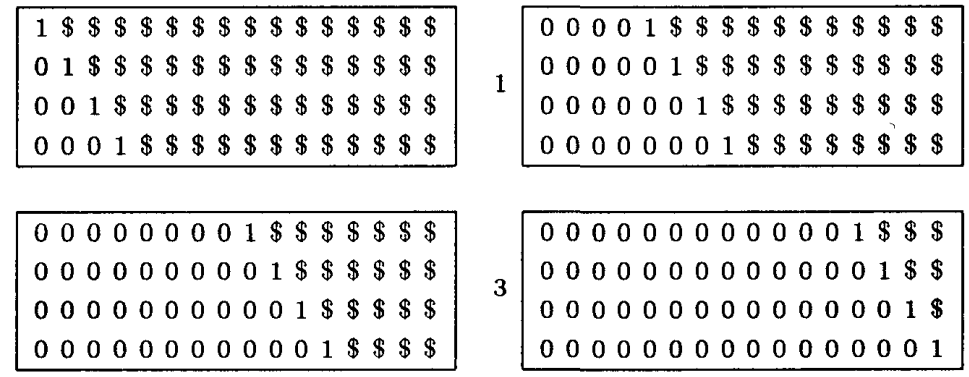


Рис. 2.12. *Вид полос после прямого хода в алгоритме 1 решения СЛАУ методом Гаусса.* Пример приведен для четырех узлов, $ – вещественные числа

Аналогично, последовательно по компьютерам, начиная с последнего, по номеру компьютера осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе, компьютеры, завершившие свою часть работы, переходят в состояние ожидания, пока не завершат эту работу другие компьютеры. Таким образом, вычислительная нагрузка распределяется по компьютерам неравномерно, не смотря на то, что *данные* изначально распределяются по компьютерам приблизительно одинаково. Простои компьютеров значительно уменьшаются при распределении матрицы циклическими горизонтальными полосами. Этот метод представлен в п. 2.4.2.

2.4.2. Второй алгоритм решения СЛАУ методом Гаусса

В алгоритме, представленном здесь, исходная матрица коэффициентов распределяется по компьютерам циклическими горизонтальными полосами с шириной полосы в одну строку, как показано ниже на рис. 2.13.

Первая строка матрицы помещается в компьютер 0, вторая строка – в компьютер 1 и т.д., p_1 – 1-я строка – в узел p_1 (где p_1 – количество узлов в системе). Затем p_1-я строка снова помещается в узел 0, p_1 + 1-я строка – в узел 1 и т.д.

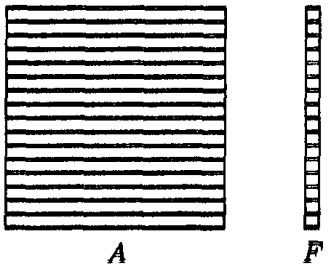


Рис. 2.13. Разрезание *данных* для параллельного алгоритма 2 решения СЛАУ методом Гаусса

При таком распределении данных, соответствующим этому распределению должен быть и алгоритм. Строку, которая вычитается из всех остальных строк (после предварительного деления на нужные коэффициенты), назовем текущей строкой. Алгоритм прямого хода заключается в следующем. Сначала текущей строкой является строка с индексом 0 в компьютере 0, затем строка с индексом 0 в компьютере 1 (здесь не нужно путать общую нумерацию строк во всей матрице и индексацию строк в каждом компьютере; в каждом компьютере индексация строк в массиве начинается с нуля) и т. д., и наконец, строка с индексом 0 в последнем по номеру компьютере. После чего цикл по компьютерам повторяется и текущей строкой становится строка с индексом 1 в компьютере 0, затем строка с индексом 1 в компьютере 1 и т.д. После прямого хода полосы матрицы в каждом компьютере будут иметь вид, показанный на рис. 2.14. Пример приведен для четырех узлов; $ – вещественные числа.

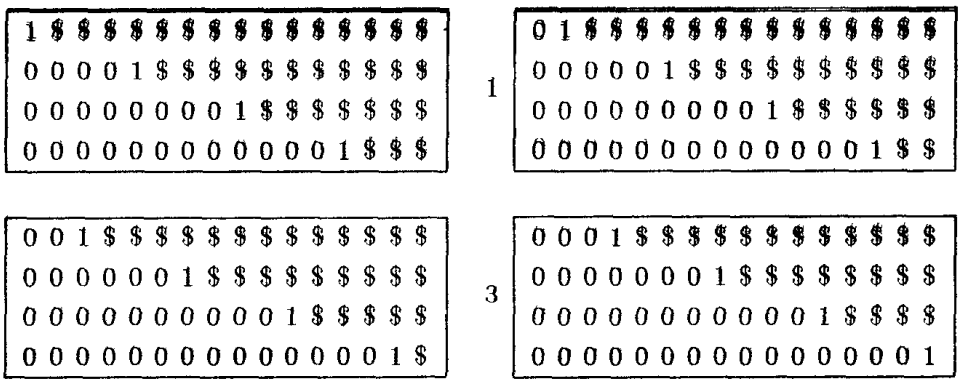


Рис. 2.14. Вид полос после прямого хода в алгоритме 2 решения СЛАУ методом Гаусса

Аналогично, последовательно по узлам, начиная с последнего по номеру компьютера, осуществляется обратный ход.

Особенностью этого алгоритма является то, что как при прямом, так и при обратном ходе компьютеры более равномерно загружены, чем в первом методе. Значит и вычислительная нагрузка распределяется по компьютерам более равномерно, чем в первом методе. Например, нулевой компьютер, завершив обработку своих строк при прямом ходе, ожидает пока другие компьютеры обработают только по одной, оставшейся у них не обработанной строке, а не полностью обработают полосы, как в первом алгоритме.

Сравним второй алгоритм с первым. При более равномерной загрузке компьютеров при вычислении одного алгоритма по сравнению с другим алгоритмом следует предположить и большую эффективность алгоритма с более равномерной загрузкой компьютеров. Загрузка компьютеров во втором алгоритме является более равномерной. Но большая его эффективность, по сравнению с первым, может проявиться только на исходных матрицах большого размера (например, начиная с исходных матриц 400×400 и более, но это зависит от конкретной системы). Это обстоятельство связано с тем, что в первом методе в процессе вычислений активных компьютеров становится все меньше, а значит и уменьшается количество пересылок своих строк другим компьютерам. С уменьшением числа активных компьютеров будет уменьшаться и общее время, затрачиваемое на пересылку строк в активные компьютеры. И это частично компенсирует неравномерность вычислительной загрузки компьютеров. Во втором методе компьютеры активны в течение всего времени вычислений и пересылка строк осуществляется всегда во все компьютеры. Затраты на пересылку одной строки из разных компьютеров в этом случае будут всегда максимальны.

2.5. Параллельный алгоритм решения СЛАУ методом простой итерации

Здесь рассматривается параллельный алгоритм решения СЛАУ методом простой итерации. Приближенные решения (итерации) системы линейных уравнений последовательно находятся по формуле:

$$y_i^{(k+1)} = y_i^{(k)} - \tau \left(\sum_{j=1}^{N} a_{ij} y_j^{(k)} - b_i^{(k)} \right), \quad i = 1, 2, \ldots, N. \tag{1}$$

Для решения этой задачи на параллельной системе исходную матрицу коэффициентов A разрезаем на p_1 горизонтальных полосы по строкам, где p_1 – количество компьютеров в системе. Аналогично, горизонтальными полосами разрезаются вектор b (правая часть) и векторы y^0 (начальное приближение), y^k (текущее приближение) и y^{k+1} (следующее приближение). Полосы последовательно распределяются по соответствующим компьютерам системы, как и в описанном выше первом алгоритме умножения матрицы на матрицу.

Здесь выражение $\sum\limits_{j=1}^{N} a_{ij} y_j^{(k)}$ есть умножение матрицы на вектор, параллельный алгоритм которого представлен в п. 9.2.1. Таким образом, этот алгоритм является составной частью, описываемого в данном пункте алгоритма. В каждом компьютере системы вычисляется "свое" подмножество корней. Поэтому после нахождения приближенных значений корней на очередном шаге итерации в каждом компьютере проверяется выполнение следующего условия для "своих" подмножеств корней:

$$\| y_i^{(k+1)} - y_i^{(k)} \| \leq \varepsilon. \tag{2}$$

Это условие в некоторых компьютерах системы в текущий момент может выполняться, а в некоторых нет. Но условием завершения работы каждого компьютера является безусловное выполнение условия (2) во всех компьютерах. Таким образом, прежде чем завершить работу, при выполнении условия (2), каждый компьютер должен предварительно узнать: во всех ли компьютерах выполнилось условие (2)? И если условие (2) не выполнилось хотя бы в одном компьютере, то все компьютеры должны продолжить работу. Это обстоятельство связано с тем, что в операции умножения матрицы на вектор участвуют все компьютеры,

взаимодействуя друг с другом. И цепочку этих взаимодействий прерывать нельзя, если хотя бы в одном из компьютеров не выполнится условие (2). При не выполнении условия (2), каждый процесс передает всем остальным процессам полученную итерацию своих корней. И тем самым вектор y полностью восстанавливается в каждом процессе для выполнения операции его умножения на матрицу коэффициентов на следующем шаге итерации.

9. Примеры параллельных программ

В этой главе приведены примеры параллельных программ, демонстрирующих, с одной стороны, методы распараллеливания некоторых классов задач, а с другой стороны, поясняющих применение средств параллельного программирования MPI. Приведены параллельные алгоритмы следующих задач: умножения матрицы на матрицу, задачи Дирихле, решения систем линейных уравнений (СЛАУ) методом Гаусса и решения СЛАУ методом простой итерации. Общие схемы распараллеливания этих задач приведены в гл. 2. Для каждой из них приводится несколько вариантов программ. Например, приведено три алгоритма умножения матрицы на матрицу. Разнообразие вариантов алгоритмов проистекает от разнообразия вычислительных систем и размеров задач. Здесь рассматриваются и разные варианты загрузки *данных* в систему: загрузка данных через один компьютер (например, нулевой) и загрузка данных непосредственно каждым компьютером с дисковой памяти. Если загрузка *данных* осуществляется через один компьютер, то в этом случае *данные* считываются этим компьютером с дисковой памяти, разрезаются и части рассылаются по компьютерам. Но *данные* могут быть подготовлены и заранее, т.е. заранее разрезаны по частям и каждая часть записана на диск в виде отдельного файла со своим именем; затем каждый компьютер непосредственно считывает с диска предназначенный для этого компьютера файл. Приводится два варианта алгоритмов решения СЛАУ методом Гаусса. Эти алгоритмы показывают, что равномерного распределения данных по компьютерам еще не достаточно для сбалансированной и эффективной работы, а важен еще и способ распределения данных.

В начале рассматривается несколько простых задач, поясняющих программирование разных ветвей параллельной программы и их взаимодействие.

9.1. Простые примеры

В данном пункте приведены простые примеры параллельных программ, демонстрирующих программирование ветвей параллельной программы в простых случаях. Первый пример – это обмен данными между двумя ветвями параллельной программы без задания топологий. Рассматривается пример приема данных неизвестной длины. Приводятся примеры с заданием виртуальных топологий и работы ветвей в этих топологиях, например, обмен данными между ветвями параллельной программы на кольце, в двумерной решетке.

