Федеральный ресурсный центр Южного Федерального Округа

Ростовский государственный университет

Южно-российский региональный центр информатизации высшей школы

А. А. Букатов, В. Н. Дацюк, А. И. Жегуло

Программирование многопроцессорных вычислительных систем

Ростов-на-Дону
2003

Издано в рамках реализации проекта создания Федерального ресурсного центра Южного Федерального округа

ISBN 5-94153-062-5

А. А. Букатов, В. Н. Дацюк, А. И. Жегуло. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

Данная книга представляет собой пособие для тех, кто желает ознакомиться с технологиями программирования для многопроцессорных вычислительных систем. В ней не обсуждаются сложные теоретические вопросы параллельного программирования. Скорее это практическое руководство, в котором авторы попытались систематизировать свой собственный опыт освоения этих технологий. Основное внимание уделено системам с распределенной памятью. К числу таких систем относятся и широко распространенные в настоящее время кластерные системы.

По своей структуре книга состоит из трех частей. В первой части приводится обзор архитектур многопроцессорных вычислительных систем и средств их программирования. Вторая часть книги посвящена рассмотрению среды параллельного программирования MPI. Третья часть представляет собой методическое руководство по работе с библиотеками параллельных подпрограмм ScaLAPACK и Aztec.

Книга предназначена для лиц, занимающихся компьютерным моделированием и решением объемных вычислительных задач. Рекомендуется преподавателям, аспирантам и студентам естественно-научных факультетов.

ISBN 5-94153-062-5

Печатается по решению редакционно-издательского совета ЮГИНФО РГУ

© А. А. Букатов, В. Н. Дацюк, А. И. Жегуло , 2003
© ЮГИНФО РГУ, 2003

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5

ЧАСТЬ 1.
ВВЕДЕНИЕ В АРХИТЕКТУРЫ И СРЕДСТВА ПРОГРАММИРОВАНИЯ МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ 13

Глава 1. ОБЗОР АРХИТЕКТУР МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ 13
- 1.1. Векторно-конвейерные суперкомпьютеры 15
- 1.2. Симметричные мультипроцессорные системы (SMP) 17
- 1.3. Системы с массовым параллелизмом (MPP) 20
- 1.4. Кластерные системы 23
- 1.5. Классификация вычислительных систем 26

Глава 2. КРАТКАЯ ХАРАКТЕРИСТИКА СРЕДСТВ ПРОГРАММИРОВАНИЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ 27
- 2.1. Системы с общей памятью 28
- 2.2. Системы с распределенной памятью 30

Глава 3. ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ НА MPP СИСТЕМАХ 34
- 3.1. Параллельное программирование на MPP системах 34
- 3.2. Эффективность параллельных программ 40
- 3.3. Использование высокопроизводительных технологий 42

Глава 4. МНОГОПРОЦЕССОРНАЯ ВЫЧИСЛИТЕЛЬНАЯ СИСТЕМА nCUBE2 48
- 4.1. Общее описание вычислительной системы 48
- 4.2. Структура программного обеспечения nCUBE2 51
- 4.3. Работа на многопроцессорной системе nCUBE2 52
- 4.4. Получение информации о системе и управление процессами 60
- 4.5. Средства параллельного программирования на nCUBE2 61
- 4.6. Библиотека подпрограмм хост-компьютера для взаимодействия с параллельными программами nCUBE2 67
- 4.7. Пример параллельной программы с использованием средств PSE 68

Глава 5. ВЫСОКОПРОИЗВОДИТЕЛЬНЫЙ ВЫЧИСЛИТЕЛЬНЫЙ КЛАСТЕР 72
- 5.1. Архитектура вычислительного кластера 72
- 5.2. Система пакетной обработки заданий 77