9.1.1. Обмен данными между двумя ветвями параллельной программы

Программируется работа двух ветвей параллельной программы. Одна ветвь передает какие-то данные другой ветви.

```
/*
 * Простая передача-прием:  MPI_Send, MPI_Recv
 * Завершение по ошибке:    MPI_Abort
 */

  #include <mpi.h>
  #include <stdio.h>
/* Идентификаторы сообщений */
  #define tagFloatData  1
  #define tagDoubleData 2
/* Этот макрос введен для удобства, */
/* он позволяет указывать длину массива в количестве ячеек */
  #define ELEMS(x)  ( sizeof(x) / sizeof(x[0]) )

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

```
{
  int size, rank, count;
  float floatData[10];
  double doubleData[20];
  MPI_Status status;
/* Инициализация библиотеки MPI*/
  MPI_Init(&argc, &argv);
/* Каждая ветвь узнает количество задач в стартовавшем приложении */
  MPI_Comm_size(MPI_COMM_WORLD, &size);
/* и свой собственный номер: от 0 до (size-1) */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Пользователь должен запустить ровно две задачи, иначе ошибка */
  if(size != 2)
    {
   /* Задача с номером 0 сообщает пользователю об ошибке */
      if(rank == 0)
        printf("Error: two processes required instead of %d, abort\n",
                                                              size);

   /* Все ветви в области связи MPI_COMM_WORLD будут стоять,
    * пока ветвь 0 не выведет сообщение.
    */
      MPI_Barrier(MPI_COMM_WORLD);
   /* Без точки синхронизации может оказаться, что одна из ветвей
    * вызовет MPI_Abort раньше, чем успеет отработать printf()
    * в ветви 0, MPI_Abort немедленно принудительно завершит
    * все ветви и сообщение выведено не будет
    */
   /* все задачи аварийно завершают работу */
      MPI_Abort(
        MPI_COMM_WORLD,  /* Описатель области связи, на которую */
                         /* распространяется действие ошибки */
        MPI_ERR_OTHER);  /* Целочисленный код ошибки */
      return -1;
    }
  if(rank == 0)
    {
   /* Ветвь 0 что-то передает ветви 1 */
      MPI_Send(
        floatData,       /* 1) адрес передаваемого массива */
        5,               /* 2) сколько: 5 элементов, т.е.
                            floatData[0]..floatData[4] */
        MPI_FLOAT,       /* 3) тип элементов */
        1,               /* 4) кому: ветви 1 */
        tagFloatData,    /* 5) идентификатор сообщения */
        MPI_COMM_WORLD); /* 6) описатель области связи, через
                            которую происходит передача */

   /* и еще одна передача: данные другого типа */

      MPI_Send(doubleData, 6, MPI_DOUBLE, 1, tagDoubleData,
                                             MPI_COMM_WORLD);
```

```
    else
      {
     /* Ветвь 1 что-то такое принимает от ветви 0 */
     /* дожидается сообщения и помещает пришедшие данные в буфер */
        MPI_Recv(
          floatData,          /* 1) адрес массива, куда складывать
                                 принятое */
          ELEMS(floatData),   /* 2) фактическая длина приемного
                                 массива в числе элементов */
          MPI_FLOAT,          /* 3) сообщаем MPI, что пришедшее
                                 сообщение состоит из чисел
                                 типа 'float' */
          0,                  /* 4) от кого: от ветви 0 */
          tagFloatData,       /* 5) ожидаем сообщение с таким
                                 идентификатором */
          MPI_COMM_WORLD,     /* 6) описатель области связи, через
                                 которую ожидается приход сообщения */
          &status);           /* 7) сюда будет записан статус завершения
                                 приема */

     /* Вычисляем фактически принятое количество данных */
        MPI_Get_count(
          &status,       /* статус завершения */
          MPI_FLOAT,     /* сообщаем MPI, что пришедшее сообщение
                            состоит из чисел типа 'float' */
          &count);      /* сюда будет записан результат */

     /* Выводим фактическую длину принятого на экран */
        printf("rank = %d Received %d elems\n", rank, count);

     /* Аналогично принимаем сообщение с данными типа double
      */
        MPI_Recv(doubleData, ELEMS(doubleData), MPI_DOUBLE,
                      0, tagDoubleData, MPI_COMM_WORLD, &status );
        MPI_Get_count(&status, MPI_DOUBLE, &count);
      }

  /* Обе ветви завершают выполнение */
     MPI_Finalize();
     return 0;
  }
```

9.1.2. Прием сообщений неизвестной длины

Прием сообщений от разных отправителей с разными идентификаторами (с содержимым разных типов) в произвольном порядке: MPI_ANY_SOURCE, MPI_ANY_TAG-(джокеры).

```
/*
 * Прием сообщений неизвестной длины:
 * MPI_Probe
 */
   #include <mpi.h>
   #include <stdio.h>
   #include <stdlib.h>
```

```
   #include <time.h>

/* Идентификаторы (теги) сообщений */
   #define tagFloatData  1
   #define tagLongData   2

/* Длина передаваемых сообщений может быть случайной от 1 до
 * maxMessageElems
 */
   #define maxMessageElems 100

int main(int argc, char **argv)
   {
     int        size, rank, count, i, n, ok;
     float     *floatPtr,
     int       *longPtr,
     char      *typeName,
     MPI_Status status,

  /* Инициализация библиотеки и сообщение об ошибке целиком перенесены
   * из предыдущего примера
   */
     MPI_Init(&argc, &argv),
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank),

  /* Пользователь должен запустить ровно ТРИ ветви (процесса), иначе
   * ошибка */
     if(size != 3)
       {
         if(rank == 0)
           printf("Error. 3 processes required instead of %d\n",size),
         MPI_Barrier(MPI_COMM_WORLD),
         MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER),
         return -1;
       }

  /* Каждая ветвь инициализирует генератор случайных чисел */
     srand( ( rank + 1 ) * (unsigned )time(0) ),

     switch(rank)
       {
         case 0:
           /* Создаем сообщение случайной длины */
              count = 1 + rand() % maxMessageElems;
              floatPtr = malloc(count * sizeof(float));
              for(i = 0, i < count, i++)
                floatPtr[i] = (float)i;
           /* Посылаем сообщение в ветвь 2 */
              MPI_Send(floatPtr, count, MPI_FLOAT, 2, tagFloatData,
                                                   MPI_COMM_WORLD),
              printf("%d  Send %d float items to proc 2\n", rank, count),
              break;
```

```
case 1:
  /* Создаем сообщение случайной длины */
     count = 1 + rand() % maxMessageElems;
     longPtr = malloc(count * sizeof(long));
     for(i = 0; i < count; i++)
     longPtr[i] = i;
  /* Посылаем сообщение в ветвь 2 */
     MPI_Send(longPtr, count, MPI_LONG, 2, tagLongData,
                                          MPI_COMM_WORLD);
     printf("%d. Send %d long items to proc.2\n", rank, count);
     break;
case 2:
  /* Ветвь 2 принимает сообщения неизвестной длины,
   * используя MPI_Probe
   */
     for(n=0; n<2; n++)  /* Всего ожидаются два сообщения */
       {
         MPI_Probe(
           MPI_ANY_SOURCE,  /* Джокер: ждем от любой задачи */
           MPI_ANY_TAG,     /* Джокер: ждем с любым идентификатором*/
           MPI_COMM_WORLD, &status);
      /* MPI_Probe вернет управление, когда сообщение будет
       * уже на приемной стороне в служебном буфере
       */

      /* Проверяем идентификатор и размер пришедшего сообщения
       */
         if(status.MPI_TAG == tagFloatData)
           {
             MPI_Get_count(&status, MPI_FLOAT, &count);
          /* Принятое будет размещено в динамической памяти:
           * заказываем в ней буфер соответствующей длины
           */
             floatPtr = malloc(count * sizeof(float));
             MPI_Recv(floatPtr, count, MPI_FLOAT, MPI_ANY_SOURCE,
                           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
          /* MPI_Recv просто скопирует уже принятые данные
           * из системного буфера в пользовательский
           */
          /* Проверяем принятое */
             for(ok = 1, i=0; i < count; i++)
               if(floatPtr[i] != (float)i)
                 ok = 0;
                 typeName = "float";
           }
         else if(status.MPI_TAG == tagLongData)
           {
          /* Действия, аналогичные выше описанным
           */
             MPI_Get_count(&status, MPI_LONG, &count);
             longPtr = malloc(count * sizeof(long));
             MPI_Recv(longPtr, count, MPI_LONG, MPI_ANY_SOURCE,
                         MPI_ANY_TAG, MPI_COMM_WORLD, &status );
```

```
                    for(ok = 1, i = 0, i < count, i++)
                      if(longPtr[i] != i)
                        ok = 0,
                        typeName = "long",
                    }

                /* Сообщаем о завершении приема */
                   printf( "%d  %d %s items are received from %d   %s\n",
                            rank, count, typeName, status MPI_SOURCE,
                            ok ? "OK"   "FAILED" ),

                 }     /* for(n) */
              break,
    }            /* switch(rank) */
  /* Завершение работы */
    MPI_Finalize(),
    return 0,
   }
```

9.1.3. Обмен данными на системе компьютеров с топологией связи "кольцо"

В этом пункте приведено два примера, демонстрирующих операцию сдвига данных вдоль кольца компьютеров. Эта операция очень часто встречается при решении задач на мультикомпьютерах. Хотя приведенные здесь примеры очень простые, они могут послужить неким шаблоном, который можно применять для решения сложных задач. В первом примере выполняется сдвиг данных соседним ветвям вдоль кольца компьютеров на один шаг, т.е. все ветви параллельной программы передают данные соседним ветвям, например, в сторону увеличения рангов (номеров) компьютеров. Во втором примере реализуется конвейер. Нулевая ветвь инициирует запуск данных вдоль кольца компьютеров, посланные данные последовательно "проходят" по всем компьютерам и возвращаются в нулевой компьютер, реализующий нулевую ветвь.

```
/* Пример 1
 *
 * Сдвиг данных на кольце компьютеров
 */
   #include <mpi h>
   #include <stdio h>
   #define NUM_DIMS 1

int main( int argc, char** argv )
   {
     int        rank, size, i, A, B, dims[NUM_DIMS],
     int        periods[NUM_DIMS], source, dest,
     int        reorder = 0,
     MPI_Comm   comm_cart,
     MPI_Status status,
     MPI_Init(&argc, &argv),
  /* Каждая ветвь узнает общее количество ветвей */
     MPI_Comm_rank(MPI_COMM_WORLD, &rank),
  /* и свой номер  от 0 до (size-1) */
     MPI_Comm_size(MPI_COMM_WORLD, &size),
     A = rank,
     B = -1,
```

```
  /* Обнуляем массив dims и заполняем массив periods для топологии "кольцо" */
     for(i = 0; i < NUM_DIMS; i++) { dims[i] = 0; periods[i] = 1; }
  /* Заполняем массив dims, в котором указываются размеры решетки */
     MPI_Dims_create(size, NUM_DIMS, dims);
  /* Создаем топологию "кольцо" с communicator(ом) comm_cart */
     MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
                                                          &comm_cart);
  /* Каждая ветвь находит своих соседей вдоль кольца, в направлении
   * больших значений рангов */
     MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
  /* Каждая ветвь передает свои данные (значение переменной A) соседней
   * ветви с большим рангом и принимает данные в B от соседней ветви с
   * меньшим рангом. Ветвь с рангом size-1 передает свои данные ветви с
   * рангом 0, а ветвь 0 принимает данные от ветви size-1. */
     MPI_Sendrecv(&A, 1, MPI_INT, dest, 12, &B, 1, MPI_INT, source, 12,
                                                   comm_cart, &status);
  /* Каждая ветвь печатает свой ранг (он же и был послан соседней ветви
   * с большим рангом) и значение переменной B (ранг соседней ветви с
   * меньшим рангом). */
     printf("rank = %d  B=%d\n", rank, B);
  /* Все ветви завершают системные процессы, связанные с топологией
   * comm_cart и завершаю выполнение программы */
     MPI_Comm_free(&comm_cart);
     MPI_Finalize();
     return 0;
   }

/* Пример 2
 *
 * Сдвиг данных на кольце компьютеров. Конвейер.
 *
 * Нулевая ветвь передает данные (значение своего ранга) ветви 1,
 * ветвь 1 передает принятое значение ветви 2, и т.д., ветвь size-1
 * передает, принятое от ветви size-2 значение, ветви 0.
 */
   #include <mpi.h>
   #include <stdio.h>
   #define NUM_DIMS 1

int main(int argc, char** argv)
   {
     int        rank, size, i, A, B, dims[NUM_DIMS];
     int        periods[NUM_DIMS], source, dest;
     int        reorder = 0;
     MPI_Comm   comm_cart;
     MPI_Status status;
     MPI_Init(&argc, &argv);
  /* Каждая ветвь узнает количество ветвей */
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* и свой номер: от 0 до (size-1) */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     A = rank;
     B = -1;
```

```
  /* Обнуляем массив dims и заполняем массив periods для топологии "кольцо" */
     for(i = 0, i < NUM_DIMS, i++) { dims[i] = 0, periods[i] = 1, }
  /* Заполняем массив dims, где указываются размеры (одномерной) решетки */
     MPI_Dims_create(size, NUM_DIMS, dims),
  /* Создаем топологию "кольцо" с communicator(ом) comm_cart */
     MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
                                                          &comm_cart),
  /* Каждая ветвь находит своих соседей вдоль кольца, в направлении
   * больших значений рангов */
     MPI_Cart_shift(comm_cart, 0, 1, &source, &dest),
  /* 0-ветвь инициирует передачу данных (значение своего ранга) вдоль
   * кольца, и принимает это же значение от ветви size-1 */
     if(rank == 0)
       { MPI_Send(&A, 1, MPI_INT, dest, 12, comm_cart),
         MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status),
         printf("rank=%d A=%d B=%d \n", rank, A, B),
       }
  /* Работа всех остальных ветвей  */
     else
       { MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status),
         MPI_Send(&B, 1, MPI_INT, dest, 12, comm_cart),
       }
/* Все ветви завершают системные процессы, связанные с топологией
 * comm_cart и завершаю выполнение программы */
     MPI_Comm_free(&comm_cart),
     MPI_Finalize(),
     return 0,
   }
```

9.1.4. Обмен данными на системе компьютеров с топологией связи "линейка"

Ниже приведен пример, демонстрирующий операцию сдвига данных на линейке компьютеров, которая очень часто встречается при решении задач на мультикомпьютерах. Хотя приведенный здесь пример очень простой, но он может, как и примеры п 9.1.3, послужить неким шаблоном, который используется для решения сложных задач. Все ветви линейки одновременно сдвигают свои данные соседним ветвям вдоль линейки на один шаг в сторону увеличения и затем в сторону уменьшения ранга ветвей. У граничных ветвей соседи имеются только с одной стороны. Поэтому здесь удобно использовать понятие MPI_PROC_NULL процессов.