ЗАКЛЮЧЕНИЕ К ЧАСТИ 1 84

ЧАСТЬ 2.
СРЕДА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ MPI 86

Глава 6. ОБЩАЯ ОРГАНИЗАЦИЯ MPI 86

Глава 7. БАЗОВЫЕ ФУНКЦИИ MPI 91

Глава 8. КОММУНИКАЦИОННЫЕ ОПЕРАЦИИ ТИПА ТОЧКА-ТОЧКА 95
8.1. Обзор коммуникационных операций типа точка-точка 95
8.2. Блокирующие коммуникационные операции 97
8.3. Неблокирующие коммуникационные операции 103

Глава 9. КОЛЛЕКТИВНЫЕ ОПЕРАЦИИ 108
9.1. Обзор коллективных операций 108
9.2. Функции сбора блоков данных от всех процессов группы 112
9.3. Функции распределения блоков данных по всем процессам группы 117
9.4. Совмещенные коллективные операции 119
9.5. Глобальные вычислительные операции над распределенными данными 120

Глава 10. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ И ПЕРЕДАЧА УПАКОВАННЫХ ДАННЫХ 126
10.1. Производные типы данных 127
10.2. Передача упакованных данных 135

Глава 11. РАБОТА С ГРУППАМИ И КОММУНИКАТОРАМИ 139
11.1. Определение основных понятий 139
11.2. Функции работы с группами 140
11.3. Функции работы с коммуникаторами 144

Глава 12. ТОПОЛОГИЯ ПРОЦЕССОВ 147
12.1. Основные понятия 147
12.2. Декартова топология 148

Глава 13. ПРИМЕРЫ ПРОГРАММ 154
13.1. Вычисление числа π 154
13.2. Перемножение матриц 156
13.3. Решение краевой задачи методом Якоби 160

ЗАКЛЮЧЕНИЕ К ЧАСТИ 2 164

Часть 3.
БИБЛИОТЕКИ ПОДПРОГРАММ ДЛЯ МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ 165

Глава 14. БИБЛИОТЕКА ПОДПРОГРАММ ScaLAPACK 165
14.1. История разработки пакета ScaLAPACK и его общая организация 165
14.2. Структура пакета ScaLAPACK 167
14.3. Использование библиотеки ScaLAPACK 169
14.4. Примеры использования пакета ScaLAPACK 181

Глава 15. Использование библиотеки параллельных подпрограмм Aztec 191
15.1. Общая организация библиотеки Aztec 191
15.2. Конфигурационные параметры библиотеки Aztec 192
15.3. Основные подпрограммы библиотеки Aztec 197
15.4. Хранение разреженных матриц в MSR формате 201
15.5. Пример использования библиотеки Aztec 202

ЗАКЛЮЧЕНИЕ К ЧАСТИ 3 206

ЛИТЕРАТУРА И ИНТЕРНЕТ-РЕСУРСЫ 207

ВВЕДЕНИЕ

Прошло немногим более 50 лет с момента появления первых электронных вычислительных машин – компьютеров. За это время сфера их применения охватила практически все области человеческой деятельности. Сегодня невозможно представить себе эффективную организацию работы без применения компьютеров в таких областях, как планирование и управление производством, проектирование и разработка сложных технических устройств, издательская деятельность, образование – словом, во всех областях, где возникает необходимость в обработке больших объемов информации. Однако наиболее важным по-прежнему остается использование их в том направлении, для которого они собственно и создавались, а именно, для решения больших задач, требующих выполнения громадных объемов вычислений. Такие задачи возникли в середине прошлого века в связи с развитием атомной энергетики, авиастроения, ракетно-космических технологий и ряда других областей науки и техники.

В наше время круг задач, требующих для своего решения применения мощных вычислительных ресурсов, еще более расширился. Это связано с тем, что произошли фундаментальные изменения в самой организации научных исследований. Вследствие широкого внедрения вычислительной техники значительно усилилось направление численного моделирования и численного эксперимента. Численное моделирование, заполняя промежуток между физическими экспериментами и аналитическими подходами, позволило изучать явления, которые являются либо слишком сложными для исследования аналитическими методами, либо слишком дорогостоящими или опасными для экспериментального изучения. При этом численный эксперимент позволил значительно удешевить процесс научного и технологического поиска. Стало возможным моделировать в реальном времени процессы интенсивных физико-химических и ядерных реакций, глобальные атмосферные

процессы, процессы экономического и промышленного развития регионов и т.д. Очевидно, что решение таких масштабных задач требует значительных вычислительных ресурсов [1].

Вычислительное направление применения компьютеров всегда оставалось основным двигателем прогресса в компьютерных технологиях. Не удивительно поэтому, что в качестве основной характеристики компьютеров используется такой показатель, как *производительность* – величина, показывающая, какое количество арифметических операций он может выполнить за единицу времени. Именно этот показатель с наибольшей очевидностью демонстрирует масштабы прогресса, достигнутого в компьютерных технологиях. Так, например, производительность одного из самых первых компьютеров EDSAC составляла всего около 100 операций в секунду, тогда как пиковая производительность самого мощного на сегодняшний день суперкомпьютера Earth Simulator оценивается в 40 триллионов операций/сек. Т.е. произошло увеличение быстродействия в 400 миллиардов раз! Невозможно назвать другую сферу человеческой деятельности, где прогресс был бы столь очевиден и так велик.

Естественно, что у любого человека сразу же возникает вопрос: за счет чего это оказалось возможным? Как ни странно, ответ довольно прост: примерно 1000-кратное увеличение скорости работы электронных схем и максимально широкое распараллеливание обработки данных.

Идея параллельной обработки данных как мощного резерва увеличения производительности вычислительных аппаратов была высказана Чарльзом Бэббиджем примерно за сто лет до появления первого электронного компьютера. Однако уровень развития технологий середины 19-го века не позволил ему реализовать эту идею. С появлением первых электронных компьютеров эти идеи неоднократно становились отправной точкой при разработке самых передовых и производительных вычислительных систем. Без преувеличения можно сказать, что вся

история развития высокопроизводительных вычислительных систем – это история реализации идей параллельной обработки на том или ином этапе развития компьютерных технологий, естественно, в сочетании с увеличением частоты работы электронных схем.

Принципиально важными решениями в повышении производительности вычислительных систем были: введение конвейерной организации выполнения команд; включение в систему команд векторных операций, позволяющих одной командой обрабатывать целые массивы данных; распределение вычислений на множество процессоров. Сочетание этих 3-х механизмов в архитектуре суперкомпьютера Earth Simulator, состоящего из 5120 векторно-конвейерных процессоров, и позволило ему достичь рекордной производительности, которая в 20000 раз превышает производительность современных персональных компьютеров.

Очевидно, что такие системы чрезвычайно дороги и изготавливаются в единичных экземплярах. Ну, а что же производится сегодня в промышленных масштабах? Широкое разнообразие производимых в мире компьютеров с большой степенью условности можно разделить на четыре класса:

- персональные компьютеры (Personal Computer – PC);
- рабочие станции (WorkStation – WS);
- суперкомпьютеры (Supercomputer – SC);
- кластерные системы.

Условность разделения связана в первую очередь с быстрым прогрессом в развитии микроэлектронных технологий. Производительность компьютеров в каждом из классов удваивается в последние годы примерно за 18 месяцев (закон Мура). В связи с этим, суперкомпьютеры начала 90-х годов зачастую уступают в производительности современным рабочим станциям, а персональные компьютеры начинают успешно конкурировать по производительности с рабочими станциями. Тем не менее, попытаемся каким-то образом классифицировать их.

Персональные компьютеры. Как правило, в этом случае подразумеваются однопроцессорные системы на платформе Intel или AMD, работающие под управлением однопользовательских операционных систем (Microsoft Windows и др.). Используются в основном как персональные рабочие места.