```
/*
   Сдвиг данных на  линейке процессов с использованием MPI_PROC_NULL процессов
*/
   #include <mpi h>
   #include <stdio h>
   #define NUM_DIMS 1

int main(int argc, char **argv)
   {
     int         rank, size, i, A, B, dims[NUM_DIMS],
     int         periods[NUM_DIMS], new_coords[NUM_DIMS],
     int         sourceb, destb, sourcem, destm,
     int         reorder = 0,
     MPI_Comm    comm_cart;
```

```
   MPI_Status  status,          .
   MPI_Init(&argc, &argv),
/* Каждая ветвь узнает количество ветвей */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank),
/* и свой номер  от 0 до (size-1) */
   MPI_Comm_size(MPI_COMM_WORLD, &size),
/* Обнуляем массив dims и заполняем массив periods для топологии "линейка" */
   for(i = 0, i < NUM_DIMS, i++) { dims[i] = 0, periods[i] = 0, }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
   MPI_Dims_create(size, NUM_DIMS, dims),
/* Создаем топологию "линейка" с communicator(ом) comm_cart */
   MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
                                                        &comm_cart),
/* Отображаем ранги в координаты и выводим их */
   MPI_Cart_coords(comm_cart, rank, NUM_DIMS, new_coords),
   A = new_coords[0],
   B = -1,
/* Каждая ветвь находит своих соседей вдоль линейки, в направлении больших
 * значений номеров компьютеров и в направлении меньших значений номеров
 * Ветви с номером new_coords[0] == 0 не имеют соседей с меньшим номером,
 * поэтому с этого направления эти ветви принимают данные от несуществующих
 * ветвей, т е  от ветвей sourcem = MPI_PROC_NULL, и, соответственно,
 * передают данные в этом направлении ветвям destm = MPI_PROC_NULL
 * Аналогично определяется соседство для ветвей с номером
 * new_coords[0] == dims[0]-1
 */
 if(new_coords[0] == 0)
   {
     sourcem = destm = MPI_PROC_NULL,
   }
 else
   { sourcem = destm = new_coords[0]-1,
   }
 if(new_coords[0] == dims[0]-1)
   {
     destb = sourceb =MPI_PROC_NULL,
   }
 else
   { destb = sourceb = new_coords[0]+1,
   }
 /* Каждая ветвь передает свои данные (значение переменной A) своей
  * соседней ветви с большим номером и принимает данные в B от
  * соседней ветви с меньшим номером  Свой номер и номер,
  * принятый в B выводятся на печать
  */
    MPI_Sendrecv(&A, 1, MPI_INT, destb, 12, &B, 1, MPI_INT, sourcem, 12,
                                                  comm_cart, &status),
    printf("new_coords[0] = %d B = %d\n", new_coords[0], B),
 /* Сдвиг данных в противоположную сторону и вывод соответствующих данных */
    MPI_Sendrecv(&A, 1, MPI_INT, destm, 12, &B, 1, MPI_INT, sourceb, 12,
                                                  comm_cart, &status),
    printf("new_coords[0] = %d B = %d\n", new_coords[0], B),
/* Все ветви завершают системные процессы, связанные с топологией
```

```
   * comm_cart и завершаю выполнение программы */
      MPI_Comm_free(&comm_cart);
      MPI_Finalize();
      return 0;
   }
```

Применение MPI_PROC_NULL процессов значительно упрощает программирование подобных операций, т. к. не нужно отдельно программировать граничные и внутренние ветви решетки.

9.2. Произведение двух матриц

В этом пункте приведено три примера произведения двух матриц, выполняемых на системе с разными топологиями. В первом примере матрицы перемножаются на системе с топологией связи "кольцо", во втором – на системе с топологией связи "двумерная решетка" и, наконец, в третьем – на топологии "трехмерная решетка". Схемы распределения данных по компьютерам приведены в гл. 2 для всех трех примеров.

9.2.1. Произведение двух матриц в топологии "кольцо"

В первом примере предполагается, что обе матрицы разрезаны на части заранее и каждая ветвь считывает свои части обеих матриц с дисковой памяти. Здесь каждая ветвь генерирует свои части матриц. Схемы распределения данных по компьютерам приведены в п. 2.2.1.

```
/* Произведение двух матриц в топологии "кольцо" компьютеров
 */
/* В примере предполагается, что количество строк матрицы A и количество
 * столбцов матрицы B делятся без остатка на количество компьютеров в системе.
 * В данном случае задачу запускаем на восьми компьютерах.
 */
   #include<stdio.h>
   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>
/* Задаем в каждой ветви размеры полос матриц A, B и C. (Здесь предполагается,
 * что размеры ветвей одинаковы во всех ветвях. */
   #define M 320
   #define N 40
/* NUM_DIMS - размер декартовой топологии. "кольцо" - одномерный тор. */
   #define NUM_DIMS 1
   #define EL(x) (sizeof(x) / sizeof(x[0][0]))
/* Задаем полосы исходных матриц. В каждой ветви, в данном случае,
 * они одинаковы */
static double A[N][M], B[M][N], C[N][M];
int main(int argc, char **argv)
   { int        rank, size, i, j, k, i1, j1, d, sour, dest;
     int        dims[NUM_DIMS], periods[NUM_DIMS], new_coords[NUM_DIMS];
     int        reorder = 0;
     MPI_Comm   comm_cart;
     MPI_Status st;
     struct timeval tv1, tv2;      /* Для засечения времени */
     int dt1;
  /* Инициализация библиотеки MPI*/
```

```
   MPI_Init(&argc, &argv),
/* Каждая ветвь узнает количество задач в стартовавшем приложении */
   MPI_Comm_size(MPI_COMM_WORLD, &size),
/* и свой собственный номер  от 0 до (size-1) */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank),
/* Обнуляем массив dims и заполняем массив periods для топологии "кольцо" */
   for(i=0, i<NUM_DIMS, i++) { dims[i] = 0, periods[i] = 1, }
/* Заполняем массив dims, где указываются размеры (одномерной) решетки */
   MPI_Dims_create(size, NUM_DIMS, dims),
/* Создаем топологию "кольцо" с communicator(ом) comm_cart */
   MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
                                                        &comm_cart),
/* Отображаем ранги на координаты компьютеров, с целью оптимизации
 * отображения заданной виртуальной топологии на физическую топологию
 * системы  */
   MPI_Cart_coords(comm_cart, rank, NUM_DIMS, new_coords),
/* Каждая ветвь находит своих соседей вдоль кольца, в направлении
 * меньших значений рангов */
   MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest),
/* Каждая ветвь генерирует полосы исходных матриц A и B, полосы C обнуляет */
   for(i = 0, i < N, i++)
     { for(j = 0, j < M, j++)
         { A[i][j] = 3 141528,
           B[j][i] = 2 812,
           C[i][j] = 0 0,
         }
     }
/* Засекаем начало умножения матриц */
   gettimeofday(&tv1, (struct timezone*)0),
/* Каждая ветвь производит умножение своих полос матриц */
/* Самый внешний цикл for(k) - цикл по компьютерам */
   for(k = 0, k < size, k++)
     {
    /* Каждая ветвь вычисляет координаты (вдоль строки) для результирующих
     * элементов матрицы C, которые зависят от номера цикла k и
     * ранга компьютера  */
       d = ((rank + k)%size)*N,
    /* Каждая ветвь производит умножение своей полосы матрицы A на
     * текущую полосу матрицы B */
       for(j = 0, j < N, j++)
         { for(i1 = 0, j1 = d, j1 < d+N, j1++, i1++)
             { for(i = 0, i < M, i++)
                 C[j][j1] += A[j][i] * B[i][i1],
             }
         }
    /* Умножение полосы строк матрицы A на полосу столбцов матрицы B в каждой
     * ветви завершено */
    /* Каждая ветвь передает своим соседним ветвям с меньшим рангом
     * вертикальные полосы матрицы B  T e  полосы матрицы B сдвигаются вдоль
     * кольца компьютеров */
       MPI_Sendrecv_replace(B, EL(B), MPI_DOUBLE, dest, 12, sour, 12,
                                                  comm_cart, &st),
     }
```

```
  /* Умножение завершено  Каждая ветвь умножила свою полосу строк матрицы A на
   * все полосы столбцов матрицы B
   * Засекаем время и результат печатаем */
     gettimeofday(&tv2, (struct timezone*)0),
     dt1 = (tv2 tv_sec - tv1 tv_sec) * 1000000 + tv2 tv_usec - tv1 tv_usec,
     printf("rank = %d Time = %d\n", rank, dt1),
  /* Для контроля печатаем первые четыре элемента первой строки результата */
     if(rank == 0)
       { for(i = 0, i < 1, i++)
           for(j = 0, j < 4, j++)
               printf("C[i][j] = %f\n", C[i][j]),
       }
  /* Все ветви завершают системные процессы, связанные с топологией comm_cart
   * и завершаю выполнение программы */
     MPI_Comm_free(&comm_cart),
     MPI_Finalize(),
     return(0),
   }
```

9.2.2. Произведение двух матриц в топологии "двумерная решетка"

В этом примере исходные и результирующая матрицы разрезаются в нулевой ветви и затем части матриц рассылаются во все другие ветви. Схемы распределения данных по компьютерам приведены в п. 2.2.2.

```
/* Произведение двух матриц в топологии "двумерная решетка" компьютеров
 */
/* В примере предполагается, что количество строк матрицы A и количество
 * столбцов матрицы B делятся без остатка на количество компьютеров в системе
 * В данном примере задачу запускаем на 4-х компьютерах и на решетке 2x2
 */
   #include<stdio h>
   #include<stdlib h>
   #include<mpi h>
   #include<time h>
   #include<sys/time h>
/* NUM_DIMS - размер декартовой топологии  "двумерная решетка" P0xP1 */
   #define NUM_DIMS 2
   #define P0 2
   #define P1 2
/* Задаем размеры матриц A = MxN, B = NxK и C = MxK (Эти размеры значимы в
 * ветви 0)
 */
   #define M 8
   #define N 8
   #define K 8
   #define A(i,j) A[N*i+j]
   #define B(i,j) B[K*i+j]
   #define C(i,j) C[K*i+j]

/* Подпрограмма, осуществляюшая перемножение матриц */

PMATMAT_2(n, A, B, C, p, comm)
 /* Аргументы A, B, C, n, p значимы в данном случае только в ветви 0 */
```

```
   int *n;                          /* Размеры исходных матриц */
   double *A; *B, *C;      /* Исходные матрицы: A[n[0]][n[1]],
                                                  B[n[1]][n[2]],
                                                  C[n[0]][n[2]]; */
/* Данные */
   int *p;
/* размеров решетки компьютеров. p[0] соответствует n[0], p[1]
 * соответствует n[2] и произведение p[0]*p[1] будет эквивалентно
 * размеру группы comm */
/* Коммуникатор для процессов, участвующих в умножении матрицы на матрицу */
   MPI_Comm comm,

 {
/* Далее все описываемые переменные значимы во всех ветвях, в том числе
 * и ветви 0 */

   double *AA, *BB, *CC;  /* Локальные подматрицы (полосы) */
   int   nn[2];              /* Размеры полос в A и B и подматриц CC в C */
   int coords[2];           /* Декартовы координаты ветвей */
   int rank;                /* Ранг ветвей */
/* Смещения и размер подматриц CC для сборки в корневом процессе (ветви) */
   int  *countc, *dispc, *countb, *dispb;
/* Типы данных и массивы для создаваемых типов */

   MPI_Datatype typeb, typec, types[2];

   int  blen[2];
   int i, j, k;
   int periods[2], remains[2];
   int sizeofdouble, disp[2];

/* Коммуникаторы для 2D решетки, для подрешеток 1D, и копии */
/* коммуникатора comm */
   MPI_Comm comm_2D, comm_1D[2], pcomm;