Рабочие станции. Это чаще всего компьютеры с RISC процессорами с многопользовательскими операционными системами, относящимися к семейству ОС UNIX. Содержат от одного до четырех процессоров. Поддерживают удаленный доступ. Могут обслуживать вычислительные потребности небольшой группы пользователей.

Суперкомпьютеры. Отличительной особенностью суперкомпьютеров является то, что это, как правило, большие и, соответственно, чрезвычайно дорогие многопроцессорные системы. В большинстве случаев в суперкомпьютерах используются те же серийно выпускаемые процессоры, что и в рабочих станциях. Поэтому зачастую различие между ними не столько качественное, сколько количественное. Например, можно говорить о 4-х процессорной рабочей станции фирмы SUN и о 64-х процессорном суперкомпьютере фирмы SUN. Скорее всего, в том и другом случае будут использоваться одни и те же микропроцессоры.

Кластерные системы. В последние годы широко используются во всем мире как дешевая альтернатива суперкомпьютерам. Система требуемой производительности собирается из готовых серийно выпускаемых компьютеров, объединенных опять же с помощью некоторого серийно выпускаемого коммуникационного оборудования.

Таким образом, многопроцессорные системы, которые ранее ассоциировались в основном с суперкомпьютерами, в настоящее время прочно утвердились во всем диапазоне производимых вычислительных систем, начиная от персональных компьютеров и заканчивая суперкомпьютерами на базе векторно-конвейерных процессоров. Это обстоятельство, с одной стороны, увеличивает доступность

суперкомпьютерных технологий, а, с другой, повышает актуальность их освоения, поскольку для всех типов многопроцессорных систем требуется использование специальных технологий программирования для того, чтобы программа могла в полной мере использовать ресурсы высокопроизводительной вычислительной системы. Обычно это достигается разделением программы с помощью того или иного механизма на параллельные ветви, каждая из которых выполняется на отдельном процессоре.

Суперкомпьютеры разрабатываются в первую очередь для того, чтобы с их помощью решать сложные задачи, требующие огромных объемов вычислений. При этом подразумевается, что может быть создана единая программа, для выполнения которой будут задействованы все ресурсы суперкомпьютера. Однако не всегда такая единая программа может быть создана или ее создание целесообразно. В самом деле, при разработке параллельной программы для многопроцессорной системы мало разбить программу на параллельные ветви. Для эффективного использования ресурсов необходимо обеспечить равномерную загрузку всех процессоров, что в свою очередь означает, что все ветви программы должны выполнить примерно одинаковый объем вычислительной работы. Однако не всегда этого можно достичь. Например, при решении некоторой параметрической задачи для разных значений параметров время поиска решения может значительно различаться. В таких случаях, видимо, разумнее независимо выполнять расчеты для каждого параметра с помощью обычной однопроцессорной программы. Но даже в таком простом случае могут потребоваться суперкомпьютерные ресурсы, поскольку выполнение полного расчета на однопроцессорной системе может потребовать слишком длительного времени. Параллельное выполнение множества программ для различных значений параметров позволяет существенно ускорить решение задачи. Наконец, следует отметить, что использование суперкомпьютеров всегда более эффективно

для обслуживания вычислительных потребностей большой группы пользователей, чем использование эквивалентного количества однопроцессорных рабочих станций, так как в этом случае с помощью некоторой системы управления заданиями легче обеспечить равномерную и более эффективную загрузку вычислительных ресурсов.

В отличие от обычных многопользовательских систем для достижения максимальной скорости выполнения программ операционные системы суперкомпьютеров, как правило, не позволяют разделять ресурсы одного процессора между разными, одновременно выполняющимися программами. Поэтому, как два крайних варианта, возможны следующие режимы использования **n**-процессорной системы:

1. все ресурсы отдаются для выполнения одной программы, и тогда мы вправе ожидать **n**-кратного ускорения работы программы по сравнению с однопроцессорной системой;
2. одновременно выполняется **n** обычных однопроцессорных программ, при этом пользователь вправе рассчитывать, что на скорость выполнения его программы не будут оказывать влияния другие программы.