/* Создаем новый коммуникатор */
   MPI_Comm_dup(comm, &pcomm);
/* Нулевая ветвь передает всем ветвям массивы n[] и p[] */
   MPI_Bcast(n, 3, MPI_INT, 0, pcomm);
   MPI_Bcast(p, 2, MPI_INT, 0, pcomm);

/* Создаем 2D решетку компьютеров размером p[0]*p[1] */
   periods[0] = 0;
   periods[1] = 0;
   MPI_Cart_create(pcomm, 2, p, periods, 0, &comm_2D);
/* Находим ранги и декартовы координаты ветвей в этой решетке */
   MPI_Comm_rank(comm_2D, &rank);
   MPI_Cart_coords(comm_2D, rank, 2, coords);

/* Нахождение коммуникаторов для подрешеток 1D для рассылки полос
 * матриц A и B */
   for(i = 0; i < 2; i++)
     { for(j = 0; j < 2; j++)
```

```
        remains[j] = (i == j),
        MPI_Cart_sub(comm_2D, remains, &comm_1D[i]),
     }
/* Во всех ветвях задаем подматрицы (полосы) */
/* Здесь предполагается, что деление без остатка */
   nn[0] = n[0]/p[0],
   nn[1] = n[2]/p[1],

 #define AA(i,j) AA[n[1]*i+j]
 #define BB(i,j) BB[nn[1]*i+j]
 #define CC(i,j) CC[nn[1]*i+j]

   AA = (double *)malloc(nn[0] * n[1] * sizeof(double)),
   BB = (double *)malloc(n[1] * nn[1] * sizeof(double)),
   CC = (double *)malloc(nn[0] * nn[1] * sizeof(double)),

/* Работа нулевой ветви */
   if(rank == 0)
     {
    /* Задание типа данных для вертикальной полосы в  B
     * Этот тип создать необходимо, т к  в языке C массив в памяти
     * располагается по строкам  Для массива A такой тип создавать
     * нет необходимости, т к  там передаются горизонтальные полосы,
     * а они в памяти расположены непрерывно  */

       MPI_Type_vector(n[1], nn[1], n[2], MPI_DOUBLE, &types[0]),
    /* и корректируем диапазон размера полосы */
       MPI_Type_extent(MPI_DOUBLE, &sizeofdouble),
       blen[0]  = 1,
       blen[1]  = 1,
       disp[0]  = 0,
       disp[1]  = sizeofdouble * nn[1],
       types[1] = MPI_UB,
       MPI_Type_struct(2, blen, disp, types, &typeb),
       MPI_Type_commit(&typeb),

    /* Вычисление размера подматрицы BB и смещений каждой
     * подматрицы в матрице B  Подматрицы BB упорядочены в B
     * в соответствии с порядком номеров компьютеров в решетке,
     * т к  массивы расположены в памяти по строкам, то подматрицы
     * BB в памяти (в B) должны располагаться в следующей
     * последовательности   BB0, BB1,`     */
       dispb =  (int *)malloc(p[1] * sizeof(int)),
       countb = (int *)malloc(p[1] * sizeof(int)),
       for(j = 0, j < p[1], j++)
         { dispb[j] = j,
           countb[j] = 1,
         }

   /* Задание типа данных для подматрицы CC в C */
       MPI_Type_vector(nn[0], nn[1], n[2], MPI_DOUBLE, &types[0]),
   /* и корректируем размер диапазона */
       MPI_Type_struct(2, blen, disp, types, &typec),
```

```
      MPI_Type_commit(&typec);
    /* Вычисление размера подматрицы CC и смещений каждой
     * подматрицы в матрице C. Подматрицы CC упорядочены в C
     * в соответствии с порядком номеров компьютеров в решетке,
     * т.к. массивы расположены в памяти по строкам, то подматрицы
     * CC в памяти (в C) должны располагаться в следующей
     * последовательности: CC0, CC1, CC2, CC3, CC4, CC5, CC6, CC7. */
      dispc =  (int *)malloc(p[0] * p[1] * sizeof(int));
      countc = (int *)malloc(p[0] * p[1] * sizeof(int));
      for(i = 0; i < p[0]; i++)
        { for(j = 0; j < p[1]; j++)
            { dispc[i*p[1]+j] = (i*p[1]*nn[0] + j);
              countc[i*p[1]+j] = 1;
            }
        }
    }        /* Нулевая ветвь завершает подготовительную работу */

/* Вычисления (этапы указаны на рис.2.4 в гл.~2) */
/* 1. Нулевая ветвь передает (scatter) горизонтальные полосы матрицы A
 *    по x координате */

  if(coords[1] == 0)
    { MPI_Scatter(A, nn[0]*n[1], MPI_DOUBLE, AA, nn[0]*n[1], MPI_DOUBLE, 0,
                                                          comm_1D[0]);
    }

    MPI_Barrier(MPI_COMM_WORLD);

/* 2. Нулевая ветвь передает (scatter) горизонтальные полосы матрицы B
 *    по y координате */

  if(coords[0] == 0)
    { MPI_Scatterv(B, countb, dispb, typeb, BB, n[1]*nn[1], MPI_DOUBLE, 0,
                                                          comm_1D[1]);
    }

/* 3. Передача подматриц  AA в измерении y */
    MPI_Bcast(AA, nn[0]*n[1], MPI_DOUBLE, 0, comm_1D[1]);
/* 4. Передача подматриц  BB в измерении x */
  MPI_Bcast(BB, n[1]*nn[1], MPI_DOUBLE, 0, comm_1D[0]);

/* 5. Вычисление подматриц CC в каждой ветви */

  for(i = 0; i < nn[0]; i++)
    { for(j = 0; j < nn[1]; j++)
        { CC(i,j) = 0.0;
          for(k = 0; k < n[1]; k++)
            { CC(i,j) = CC(i,j) + AA(i,k) * BB(k,j);
            }
        }
    }

/* 6. Сбор всех подматриц CC в ветви 0 */
```

```
    MPI_Gatherv(CC, nn[0]*nn[1], MPI_DOUBLE, C, countc, dispc, typec, 0,
                                                              comm_2D);

 /* Освобождение памяти всеми ветвями и завершение подпрограммы */
    free(AA);
    free(BB);
    free(CC);

    MPI_Comm_free(&pcomm);
    MPI_Comm_free(&comm_2D);
    for(i = 0; i < 2; i++)
      { MPI_Comm_free(&comm_1D[i]);
      }
    if(rank == 0)
      { free(countc);
        free(dispc);
        MPI_Type_free(&typeb);
        MPI_Type_free(&typec);
        MPI_Type_free(&types[0]);
      }

    return 0;
  }

 /* Главная программа */

int main(int argc, char **argv)
  {
    int        size, MyP, n[3], p[2], i, j, k;
    int        dims[NUM_DIMS], periods[NUM_DIMS];
    double     *A, *B, *C;
    int        reorder = 0;
    struct timeval tv1, tv2;        /* Для засечения времени */
    int dt1;
    MPI_Comm   comm;
 /* Инициализация библиотеки MPI */
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает количество задач в стартовавшем приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* и свой собственный номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);
 /* Обнуляем массив dims и заполняем массив periods для топологии
  * "двумерная решетка" */
    for(i = 0; i < NUM_DIMS; i++) { dims[i] = 0; periods[i] = 0; }
 /* Заполняем массив dims, где указываются размеры двумерной решетки */
    MPI_Dims_create(size, NUM_DIMS, dims);
 /* Создаем топологию "двумерная решетка" с communicator(ом) comm */
    MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder, &comm)
 /* В первой ветви выделяем в памяти место для исходных матриц */
    if(MyP == 0)
      {
     /* Задаем размеры матриц и размеры двумерной решетки компьютеров */
        n[0] = M;
```

```
      n[1] = N,
      n[2] = K,
      p[0] = P0,
      p[1] = P1,

      A = (double *)malloc(M * N * sizeof(double)),
      B = (double *)malloc(N * K * sizeof(double)),
      C = (double *)malloc(M * K * sizeof(double)),

    /* Генерируем в первой ветви исходные матрицы A и B, матрицу C обнуляем */
      for(i = 0, i < M, i++)
        for(j = 0, j < N, j++)
          A(i,j) = i+1,
      for(j = 0, j < N, j++)
        for(k = 0, k < K, k++)
          B(j,k) = 21+j,
      for(i = 0, i < M, i++)
        for(k = 0, k < K, k++)
          C(i,k) = 0 0,
    }           /* Подготовка матриц ветвью 0 завершена */
/* Засекаем начало умножения матриц во всех ветвях */
   gettimeofday(&tv1, (struct timezone*)0),
/* Все ветви вызывают функцию перемножения матриц */
   PMATMAT_2(n, A, B, C, p, comm),
/* Умножение завершено  Каждая ветвь умножила свою полосу строк матрицы A на
 * полосу столбцов матрицы B  Результат находится в нулевой ветви
 * Засекаем время и результат печатаем */
   gettimeofday(&tv2, (struct timezone*)0),
   dt1 = (tv2 tv_sec - tv1 tv_sec) * 1000000 + tv2 tv_usec - tv1 tv_usec,
   printf("MyP = %d Time = %d\n", MyP, dt1),
/* Для контроля 0-я ветвь печатает результат */

   if(MyP == 0)
     { for(i = 0, i < M, i++)
         { for(j = 0, j < K, j++)
             printf(" %3 1f",C(i,j)),
           printf("\n"),
         }
     }

/* Все ветви завершают системные процессы, связанные с топологией comm
 * и завершают выполнение программы */

   if(MyP == 0)
     { free(A),
       free(B),
       free(C),
     }

   MPI_Comm_free(&comm),
   MPI_Finalize(),
   return(0),
```

9.2.3. Произведение двух матриц в топологии "трехмерная решетка"

В этом примере исходные и результирующая матрицы разрезаются в нулевой ветви и затем части матриц рассылаются во все другие ветви. Схемы распределения данных по компьютерам приведены в п. 2.2.3.

```
/* Произведение двух матриц в топологии "трехмерная решетка" компьютеров
 */
/* В примере предполагается, что количество строк и столбцов матрицы A
 * и количество строк и столбцов матрицы B делятся без остатка на количество
 * компьютеров в системе
 * В данном примере задачу запускаем на 8-х компьютерах и на решетке 2x2x2
 */
   #include<stdio h>
   #include<stdlib h>
   #include<mpi h>
   #include<time h>
   #include<sys/time h>
/* NUM_DIMS - размер декартовой топологии  "трехмерная решетка" P0xP1xP2 */
   #define NUM_DIMS 3
   #define P0 2
   #define P1 2
   #define P2 2
/* Задаем размеры матриц A = MxN, B = NxK и C = MxK (Эти размеры значимы в
   ветви 0) */
   #define M 8
   #define N 8
   #define K 8

   #define A(i,j) A[N*i+j]
   #define B(i,j) B[K*i+j]
   #define C(i,j) C[K*i+j]

  #define C1(i,j) C1[K*i+j]

/* Подпрограмма, осуществляющая перемножение матриц */

PMATMAT_3(n, A, B, C, p, comm)
 /* Аргументы A, B, C, n, p значимы только в ветви 0 */
    int *n,                  /* Размеры исходных матриц */
    double *A, *B, *C,        /* Исходные матрицы  A[n[0]][n[1]],
                                                    B[n[1]][n[2]],
                                                    C[n[0]][n[2]], */
 /* Данные */
    int *p,
 /* размеров решетки процессов  p[i] соответствует n[i]
  * и произведение 3-х измерений эквивалентно размеру
  * группы  comm */
 /* Коммуникатор для процессов, умножающих матрицу на матрицу */
    MPI_Comm comm,

  {
 /* Далее все описываемые переменные значимы во всех ветвях, в том числе и
    ветви 0 */
```

```
   double *AA, *BB, *CC, *CC1,    /* Локальные подматрицы (полосы) */
   int nn[3],                  /* Размеры полос в A и B и подматриц CC в C */

/* Коммуникаторы для 3D решетки, для подрешеток, и копии comm */

   int coords[3],           /* Декартовы координаты */
   int rank,                /* Ранг процесса (ветви) */
/* Смешения и размеры для операций scatter/gather */
   int *dispa, *dispb, *dispc, *counta, *countb, *countc,
/* Переменные и массивы для создаваемых типов данных */
   MPI_Datatype typea, typeb, typec, types[2],

   int periods[3], remains[3],
   MPI_Comm comm_3D, comm_2D[3], comm_1D[3], pcomm,

   int i, j, k, sizeofdouble, blen[2], disp[2],

   MPI_Comm_dup(comm, &pcomm),                /* Копия коммуникатора comm */

/* Передача параметров n[3] и p[3] */
   MPI_Bcast(n, 3, MPI_INT, 0, pcomm),
   MPI_Bcast(p, 3, MPI_INT, 0, pcomm),
/* Создание 3D решетки процессов */
   for(i = 0, i < 3, i++)
     periods[i] = 0,
   MPI_Cart_create(pcomm, 3, p, periods, 0, &comm_3D),

/* Нахождение ранга и декартовых координат */
   MPI_Comm_rank(comm_3D, &rank),
   MPI_Cart_coords(comm_3D, rank, 3, coords),

/* Получение коммуникаторов для подрешеток размерности 2D */
   for(i = 0, i < 3, i++)
     { for(j = 0, j < 3, j++)
         remains[j] = (i != j),
       MPI_Cart_sub(comm_3D, remains, &comm_2D[i]),
     }
/* Получение коммуникаторов для подрешеток размерности 1D */
   for(i = 0, i < 3, i++)
     { for(j = 0, j < 3, j++)
         remains[j] = (i == j),
       MPI_Cart_sub(comm_3D, remains, &comm_1D[i]),
     }
/* Выделение памяти для подматриц */
   for(i = 0, i < 3, i++)
     nn[i] = n[i]/p[i],

  #define AA(i,j) AA[nn[1]*i+j]
  #define BB(i,j) BB[nn[2]*i+j]
  #define CC(i,j) CC[nn[2]*i+j]

   AA = (double *)malloc(nn[0] * nn[1] * sizeof(double)),
   BB = (double *)malloc(nn[1] * nn[2] * sizeof(double)),
```

```
   CC = (double *)malloc(nn[0] * nn[2] * sizeof(double)),

/* Работа нулевой ветви */
   if(rank == 0)
     {
    /* Создание типа данных для подматриц в A */
      MPI_Type_vector(nn[0], nn[1], n[1], MPI_DOUBLE, &types[0]),
    /* и корректировка размеров подстрок */
      MPI_Type_extent(MPI_DOUBLE, &sizeofdouble),
      blen[0] = 1,
      blen[1] = 1,
      disp[0] = 0,
      disp[1] = sizeofdouble * nn[1],
      types[1] = MPI_UB,
      MPI_Type_struct(2, blen, disp, types, &typea),
      MPI_Type_commit(&typea),
    /* Нахождение смещений и размеров подматриц в A
     * Подматрицы в A упорядочены аналогично упорядочению процессов
     * в решетке  */
      dispa = (int *)malloc(p[0]*p[1] * sizeof(int)),
      counta = (int *)malloc(p[0]*p[1] * sizeof(int)),
      for(j = 0, j < p[0], j++)
        for(i = 0, i < p[1], i++)
          { dispa[j*p[1]+i] = (j*p[1]*nn[0] +i),
            counta[j*p[1]+i] = 1,
          }
    /* Тожé самое для массива B */
      MPI_Type_vector(nn[1], nn[2], n[2], MPI_DOUBLE, &types[0]),
      disp[1] = sizeofdouble*nn[2],
      MPI_Type_struct(2,blen,disp,types,&typeb),
      MPI_Type_commit(&typeb),
      dispb = (int *)malloc(p[1]*p[2] * sizeof(int)),
      countb = (int *)malloc(p[1]*p[2] * sizeof(int)),
      for(j = 0, j < p[1], j++)
         for(i = 0, i < p[2], i++)
          { dispb[j*p[2]+i] = (j*p[2]*nn[1] +i),
            countb[j*p[2]+i] = 1,
          }
    /* Тоже самое для массива C */
      MPI_Type_vector(nn[0], nn[2], n[2], MPI_DOUBLE, &types[0]),
      disp[1] = sizeofdouble*nn[2],
      MPI_Type_struct(2, blen, disp, types, &typec),
      MPI_Type_commit(&typec),
      dispc = (int *)malloc(p[0]*p[2] * sizeof(int)),
      countc = (int *)malloc(p[0]*p[2] * sizeof(int)),
      for(j = 0, j < p[0], j++)
        for(i = 0, i < p[2], i++)
          { dispc[j*p[2]+i] = (j*p[2]*nn[0] +i),
            countc[j*p[2]+i] = 1,
          }
     }      /* Нулевая ветвь завершает подготовительную работу */
```

```
/* Вычисления  Этапы представлены на рис  2 6 в гл ~2  */
/* 1  Нулевая ветвь передает (scatter) горизонтальные полосы матрицы A
 *    по x координате */

 if(coords[2] == 0)
   MPI_Scatterv(A, counta, dispa, typea, AA, nn[0]*nn[1], MPI_DOUBLE, 0,
                                                          comm_2D[2]),

   MPI_Barrier(MPI_COMM_WORLD),

/* 2  Нулевая ветвь передает (scatter) вертикальные полосы матрицы B
 *    по y координате */
   if(coords[0] == 0)
   MPI_Scatterv(B, countb, dispb, typeb, BB, nn[1]*nn[2], MPI_DOUBLE, 0,
                                                          comm_2D[0]),

   MPI_Barrier(MPI_COMM_WORLD),

/* 3  Рассылка (broadcast) подматриц AA в измерении z */
   MPI_Bcast(AA, nn[0]*nn[1], MPI_DOUBLE, 0, comm_1D[2]),

/* 4  Рассылка (broadcast) подматриц BB в измерении x */
   MPI_Bcast(BB, nn[1]*nn[2], MPI_DOUBLE, 0, comm_1D[0]),

/* 5  Вычисление всеми ветвями подматриц CC */
   for(i = 0, i < nn[0], i++)
     for(j = 0, j < nn[2], j++)
       { CC(i,j) = 0,
         for(k = 0, k < nn[1], k++)
           CC(i,j) = CC(i,j) + AA(i,k) * BB(k,j),
       }

/* 6  Сбор подматриц CC в измерении y */
/* Редукция подматриц CC вначале осуществляется "в плоскости" */
   CC1 = (double *)malloc(nn[0] * nn[2] * sizeof(double)),
   MPI_Reduce(CC, CC1, nn[0]*nn[2], MPI_DOUBLE, MPI_SUM, 0, comm_1D[1]),

/* 7  Сбор результатов из плоскости (x,0,z) в узел 0 */
   if(coords[1] == 0)
     MPI_Gatherv(CC1, nn[0]*nn[2], MPI_DOUBLE, C, countc, dispc, typec, 0,
                                                          comm_2D[1]),

   MPI_Barrier(MPI_COMM_WORLD),

/*
   if(rank == 0)
     { for(i = 0, i < M, i++)
        { for(k = 0, k < K, k++)
           printf("%3 1f  ", C(i,k)),
         printf("\n"),
        }
     }
*/
```

```
 /* Освобождение памяти всеми ветвями и завершение подпрограммы */
    free(AA),
    free(BB),
    free(CC),
    free(CC1),
    MPI_Comm_free(&pcomm),
    MPI_Comm_free(&comm_3D),
    for(i = 0, i < 3, i++)
      { MPI_Comm_free(&comm_2D[i]),
        MPI_Comm_free(&comm_1D[i]),
      }
    if(rank == 0)
      { free(counta),
        free(countb),
        free(countc),
        free(dispa),
        free(dispb),
        free(dispc),
        MPI_Type_free(&typea),
        MPI_Type_free(&typeb),
        MPI_Type_free(&typec),
        MPI_Type_free(&types[0]),
      }
    return 0,
  }

  /* Главная программа */
int main(int argc, char **argv)
  {
    double *A, *B, *C,
    int         size, MyP, n[3], p[3], i, j, k,
    int         dims[NUM_DIMS], periods[NUM_DIMS],
    int         reorder = 0,

    struct timeval tv1, tv2,        /* Для засечения времени */
    int dt1,
    MPI_Comm    comm,
   /* Инициализация библиотеки MPI */
    MPI_Init(&argc, &argv),
 /* Каждая ветвь узнает количество задач в стартовавшем приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size),
 /* и свой собственный номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP),
 /* Обнуляем массив dims и заполняем массив periods для топологии
  * "трехмерная решетка" */
    for(i = 0, i < NUM_DIMS, i++) { dims[i] = 0, periods[i] = 0, }
 /* Заполняем массив dims, где указываются размеры трехмерной решетки */
    MPI_Dims_create(size, NUM_DIMS, dims),
 /* Создаем топологию "трехмерная решетка" с communicator(ом) comm */
    MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder, &comm),
 /* Задаем размеры матриц и размеры трехмерной решетки компьютеров */
    n[0] = M,
```

```
   n[1] = N;
   n[2] = K;
   p[0] = P0;
   p[1] = P1;
   p[2] = P2;
/* В первой ветви выделяем в памяти место для исходных матриц */
   if(MyP == 0)
     { A = (double *)malloc(n[0] * n[1] * sizeof(double));
       B = (double *)malloc(n[1] * n[2] * sizeof(double));
       C = (double *)malloc(n[0] * n[2] * sizeof(double));

    /* Ветвь 0 генерирует исходные матрицы A и B, матрицу C обнуляет */
       for(i = 0; i < M; i++)
         for(j = 0; j < N; j++)
           A(i,j) = i+1;
       for(j = 0; j < N; j++)
         for(k = 0; k < K; k++)
           B(j,k) = 21+j;
       for(i = 0; i < M; i++)
         for(j = 0; j < K; j++)
           C(i,j) = 0.0;
     }              /* Подготовка матриц ветвью 0 завершена */
/* Каждая ветвь засекает начало умножения матриц */
   gettimeofday(&tv1, (struct timezone*)0);
/* Все ветви вызывают функцию перемножения матриц */

   PMATMAT_3(n, A, B, C, p, comm);

/* Умножение завершено. Каждая ветвь умножила свою полосу строк матрицы A на
 * полосу столбцов матрицы B. Результат находится в нулевой ветви.
 * Засекаем время и результат печатаем */

   gettimeofday(&tv2, (struct timezone*)0);

   dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
   printf("MyP = %d Time = %d\n", MyP, dt1);
/* Для контроля ветвь 0 печатает результат */

   if(MyP == 0)
     { for(i = 0; i < M; i++)
        { for(j = 0; j < K; j++)
            printf("%3.1f  ", C(i,j));
          printf("\n");
        }
     }

/* Все ветви завершают системные процессы, связанные с топологией comm
 * и завершаю выполнение программы */
   if(MyP == 0)
     { free(A);
       free(B);
       free(C);
     }
```

```
    MPI_Comm_free(&comm);
    MPI_Finalize();
    return(0);
  }
```

9.3. Задача Дирихле. Явная разностная схема для уравнения Пуассона

В этом пункте приведен пример решения дифференциальных уравнений в частных производных, показывающий, с одной стороны, параллельный алгоритм решения этой задачи, а с другой, поясняющий функции обменов, представленные в гл. 5. Здесь приведен фрагмент алгоритма решения указанной задачи, но его вполне достаточно, чтобы создавать параллельные алгоритмы для конкретных задач.

Для решения задачи применяется алгоритм, использующий 1D разрезание данных (показанное на рис. 2.8), распределение данных по компьютерам с перекрытием в один столбец (см. рис. 2.9) и реализующих схему обменов между ветвями параллельной программы (см. рис. 2.10). В подпунктах п. 9.3 в основном демонстрируются реализации указанной схемы обменов данными разными обменными функциями. Вид уравнения представлен в гл. 2.

Алгоритм реализуется на виртуальной топологии "линейка". В подпунктах приводятся только фрагменты программ, реализующих схемы обменов. Главный цикл приведем здесь, а в других примерах будем обозначать его, как "вычислительная часть".

С целью удобства везде предполагается, что двумерные массивы данных располагаются в памяти компьютеров по столбцам, т.е. элементы столбца располагаются в памяти непрерывно. Поэтому вместо передачи или приема столбцов будут передаваться или приниматься строки. Хотя в гл. 2 на рис. 2.10 приведена схема обменов для столбцов, суть алгоритма это не меняет.