Вопрос о том, насколько эффективно могут использоваться ресурсы многопроцессорной системы в этих двух режимах работы, и будет нас интересовать при обсуждении архитектур различных суперкомпьютеров и других многопроцессорных систем.

В силу ряда причин из перечисленных выше 4-х классов вычислительных систем в России достаточное распространение получили только персональные компьютеры, и соответственно ситуация с освоением технологий работы с этими компьютерами более или менее благополучная. Компьютерный парк систем, относящихся к классу мультипроцессорных рабочих станций и, особенно, к классу суперкомпьютеров, чрезвычайно мал. Вследствие этого наметилось существенное отставание в подготовке специалистов в области программирования для таких систем, без которых,

в свою очередь, невозможно эффективное использование даже имеющегося компьютерного парка. В этом плане весьма показателен опыт эксплуатации в Ростовском государственном университете многопроцессорной системы nCUBE2. Система была установлена в Ростовском госуниверситете в 1997 г., однако потребовалось более 2-х лет, прежде чем на ней заработали первые реальные параллельные прикладные программы, предназначенные для решения задач математического моделирования. Более того, в значительной степени это оказалось возможным только после включения в программное обеспечение nCUBE2 мобильной среды параллельного программирования MPI и основанных на ней библиотек параллельных подпрограмм. Учитывая, что моральное старение компьютеров в настоящее время происходит очень быстро, столь длительный период освоения, конечно, недопустим. В качестве оправдания отметим, что это был первый опыт эксплуатации таких систем не только в Ростовском университете, но и вообще в России, поэтому практически отсутствовала какая-либо литература по данному вопросу. К счастью, оказалось, что накопленное программное обеспечение легко переносится на другие многопроцессорные системы, а освоенные технологии программирования имеют достаточно универсальный характер.

Данная книга адресована тем, кто желает ознакомиться с технологиями программирования для многопроцессорных систем. В ней не обсуждаются сложные теоретические вопросы параллельного программирования. Скорее это практическое руководство, в котором авторы попытались систематизировать свой собственный опыт освоения этих технологий. Мы, конечно, ни в коей мере не претендуем на полноту освещения данной темы. Речь в основном пойдет о системах, подобных nCUBE2 – системах с распределенной памятью. К числу таких систем относятся и широко распространенные в настоящее время кластерные системы.

По своей структуре книга состоит из трех частей. В первой части приводится обзор архитектур многопроцессорных вычислительных систем (МВС) и средств их программирования. Рассматриваются вопросы, связанные с повышением производительности вычислений. В заключительных главах первой части рассматриваются архитектура и средства программирования вычислительных систем суперкомпьютерного центра Ростовского госуниверситета: МВС nCUBE2 и гетерогенного вычислительного кластера. Вторая часть книги полностью посвящена рассмотрению наиболее распространенной среды параллельного программирования – коммуникационной библиотеки MPI, ставшей в настоящее время фактическим стандартом для разработки мобильных параллельных программ. Третья часть, по сути, представляет собой методическое руководство по работе с библиотеками параллельных подпрограмм ScaLAPACK и Aztec. Первая из них предназначена для работы с плотными матрицами общего вида, а вторая – для решения больших систем линейных уравнений с разреженными матрицами.

Введение мы завершим определением некоторых понятий, используемых в литературе по технологиям высокопроизводительных вычислений.

Единицы измерения объема данных:

1 байт – минимальная адресуемая единица информации = 8 бит

1 Кб (килобайт) = 1024 байт

1 Мб (мегабайт) = 1024 Кб

1 Гб (гигабайт) = 1024 Мб

Единицы измерения производительности вычислительных систем:

1 Mflops (мегафлопс) = 1 миллион оп/сек

1 Gflops (гигафлопс) = 1 миллиард оп/сек

1 Tflops (терафлопс) = 1 триллион оп/сек

Часть 1.
ВВЕДЕНИЕ В АРХИТЕКТУРЫ И СРЕДСТВА ПРОГРАММИРОВАНИЯ МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Глава 1.
ОБЗОР АРХИТЕКТУР МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

В процессе развития суперкомпьютерных технологий идею повышения производительности вычислительной системы за счет увеличения числа процессоров использовали неоднократно. Если не вдаваться в исторический экскурс и обсуждение всех таких попыток, то можно следующим образом вкратце описать развитие событий.

Экспериментальные разработки по созданию многопроцессорных вычислительных систем начались в 70-х годах 20 века. Одной из первых таких систем стала разработанная в Иллинойском университете МВС ILLIAC IV, которая включала 64 (в проекте до 256) процессорных элемента (ПЭ), работающих по единой программе, применяемой к содержимому собственной оперативной памяти каждого ПЭ. Обмен данными между процессорами осуществлялся через специальную матрицу коммуникационных каналов. Указанная особенность коммуникационной системы дала название “матричные суперкомпьютеры” соответствующему классу МВС. Отметим, что более широкий класс МВС с распределенной памятью и с произвольной коммуникационной системой получил впоследствии название “многопроцессорные системы с массовым параллелизмом”, или МВС с MPP-архитектурой (MPP – Massively Parallel Processing). При этом, как правило, каждый из ПЭ MPP системы является универсальным процессором, действующим по своей

собственной программе (в отличие от общей программы для всех ПЭ матричной МВС).

Первые матричные МВС выпускались буквально поштучно, поэтому их стоимость была фантастически высокой. Серийные же образцы подобных систем, такие как ICL DAP, включавшие до 8192 ПЭ, появились значительно позже, однако не получили широкого распространения ввиду сложности программирования МВС с одним потоком управления (с одной программой, общей для всех ПЭ).

Первые промышленные образцы мультипроцессорных систем появились на базе векторно-конвейерных компьютеров в середине 80-х годов. Наиболее распространенными МВС такого типа были суперкомпьютеры фирмы Cray. Однако такие системы были чрезвычайно дорогими и производились небольшими сериями. Как правило, в подобных компьютерах объединялось от 2 до 16 процессоров, которые имели равноправный (симметричный) доступ к общей оперативной памяти. В связи с этим они получили название симметричные мультипроцессорные системы (Symmetric Multi-Processing – SMP).

Как альтернатива таким дорогим мультипроцессорным системам на базе векторно-конвейерных процессоров была предложена идея строить эквивалентные по мощности многопроцессорные системы из большого числа дешевых серийно выпускаемых микропроцессоров. Однако очень скоро обнаружилось, что SMP архитектура обладает весьма ограниченными возможностями по наращиванию числа процессоров в системе из-за резкого увеличения числа конфликтов при обращении к общей шине памяти. В связи с этим оправданной представлялась идея снабдить каждый процессор собственной оперативной памятью, превращая компьютер в объединение независимых вычислительных узлов. Такой подход значительно увеличил степень масштабируемости многопроцессорных систем, но в свою очередь потребовал разработки специального способа обмена данными между вычислительными узлами,

реализуемого обычно в виде механизма передачи сообщений (Message Passing). Компьютеры с такой архитектурой являются наиболее яркими представителями MPP систем. В настоящее время эти два направления (или какие-то их комбинации) являются доминирующими в развитии суперкомпьютерных технологий.

Нечто среднее между SMP и MPP представляют собой NUMA-архитектуры (Non Uniform Memory Access), в которых память физически разделена, но логически общедоступна. При этом время доступа к различным блокам памяти становится неодинаковым. В одной из первых систем этого типа Cray T3D время доступа к памяти другого процессора было в 6 раз больше, чем к своей собственной.