```
/* Главный цикл задачи Дирихле. Фрагмент программы. */
   double A[m+2][n+2], B[m][n];
...
   while(! converged) {
/* Выполнение четырехточечного шаблона */
   for(j = 1; j <= m, j++)
     for(i = 1; i <= n; i++)
       B[j-1][i-1] = 0.25*(A[j][i-1] + A[j][i+1] + A[j-1][i] + A[j+1][i]);
/* Копирование результата обратно в массив A */
   for(j = 1; j <= m; j++)
     for(i = 1; i <= n; i++)
       A[j][i] = B[j][i-1];
...
 }
/* Каждая ветвь узнает количество ветвей*/
   MPI_Comm_size(comm, p);
/* и свой номер: от 0 до (size-1) */
   MPI_Comm_rank(comm, myrank);
```

Этот фрагмент программы описывает главный цикл итерационного процесса решения, где на каждой итерации значение в окрестности точки заменяется средним значением сумм значений ее соседних точек на предыдущем шаге итерации. Граничные значения не изменяются. В массиве B вычисляются значения следующей итерации, а в массиве A находятся значения предыдущей итерации. Здесь приведен внутренний цикл, где выполнено большинство вычислений. В первой и последней строке, а так же в первом и последнем столбце двумерного массива A записаны граничные значения.

9.3.1. Задача Дирихле. Реализация обменов функциями MPI_Send, MPI_Secv

Для получения безопасной версии этой параллельной программы при использовании блокированных функций обмена данными нужно построить соответствующую модель связи между процессорами. Все процессоры разбиваются на два подмножества: с четными и нечетными номерами. Модель обмена состоит в следующем. Напомним, что топология связи между компьютерами – линейка. Обмен выполняется в два полушага. На первом полушаге нечетные процессоры передают данные четным процессорам с номером, на единицу меньшим (в сторону уменьшения номеров в линейке), а четные принимают от них, т.е. от процессоров с бóльшими номерами. На втором полушаге обмен выполняется наоборот. Напомним, что массив данных в этом случае располагается по столбцам. Далее будут приведены более простые способы обменов данными.

```
/* Обмен данными в задаче Дирихле. Первая версия параллельной программы.
 * Топология - "линейка".
 */
/* Вычислительная часть */
. . . . . .
/* Обмен граничными столбцами между ветвями программы */
   if(myrank%2 == 1)
     {
/* Работа нечетных ветвей */
/* Передается 0-я строка массива B к ветви myrank-1 */
       MPI_Send(B, n, MPI_FLOAT, myrank-1, tag, comm);
/* Принимается m-1-я строка массива B от ветви myrank-1 и записывается
 * в 0-ю строку массива A, начиная с первого элемента.
 */
       MPI_Recv(&A[0][1], n, MPI_FLOAT, myrank-1, tag, comm, status);
       if(myrank < p-1)            /* Здесь p = количеству ветвей */
/* Передается m-1-я строка массива B к ветви myrank+1. */
 * У p-й ветви нет соседней ветви с рангом myrank+1 , поэтому она
 * исключена из этого обмена
 */
       MPI_Send(&B[m-1][0], n, MPI_FLOAT, myrank+1, tag, comm);
/* Принимается 0-ая строка массива B от ветви myrank+1 и
 * записывается в m+1-ю строку массива A, начиная с первого элемента*/
       MPI_Recv(A[m+1][1], n, MPI_FLOAT, myrank+1, tag, comm, status);
     }
/* Работа четных ветвей */
   else if(myrank < p-1)
     {
/* Принимается 0-ая строка массива B от ветви myrank+1 и
 * записывается в m+1-ю строку массива A, начиная с первого элемента.
 * У p-й ветви нет соседней ветви с рангом myrank+1 , поэтому она
 * исключена из этого обмена
 */
       MPI_Recv(A[m+1][1], n, MPI_FLOAT, myrank+1, tag, comm, status);
/* Передается m-1-я строка массива B к ветви myrank+1 */
       MPI_Send(B[m-1][0], n, MPI_FLOAT, myrank+1, tag, comm);
     }
   else if(myrank > 0)
     {
/* Принимается m-1-я строка массива B от ветви myrank-1 и
 * записывается в 0-ю строку массива A.
```

```
 * У 0-й ветви нет соседней ветви с рангом myrank-1 , поэтому она
 * исключена из этого обмена
 */
      MPI_Recv(A[0][1], n, MPI_FLOAT, myrank-1, tag, comm, status),
/* Передается 0-ой столбец массива B к ветви myrank-1 */
      MPI_Send(B, n, MPI_FLOAT, myrank-1, tag, comm),
    }
```

9.3.2. Задача Дирихле. Реализация обменов функцией MPI_Sendrecv

```
/* Обмен данными в задаче Дирихле  Вторая версия параллельной
 * программы  Топология - "линейка"
 */
/* Вычислительная часть */

/* Обмен граничными столбцами между ветвями программы */
   if(myrank > 0)
     {
/* Передается 0-я строка массива B к ветви myrank-1
 * и принимается m-1-я строка массива B от ветви myrank-1 и
 * записывается в 0-ю строку массива A
 * У 0-й ветви нет соседней ветви с рангом myrank-1, поэтому она
 * исключена из этого обмена
 */
      MPI_Sendrecv(B, n, MPI_FLOAT, myrank-1, tag, A[0][1], n, MPI_FLOAT,
                                          myrank-1, tag, comm, status),
    }
   if(myrank < p-1)              /* Здесь p = количеству процессоров */
     {
/* Передается m-1-я строка массива B к ветви myrank+1
 * и принимается 0-я строка массива B от ветви myrank+1 и
 * записывается в m+1-ю строку массива A */
 * У p-й ветви нет соседней ветви с рангом myrank+1 , поэтому она
 * исключена из этого обмена
 */
      MPI_Sendrecv(B[m-1][0], n, MPI_FLOAT, myrank+1, tag, A[m+1][1], n,
                                 MPI_FLOAT, myrank+1, tag, comm, status),
    }
```

Можно заметить, что программа стала значительно короче по сравнению с первым вариантом.

9.3.3. Задача Дирихле. Реализация обменов с использованием MPI_PROC_NULL процессов

```
/* Обмен данными в задаче Дирихле  Третья версия параллельной
 * программы с использованием  MPI_PROC_NULL процессов
 * Топология - "линейка"
 */
/* Вычислительная часть */

/* Каждая ветвь находит своих соседей */
   if(myrank == 0)
```

```
    left = MPI_PROC_NULL;
  else
    left = myrank-1;
  if(myrank == p-1)
    right = MPI_PROC_NULL;
  else
    right = myrank+1;
/* Обмен граничными столбцами между ветвями программы */
/* Передается 0-я строка массива B к ветви myrank-1
 * и принимается m-1-я строка массива B от ветви myrank-1 и
 * записывается в 0-ю строку массива A
 */
  MPI_Sendrecv(B, n, MPI_FLOAT, left, tag, A[0][1], n, MPI_FLOAT,
                                          left, tag, comm, status);
/* Передается m-1-я строка массива B к ветви myrank+1
 * и принимается 0-я строка массива B от ветви myrank+1 и
 * записывается в m+1-й строку массива A */
 */
  MPI_Sendrecv(B[m-1][0], n, MPI_FLOAT, right, tag, A[m+1][1], n,
                              MPI_FLOAT, right, tag, comm, status);
...
```

Объем программы сильно не изменяется, но она становится проще для понимания.

9.3.4. Задача Дирихле. Реализация обменов неблокированными функциями MPI_Isend и MPI_Irecv

Здесь поясняется использование неблокированных операций. Чтобы достичь максимального перекрытия между вычислением и обменом данными, связь должна быть начата как можно скорее и закончена, когда возможно. То есть передача данных должна быть инициирована, как только данные, которые будут посланы, доступны. Прием данных должен быть инициирован, как только буфер приема может повторно использоваться. Передача должна быть закончена непосредственно перед тем, как посылающийся буфер должен повторно использоваться. И прием должен быть закончен непосредственно перед тем, как данные в буфере приема должны использоваться. Иногда перекрытие может быть увеличено путем переупорядочивания вычислений.

```
/* Обмен данными в задаче Дирихле. Четвертая версия параллельной
 * программы с использованием неблокированных функций
 * Топология - "линейка".
 */
...
/* Каждая ветвь находит своих соседей */
  if(myrank == 0)
    left = MPI_PROC_NULL;
  else
    left = myrank-1;
  if(myrank == p-1)
    right = MPI_PROC_NULL;
  else
    right = myrank+1;
...
/* Вычисление граничных строк массива B */
  for(i = 1; i <= n; i++)
```

```
    { B[0][i-1]  =  0 25 * (A[0][i] + A[2][i] + A[1][i+1] + A[1][i-1]),
      B[m-1][i-1] = 0 25 * (A[m][i-1] + A[m][i+1] + A[m-1][i] + A[m+1][i]),
    }
/* Старт неблокированных функций для обмена граничными строками между
 * параллельными ветвями */
/* Передается 0-я строка массива B к ветви myrank-1
 */
   MPI_ISend(B, n, MPI_FLOAT, left, tag, comm, req[0]),
/* Передается m-1-я строка массива B к ветви myrank+1 */
   MPI_ISend(B[m-1][0], n, MPI_FLOAT, right, tag, comm, req[1]),
/* Принимается m-1-я строка массива B от ветви myrank-1 и
 * записывается в 0-ю строку массива A */
   MPI_IRecv(A[0][1], n, MPI_FLOAT, left, tag, comm, req[2]),
/* Принимается 0-я строка массива B от процессора myrank+1 и
 * записывается в m+1-ю строку массива A */
   MPI_IRecv(A[m+1][1], n, MPI_FLOAT, right, tag, comm, req[3]),
/* Выполнение четырехточечного шаблона для внутренних строк
 * массива B
 */
   for(j = 2, j <= m-1, j++)
     for(i = 1, i <= n, i++)
       B[j-1][i-1] = 0 25 * (A[j][i-1] + A[j][i+1] + A[i][j-1] + A[i][j+1]),
/* Копирование результата обратно в массив A */
   for(j = 1, j <= m, j++)
     for(i = 1, i <= n, i++)
       A[j][i] = B[j-1][i-1],
/* Завершение операций обмена граничными строками */
   for(i = 0,i <= 3,i++)
     MPI_Wait(req[i], status[0][i]),
```

Эта программа стала несколько длиннее предыдущих, но в ней передача соседям вычисленных границ перекрывается с вычислениями. Вначале все процессы вычисляют крайний левый и крайний правый столбцы массива B и после этого сразу запускают процессы передачи их соседям. Пока выполняется передача процессы продолжают вычислять остальные, внутренние, столбцы массива B. Таким образом, передача и вычисления осуществляются с перекрытием.

9.3.5. Задача Дирихле. Реализация обменов неблокированными функциями MPI_Isend и MPI_Irecv и функцией завершения MPI_Waitall

Эта задача аналогична предыдущей, но здесь используется функция завершения обменов MPI_Waitall.

```
/* Обмен данными в задаче Дирихле  Пятая версия параллельной
 * программы с использованием неблокированных функций и функции
 * завершения MPI_Waitall  Топология - "линейка"
 */

/* Каждая ветвь находит своих соседей */
   if(myrank == 0)
     left = MPI_PROC_NULL,
   else
     left = myrank-1,
```

```
   if(myrank == p-1)
     right = MPI_PROC_NULL;
   else
     right = myrank+1;
...
/* Вычисление граничных строк массива B */
   for(i = 1, i <= n; i++)
     { B[0][i-1]  =  0.25 * (A[0][i] + A[2][i] + A[1][i+1] + A[1][i-1]);
       B[m-1][i-1] = 0.25 * (A[m][i-1] + A[m][i+1] + A[m-1][i] + A[m+1][i]);
     }
/* Старт неблокированных функций для обмена граничными строками между
 * параллельными ветвями */
/* Передается 0-я строка массива B к ветви myrank-1
 */
   MPI_ISend(B, n, MPI_FLOAT, left, tag, comm, req[0]);
/* Передается m-1-я строка массива B к ветви myrank+1 */
   MPI_ISend(B[m-1][0], n, MPI_FLOAT, right, tag, comm, req[1]);
/* Принимается m-1-я строка массива B от ветви myrank-1 и
 * записывается в 0-ю строку массива A */
   MPI_IRecv(A[0][1], n, MPI_FLOAT, left, tag, comm, req[2]);
/* Принимается 0-я строка массива B от процессора myrank+1 и
 * записывается в m+1-ю строку массива A */
   MPI_IRecv(A[m+1][1],n, MPI_FLOAT, right, tag, comm, req[3]);
/* Выполнение четырехточечного шаблона для внутренних строк массива B
 */
   for(j = 2; j <= m-1; j++)
     for(i = 1; i <= n; i++)
       B[j-1][i-1] = 0.25 * (A[j][i-1] + A[j][i+1] + A[i][j-1] + A[i][j+1]);
/* Копирование результата обратно в массив A */
   for(j = 1; j <= m; j++)
     for(i = 1; i <= n; i++)
       A[j][i] = B[j-1][i-1];
/* Завершение операций обмена граничными строками */
   MPI_Waitall(4, req, status);
...
```

Эта программа та же самая, что и в предыдущем примере. Здесь заменены четыре вызова функции MPI_Wait одним вызовом функции MPI_Waitall.

9.3.6. Задача Дирихле. Реализация обменов функциями предварительной инициализации обменов MPI_Send_init и MPI_Recv_init

Здесь используются функции предварительной инициализации обменов данными MPI_Send_init и MPI_Recv_init, ускоряющие обмен данными.

```
/* Обмен данными в задаче Дирихле. Шестая версия параллельной
 * программы с использованием функций предварительной инициализации
 * обменов MPI_Send_init и MPI_Recv_init. Топология - "линейка".
 */
...
/* Каждая ветвь находит своих соседей */
   if(myrank == 0)
     left = MPI_PROC_NULL;
   else
```

```
    left = myrank-1,
  if(myrank == p-1)
    right = MPI_PROC_NULL,
  else
    right = myrank+1,

/* Инициализация функций обмена данными*/
  MPI_Send_init(B, n, MPI_FLOAT, left, tag, comm, req[0]),
  MPI_Send_init(B[m-1][0], n, MPI_FLOAT, right, tagcomm, req[1]),
  MPI_Recv_init(A[0][1], n, MPI_FLOAT, left, tag, comm, req[2]),
  MPI_Recv_init(A[m+1][1], n, MPI_FLOAT, right, tag, comm, req[3]),

/* Вычисление граничных строк массива B */
  for(i = 1, i <= n, i++)
    { B[0][i-1]  =  0 25*(A[0][i]+A[2][i]+A[1][i+1]+A[1][i-1]),
      B[m-1][i-1] = 0 25*(A[m][i-1]+A[m][i+1]+A[m-1][i]+A[m+1][i]),
    }
/* Старт функций для обмена граничными строками между параллельными ветвями */
  MPI_Startall(4, req),
/* Выполнение четырехточечного шаблона для внутренних строк массива B
 */
  for(j = 2, j <= m-1, j++)
    for(i = 1, i <= n, i++)
      B[j-1][i-1] = 0 25*(A[j][i-1]+A[j][i+1]+A[i][j-1]+A[i][j+1]),
/* Копирование результата обратно в массив A */
  for(j = 1, j <= m, j++)
    for(i = 1, i <= n, i++)
      A[j][i] = B[j-1][i-1],
/* Завершение операций обмена граничными строками */
  MPI_Waitall(4, req, status),
```

9.4. Решение СЛАУ методом Гаусса

Здесь рассматриваются два алгоритма решения СЛАУ методом Гаусса. Они связаны с разными способами представления *данных* (матриц коэффициентов и правых частей) в распределенной памяти мультикомпьютера. Схемы распределения данных по компьютерам приведены в гл. 2 для обоих примеров. Хотя данные распределены в памяти мультикомпьютера в каждом алгоритме по-разному, но оба они реализуются на одной и той же топологии связи компьютеров – "полный граф". Топология "полный граф" обладает одной нехорошей особенностью, а именно, с ростом количества компьютеров в системе увеличивается и время выполнения коллективной операции обмена данными. Поэтому при решении этой задачи, нужно выбрать оптимальное соотношение между объемом данных и размером вычислительной системы. И это соотношение зависит от скорости обменов между компьютерами.

9.4.1. Решение СЛАУ методом Гаусса. Первый алгоритм

В алгоритме, представленном в данном пункте, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами, как показано на рис. 2.9. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер и т. д., последняя полоса – в p_1 компьютер. В примере предполагается, что матрица коэффициентов A и вектор правых частей F разрезаны на части заранее и каждая ветвь считывает свои части с дисковой памяти.

Здесь, в примере, каждая ветвь генерирует свои части матрицы. Схемы распределения данных по компьютерам приведены в п. 2.4.1.

```
/* Первый алгоритм
 *
 * Решение СЛАУ методом Гаусса  Распределение данных - горизонтальными полосами
 * (Запуск задачи на 8-ми компьютерах)
 */
   #include<stdio h>
   #include<mpi h>
   #include<sys/time h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора правой части
 * (Предполагаем, что размеры данных делятся без остатка на количество
 * компьютеров ) */
   #define M 400
   #define N 50
   #define tegD 1
   #define EL(x) (sizeof(x) / sizeof(x[0]))
/* Описываем массивы для полос исходной матрицы - MA и вектор V для приема
 * данных  Для простоты, вектор правой части уравнений присоединяем
 * дополнительным столбцом к матрице коэффициентов  В этом дополнительном
 * столбце и получим результат  */
double MA[N][M+1], V[M+1], MAD, R,

int main(int args, char **argv)
  { int size, MyP, i, j, v, k, d, p,
    int       *index, *edges,
    MPI_Comm   comm_gr,
    MPI_Status status,
    struct timeval tv1, tv2,
    int dt1,
    int reord = 1,
 /* Инициализация библиотеки */
    MPI_Init(&args, &argv),
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size),
 /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP),
 /* Выделяем память под массивы для описания вершин и ребер в топологии
  * полный граф */
    index = (int *)malloc(size * sizeof(int)),
    edges = (int *)malloc(size*(size-1) * sizeof(int)),
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф"  */
    for(i = 0, i < size, i++)
      { index[i] = (size - 1)*(i + 1),
        v = 0,
        for(j = 0, j < size, j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j,
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord, &comm_gr),
```

```
/* Каждая ветвь генерирует свою полосу матрицы A и свой отрезок вектора
 * правой части, который присоединяется дополнительным столбцом к A
 * Нулевая ветвь генерирует нулевую полосу, первая ветвь - первую полосу
 * и т д  (По диагонали исходной матрицы - числа = 2, остальные числа = 1)  */
   for(i = 0, i < N, i++)
     { for(j = 0, j < M, j++)
         { if((N*MyP+i) == j)
             MA[i][j] = 2 0,
           else
             MA[i][j] = 1 0,
         }
       MA[i][M] = 1 0*(M)+1 0,
     }
/* Каждая ветвь засекает начало вычислений и производит вычисления */
   gettimeofday(&tv1, (struct timezone*)0),
/* Прямой ход */
/* Цикл p - цикл по компьютерам  Все ветви, начиная с нулевой, последовательно
 * приводят к диагональному виду свои строки  Ветвь, приводяшая свои строки
 * к диагональному виду, назовем активной, строка, с которой производятся
 * вычисления, так же назовем активной  */
   for(p = 0, p < size, p++)
     {
    /* Цикл k - цикл по строкам  (Все ветви "крутят" этот цикл)  */
      for(k = 0, k < N, k++)
        { if(MyP == p)
            {
           /* Активная ветвь с номером MyP == p приводит свои строки к
            * диагональному виду
            * Активная строка k передается ветвям, с номером большим чем MyP*/
              MAD = 1 0/MA[k][N*p+k],
              for(j = M, j >= N*p+k, j--)
                MA[k][j] = MA[k][j] * MAD,
              for(d = p+1, d < size, d++)
                MPI_Send(&MA[k][0], M+1, MPI_DOUBLE, d, tegD, comm_gr),
              for(i = k+1, i < N, i++)
                { for(j = M, j >= N*p+k, j--)
                    MA[i][j] = MA[i][j]-MA[i][N*p+k]*MA[k][j],
                }
            }
       /* Работа принимающих ветвей с номерами MyP > p */
          else if(MyP > p)
            {  MPI_Recv(V, EL(V), MPI_DOUBLE, p, tegD, comm_gr, &status),
               for(i = 0, i < N, i++)
                 { for(j = M, j >= N*p+k, j--)
                     MA[i][j] = MA[i][j]-MA[i][N*p+k]*V[j],
                 }
            }
        }          /* for k */
     }             /* for p */
/* Обратный ход */
/* Циклы по p и k аналогичны, как и при прямом ходе  */
   for(p = size-1, p >= 0, p--)
     { for(k = N-1, k >= 0, k--)
```

```
        {
       /* Работа активной ветви */
          if(MyP == p)
            { for(d = p-1; d >= 0; d--)
                MPI_Send(&MA[k][M], 1, MPI_DOUBLE, d, tegD, comm_gr);
              for(i = k-1; i >= 0; i--)
                MA[i][M] -= MA[k][M]*MA[i][N*p+k];
            }
       /* Работа ветвей с номерами MyP < p */
          else
            { if(MyP < p)
                { MPI_Recv(&R, 1, MPI_DOUBLE, p, tegD, comm_gr, &status);
                  for(i = N-1; i >= 0; i--)
                    MA[i][M] -= R*MA[i][N*p+k];
                }
            }
        }                  /* for k */
    }                      /* for p */
 /* Все ветви засекают время и печатают */
    gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec)*1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("MyP = %d Time = %d\n", MyP, dt1);
 /* Все ветви печатают, для контроля, свои первые четыре значения корня */
    printf("MyP = %d %f %f %f %f\n",MyP,MA[0][M],MA[1][M],MA[2][M],MA[3][M]);
/* Все ветви завершают выполнение */
    MPI_Finalize();
    return(0);
  }
```

9.4.2. Решение СЛАУ методом Гаусса. Второй алгоритм

В алгоритме, представленном в данном пункте, исходная матрица коэффициентов A и вектор правых частей F разрезаны циклическими горизонтальными полосами, как показано на рис. 2.11. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер и т. д. В примере предполагается, что матрица A и вектор правых частей F разрезаны на части заранее и каждая ветвь считывает свои части с дисковой памяти. Здесь, в примере, каждая ветвь генерирует свои части матрицы. Схемы распределения данных по компьютерам приведены в п. 2.4.2.

```
/* Второй алгоритм
 *
 * Решение СЛАУ методом Гаусса. Распределение данных - циклическими
 * горизонтальными полосами.
 * (Запуск задачи на 8-ми компьютерах).
 */
   #include<stdio.h>
   #include<mpi.h>
   #include<sys/time.h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора правой части.
 * (Предполагаем, что размеры данных делятся без остатка на количество
 * компьютеров.) */
   #define M 400
   #define N 50
   #define tegD 1
```

```
/* Описываем массив для циклических полос исходной матрицы - MA и вектор V для
 * приема данных  Для простоты, вектор правой части уравнений присоединяем
 * дополнительным столбцом к матрице коэффициентов  В этом дополнительном
 * столбце и получим результат  */
double MA[N][M+1], V[M+1], MAD, R,

int main(int args, char **argv)
  { int        size, MyP, i, j, v, k, k1, p,
    int       *index, *edges,
    MPI_Comm   comm_gr,
    struct timeval tv1, tv2,
    int dt1,
    int reord = 1,
 /* Инициализация библиотеки */
    MPI_Init(&args, &argv),
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size),
 /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP),
 /* Выделяем память под массивы для описания вершин и ребер в топологии
  * полный граф */
    index = (int *)malloc(size * sizeof(int)),
    edges = (int *)malloc(size*(size-1)*sizeof(int)),
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф"  */
    for(i = 0, i < size, i++)
      { index[i] = (size - 1)*(i + 1),
        v = 0,
        for(j = 0, j < size, j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j,
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord, &comm_gr),
 /* Каждая ветвь генерирует свои циклические полосы матрицы A и свой отрезок
  * вектора правой части, который присоединяется дополнительным столбцом к A
  * Нулевая ветвь генерирует следующие строки исходной матрицы  0, size,
  * 2*size, 3*size, и т д  Первая ветвь - строки  1, 1+size, 1+2*size, 1+3*size
  * и т д  Вторая ветвь - строки  2, 2+size, 2+2*size, 2+3*size и т д
  * (По диагонали исходной матрицы - числа = 2, остальные числа = 1)  */
    for(i = 0, i < N, i++)
      { for(j = 0, j < M, j++)
          { if((MyP+size*i) == j)
              MA[i][j] = 2 0,
            else
              MA[i][j] = 1 0,
          }
        MA[i][M] = 1 0*(M)+1 0,
      }
 /* Каждая ветвь засекает начало вычислений и производит вычисления */
    gettimeofday(&tv1, (struct timezone*)0),
 /* Прямой ход */
 /* Цикл k - цикл по строкам  Все ветви, начиная с нулевой, последовательно
```