В настоящее время развитие суперкомпьютерных технологий идет по четырем основным направлениям: векторно-конвейерные суперкомпьютеры, SMP системы, MPP системы и кластерные системы. Рассмотрим основные особенности перечисленных архитектур.

1.1. Векторно-конвейерные суперкомпьютеры

Первый векторно-конвейерный компьютер Cray-1 появился в 1976 году. Архитектура его оказалась настолько удачной, что он положил начало целому семейству компьютеров. Название этому семейству компьютеров дали два принципа, заложенные в архитектуре процессоров:

- конвейерная организация обработки потока команд
- введение в систему команд набора векторных операций, которые позволяют оперировать с целыми массивами данных [2].

Длина одновременно обрабатываемых векторов в современных векторных компьютерах составляет, как правило, 128 или 256 элементов. Очевидно, что векторные процессоры должны иметь гораздо более сложную структуру и по сути дела содержать множество арифметических устройств. Основное назначение векторных операций состоит в распараллеливании выполнения операторов цикла, в которых в основном и

сосредоточена большая часть вычислительной работы. Для этого циклы подвергаются процедуре векторизации с тем, чтобы они могли реализовываться с использованием векторных команд. Как правило, это выполняется автоматически компиляторами при изготовлении ими исполнимого кода программы. Поэтому векторно-конвейерные компьютеры не требовали какой-то специальной технологии программирования, что и явилось решающим фактором в их успехе на компьютерном рынке. Тем не менее, требовалось соблюдение некоторых правил при написании циклов с тем, чтобы компилятор мог их эффективно векторизовать.

Исторически это были первые компьютеры, к которым в полной мере было применимо понятие суперкомпьютер. Как правило, несколько векторно-конвейерных процессоров (2-16) работают в режиме с общей памятью (SMP), образуя вычислительный узел, а несколько таких узлов объединяются с помощью коммутаторов, образуя либо NUMA, либо MPP систему. Типичными представителями такой архитектуры являются компьютеры CRAY J90/T90, CRAY SV1, NEC SX-4/SX-5. Уровень развития микроэлектронных технологий не позволяет в настоящее время производить однокристальные векторные процессоры, поэтому эти системы довольно громоздки и чрезвычайно дороги. В связи с этим, начиная с середины 90-х годов, когда появились достаточно мощные суперскалярные микропроцессоры, интерес к этому направлению был в значительной степени ослаблен. Суперкомпьютеры с векторно-конвейерной архитектурой стали проигрывать системам с массовым параллелизмом. Однако в марте 2002 г. корпорация NEC представила систему Earth Simulator из 5120 векторно-конвейерных процессоров, которая в 5 раз превысила производительность предыдущего обладателя рекорда – MPP системы ASCI White из 8192 суперскалярных микропроцессоров. Это, конечно же, заставило многих по-новому взглянуть на перспективы векторно-конвейерных систем.

1.2. Симметричные мультипроцессорные системы (SMP)

Характерной чертой многопроцессорных систем SMP архитектуры является то, что все процессоры имеют прямой и равноправный доступ к любой точке общей памяти. Первые SMP системы состояли из нескольких однородных процессоров и массива общей памяти, к которой процессоры подключались через общую системную шину. Однако очень скоро обнаружилось, что такая архитектура непригодна для создания сколь либо масштабных систем. Первая возникшая проблема – большое число конфликтов при обращении к общей шине. Остроту этой проблемы удалось частично снять разделением памяти на блоки, подключение к которым с помощью коммутаторов позволило распараллелить обращения от различных процессоров. Однако и в таком подходе неприемлемо большими казались накладные расходы для систем более чем с 32-мя процессорами.

Современные системы SMP архитектуры состоят, как правило, из нескольких однородных серийно выпускаемых микропроцессоров и массива общей памяти, подключение к которой производится либо с помощью общей шины, либо с помощью коммутатора (рис. 1.1).