```
  * приводят к диагональному виду свои строки. Ветвь, приводящая свои строки
  * к диагональному виду, назовем активной, строка, с которой производятся
  * вычисления, так же назовем активной. */
    for(k = 0; k < N; k++)
      {
     /* Цикл p - цикл по компьютерам. (Все ветви "крутят" этот цикл). */
        for(p = 0; p < size; p++)
          { if(MyP == p)
              {
             /* Активная ветвь с номером MyP == p приводит свою строку с
              * номером k к диагональному виду
              * Активная строка - k передается всем ветвям. */
                MAD = 1 0/MA[k][size*k+p];
                for(j = M; j >= size*k+p; j--)
                  MA[k][j] = MA[k][j] * MAD;
                for(j = 0; j <= M; j++)
                  V[j] = MA[k][j];
                MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
                for(i = k+1; i < N; i++)
                  { for(j = M; j >= size*k+p; j--)
                      MA[i][j] = MA[i][j]-MA[i][size*k+p]*MA[k][j];
                  }
              }
          /* Работа принимающих ветвей с номерами MyP < p */
             else if(MyP < p)
              { MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
                for(i = k+1; i < N; i++)
                  { for(j = M; j >= size*k+p; j--)
                      MA[i][j] = MA[i][j] - MA[i][size*k+p]*V[j];
                  }
              }
           /* Работа принимающих ветвей с номерами MyP > p */
              else if(MyP > p)
                { MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
                    for(i = k; i < N; i++)
                      { for(j = M; j >= size*k+p; j--)
                          MA[i][j] = MA[i][j] - MA[i][ size*k+p]*V[j];
                      }
                 }
          }           /*for p */
      }               /*for k */
/* Обратный ход */
 /* Циклы по k и p аналогичны, как и при прямом ходе. */
     for(k1 = N-2, k = N-1; k >= 0; k--,k1--)
       { for(p = size-1; p >= 0; p--)
           { if(MyP == p)
               {
              /* Работа активной ветви */
                 R = MA[k][M];
                 MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
                 for(i = k-1; i >= 0; i--)
                   MA[i][M] -= MA[k][M]*MA[i][size*k+p];
               }
```

```
          /* Работа ветвей с номерами MyP < p */
             else if(MyP < p)
               { MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
                 for(i = k; i >= 0; i--)
                   MA[i][M] -= R*MA[i][ size*k+p];
               }
          /* Работа ветвей с номерами MyP > p */
             else if(MyP > p)
               { MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
                   for(i = k1; i >= 0; i--)
                     MA[i][M] -= R*MA[i][size*k+p];
               }
           }                  /* for p */
       }                      /* for k */
 /* Все ветви засекают время и печатают */
    gettimeofday(&tv2, (struct timezone*)0);
    dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
    printf("MyP = %d Time = %d\n", MyP, dt1);
 /* Все ветви печатают, для контроля, свои первые четыре значения корня */
    printf("MyP = %d %f %f %f %f\n", MyP, MA[0][M],MA[1][M],MA[2][M],MA[3][M]);
 /* Все ветви завершают выполнение */
    MPI_Finalize();
    return(0);
  }
```

9.5. Решение СЛАУ методом простой итерации

В алгоритме, представленном в данном пункте, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами, как показано на рис. 2.9. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер, и т. д., последняя полоса – в p_1 компьютер. В примере предполагается, что матрица коэффициентов A и вектор правых частей F разрезаны на части заранее и каждая ветвь считывает свои части с дисковой памяти. Здесь, в примере, каждая ветвь генерирует свои части матрицы. Схемы распределения данных по компьютерам приведены в п. 2.4.1. Вид уравнения и общая схема решения представлены в п. 2.5.

```
/*
 * Решение СЛАУ методом простой итерации. Распределение данных -
 * горизонтальными полосами. (Запуск задачи на 8-ми компьютерах).
 */

   #include<stdio.h>
   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора правой части.
 * (Предполагаем, что размеры данных делятся без остатка на количество
 * компьютеров.) */
   #define M 64
   #define N 8
   #define EL(x) (sizeof(x) / sizeof(x[0]))
   #define ABS(X) ((X) < 0 ? -(X) : (X))
```

```
/* Задаем необходимую точность приближенных корней */
   #define E 0.0001
/* Задаем шаг итерации */
   #define T 0.1
/* Описываем массивы для полос исходной матрицы - MA, вектора правой части - F,
 * значения приближений на предыдущей итерации - Y и текущей - Y1, результата
 * умножения матрицы коэффициентов на вектор - S, и всего вектора значения
 * приближений на предыдущей итерации - V  */
static double MA[N][M], F[N], Y[N], Y1[N], S[N], V[M];

int main(int argc, char **argv)
  { int         i, j, z, H, MyP, size, v;
    int        *index, *edges;
    MPI_Comm    comm_gr;
    struct timeval tv1, tv2;       /* Для засечения времени */
    int dt1;
    int         reord = 1;
 /* Инициализация библиотеки */
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);
 /* Выделяем память под массивы для описания вершин и ребер в топологии
  * полный граф */
    index = (int *)malloc(size * sizeof(int));
    edges = (int *)malloc(size*(size-1)*sizeof(int));
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф". */
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord, &comm_gr);
 /* Каждая ветвь генерирует свои полосы матрицы A и свой отрезок вектора
  * правой части.
  * (По диагонали исходной матрицы - числа = 2, остальные числа = 1). */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((N*MyP + i) == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        F[i] = M + 1;
      }
 /* Каждая ветвь засекает начало вычислений и производит вычисления */
    gettimeofday(&tv1, (struct timezone*)0);
 /* Каждая ветвь задает начальное приближение корней. */
```

```
   for(i = 0, i < N, i++)
     Y1[i] = 0 8,
/* Начало вычислений */
   do
     { for(i = 0, i < N, i++)
         { S[i] = 0 0,
           Y[i] = Y1[i],
         }
     /* В каждой ветви формируем весь вектор предыдущей итерации и умножаем
      * матрицу коэффициентов на этот вектор */
       MPI_Allgather(Y, EL(Y), MPI_DOUBLE, V, EL(Y), MPI_DOUBLE, comm_gr),
       for(j = 0, j < N, j++)
         for(i = 0, i < M, i++)
           S[j] += MA[j][i] * V[i],

       z = 0,              /* Флаг завершения вычислений всеми ветвями */
       for(i = 0, i < N, i++)
         { Y1[i] = Y[i] - T*(S[i] - F[i]),
           if(ABS(ABS(Y1[i]) - ABS(Y[i])) > E)
           z = 1,
         }
     /* Суммируем все флаги (по всем ветвям) и результат записываем в H в
      * каждой ветви */
       MPI_Allreduce(&z, &H, 1, MPI_INT, MPI_SUM, comm_gr),
     }
   while(H > 0),
/* Все ветви засекают время и печатают */
   gettimeofday(&tv2, (struct timezone*)0),
   dt1 = (tv2 tv_sec - tv1 tv_sec) * 1000000 + tv2 tv_usec - tv1 tv_usec,
   printf("MyP = %d Time = %d\n", MyP, dt1),
/* Все ветви печатают, для контроля, свои первые четыре значения корня */
   printf("Rez MyP = %d Y0=%f Y1=%f Y2=%f Y3=%f\n", MyP,Y[0],Y[1],Y[2],Y[3]),
/* Все ветви завершают выполнение */
   MPI_Finalize(),
   return(0),
 }
```

9.6. Упорядочивание множеств

Алгоритм, представленный в данном пункте, упорядочивает заданные множества чисел по возрастанию (убыванию). Части исходного множества, расположенные в каждой ветви, должны быть неизменными как в процессе упорядочивания, так и в конечном итоге.

```
/*
 * Упорядочивание множеств чисел
 * (Запуск задачи на 8-ми компьютерах)
 */
   #include<stdio h>
   #include<mpi h>
   #include<time h>
   #include<sys/time h>
/* Каждая ветвь задает размеры своих частей множества
 * (Предполагаем, что размеры данных делятся без остатка на
```

```
  * количество компьютеров.) */
    #define M 80
 /* Задаем размерность декартовой топологии.
  * Задачу решаем на топологии "линейка". */
    #define NUM_DIMS 1
 /* Каждая ветвь задает массив для своей части множества. */
static int MNOJ[M];
 /* Подпрограмма генерации целых чисел и их упорядочивания по возрастанию. */
 Generac(int rn)
  { int i, k, R;
    srand(( rn + 1 )*(unsigned)time(0));
    for(i = 0; i < M; i++)
    MNOJ[i] = 1 + rand()%10000;
    do
      { k = 0;
        for(i = 0; i < M-1; i++)
          { if(MNOJ[i+1] < MNOJ[i])
              { R = MNOJ[i];
                MNOJ[i] = MNOJ[i+1];
                MNOJ[i+1] = R;
                k = 1;
              }
          }
       }
    while(k > 0);
  }
 /* Подпрограмма вставки целых чисел в MNOJ, начиная с большего при pr == 1,
  * и их упорядочивания по возрастанию, и вставки, начиная с меньшего при
  * pr == 0, и их упорядочивания по возрастанию. */
 Vstav(int pr, int ch)
  { int i, k;
    k = 0;
    if(pr == 1)
      { for(i = 1; i < M; i++)
          { if(MNOJ[M-i-1] <= ch)
              { MNOJ[M-i] = ch;
                k = 1;
                break;
              }
            else
              MNOJ[M-i] = MNOJ[M-i-1];
          }
        if( k == 0 )
          MNOJ[0] = ch;
      }
    else
      { for(i = 1; i < M; i++)
          { if(MNOJ[i] >= ch)
              { MNOJ[i-1] = ch;
                k = 1;
                break;
              }
            else
```

```
                MNOJ[i-1] = MNOJ[i],
            }
          if( k == 0 )
             MNOJ[M-1] = ch,
       }
  }
 /* Главная программа  */
int main(int argc, char **argv)
  { int   i, z, H, RB, RM, MAX, MIN,
    int   MyP, size, destm, sourm, destb, sourb,
    int   dims[NUM_DIMS], periods[NUM_DIMS], new_coords[NUM_DIMS],
    int   reorder = 0,
    MPI_Comm       comm_cart,
    MPI_Status     st,
    struct timeval tv1, tv2,        /* Для засечения времени */
    int   dt1,
 /* Инициализация библиотеки */
    MPI_Init( &argc, &argv ),
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size),
 /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP),
 /* Обнуляем массив dims и заполняем массив periods для топологии "линейка" */
    for(i = 0, i < NUM_DIMS, i++) { dims[i] = 0, periods[i] = 0, }
 /* Заполняем массив dims, где указываются размеры (одномерной) решетки */
    MPI_Dims_create(size, NUM_DIMS, dims),
 /* Создаем топологию "линейка" с communicator(ом) comm_cart */
    MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
                                                          &comm_cart),
 /* Отображаем ранги на координаты компьютеров, с целью оптимизации
  * отображения заданной виртуальной топологии на физическую топологию
  * системы  */
    MPI_Cart_coords(comm_cart, MyP, NUM_DIMS, new_coords),
 /* Каждая ветвь находит своих соседей вдоль линейки  */
    if(new_coords[0] == 0)
      { destm = sourm = MPI_PROC_NULL,
        destb = sourb = 1,
      }
    else if(new_coords[0] == size-1)
      { destm = sourm = size-2,
        destb = sourb = MPI_PROC_NULL,
      }
    else
      { destm = sourm = new_coords[0]-1,
        destb = sourb = new_coords[0]+1,
      }
 /* Каждая ветвь генерирует в MNOJ свою часть множества целых чисел и
  * упорядочивает их по возрастанию */
    Generac(MyP),
 /* Засекаем начальный момент времени  */
    gettimeofday(&tv1, (struct timezone*)0),

    do
```

```
    { z=0;
      MIN = MNOJ[0];
      MPI_Sendrecv(&MIN, 1, MPI_INT, destm, 12, &RB, 1, MPI_INT, sourb, 12,
                                                          comm_cart, &st),
      if(MyP != size-1)
        { if( MNOJ[M-1] > RB )
            { MAX = MNOJ[M-1];
              Vstav(1, RB),
              z = 1;
            }
          else
            MAX = RB,
        }
      MPI_Sendrecv(&MAX, 1, MPI_INT, destb, 12, &RM, 1, MPI_INT, sourm, 12,
                                                          comm_cart, &st),
      if(MyP != 0)
        { if( MNOJ[1] < RM )
            { Vstav(0, RM),
              z = 1,
            }
          else
            MNOJ[0] = RM;
        }
      MPI_Allreduce(&z, &H, 1, MPI_INT, MPI_SUM, comm_cart);
    }
  while(H > 0);
/* Все ветви засекают время и печатают */
  gettimeofday(&tv2, (struct timezone*)0);
  dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec,
  printf("MyP = %d Time = %d\n", MyP, dt1);
/* Все ветви печатают, для контроля, значения своих частей множества  */
  printf("MyP = %d   %d - %d\n", MyP, MNOJ[0], MNOJ[M-1]);

/* Все остальные ветви в области связи comm_cart будут стоять,
 * пока ветвь 1 не выведет сообщение
 */
       MPI_Barrier(MPI_COMM_WORLD);
/* Без точки синхронизации может оказаться, что одна из ветвей
 * вызовет printf() раньше, чем успеет отработать printf()
 * предыдущей ветви, и выдача на экран будет хаотичной. */
/* Все ветви завершают выполнение */
  MPI_Finalize();
  return(0);
}
```