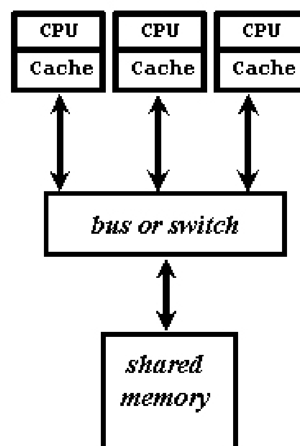

Рис. 1.1. Архитектура симметричных мультипроцессорных систем.

Наличие общей памяти значительно упрощает организацию взаимодействия процессоров между собой и упрощает программирование,

поскольку параллельная программа работает в едином адресном пространстве. Однако за этой кажущейся простотой скрываются большие проблемы, присущие системам этого типа. Все они, так или иначе, связаны с оперативной памятью. Дело в том, что в настоящее время даже в однопроцессорных системах самым узким местом является оперативная память, скорость работы которой значительно отстала от скорости работы процессора. Для того чтобы сгладить этот разрыв, современные процессоры снабжаются скоростной буферной памятью (кэш-памятью), скорость работы которой значительно выше, чем скорость работы основной памяти. В качестве примера приведем данные измерения пропускной способности кэш-памяти и основной памяти для персонального компьютера на базе процессора Pentium III 1000 Мгц. В данном процессоре кэш-память имеет два уровня:

- L1 (буферная память команд) – объем 32 Кб, скорость обмена 9976 Мб/сек;
- L2 (буферная память данных) – объем 256 Кб, скорость обмена 4446 Мб/сек.

В то же время скорость обмена с основной памятью составляет всего 255 Мб/сек. Это означает, что для 100% согласованности со скоростью работы процессора (1000 Мгц) скорость работы основной памяти должна быть в 40 раз выше!

Очевидно, что при проектировании многопроцессорных систем эти проблемы еще более обостряются. Помимо хорошо известной проблемы конфликтов при обращении к общей шине памяти возникла и новая проблема, связанная с иерархической структурой организации памяти современных компьютеров. В многопроцессорных системах, построенных на базе микропроцессоров со встроенной кэш-памятью, нарушается принцип равноправного доступа к любой точке памяти. Данные, находящиеся в кэш-памяти некоторого процессора, недоступны для других процессоров. Это означает, что после каждой модификации копии

некоторой переменной, находящейся в кэш-памяти какого-либо процессора, необходимо производить синхронную модификацию самой этой переменной, расположенной в основной памяти.

С большим или меньшим успехом эти проблемы решаются в рамках общепринятой в настоящее время архитектуры ccNUMA (cache coherent Non Uniform Memory Access). В этой архитектуре память физически распределена, но логически общедоступна. Это, с одной стороны, позволяет работать с единым адресным пространством, а, с другой, увеличивает масштабируемость систем. Когерентность кэш-памяти поддерживается на аппаратном уровне, что не избавляет, однако, от накладных расходов на ее поддержание. В отличие от классических SMP систем память становится трехуровневой:

- кэш-память процессора;
- локальная оперативная память;
- удаленная оперативная память.

Время обращения к различным уровням может отличаться на порядок, что сильно усложняет написание эффективных параллельных программ для таких систем.

Перечисленные обстоятельства значительно ограничивают возможности по наращиванию производительности ccNUMA систем путем простого увеличения числа процессоров. Тем не менее, эта технология позволяет в настоящее время создавать системы, содержащие до 256 процессоров с общей производительностью порядка 200 млрд. операций в секунду. Системы этого типа серийно производятся многими компьютерными фирмами как многопроцессорные серверы с числом процессоров от 2 до 128 и прочно удерживают лидерство в классе малых суперкомпьютеров. Типичными представителями данного класса суперкомпьютеров являются компьютеры SUN StarFire 15K, SGI Origin 3000, HP Superdome. Хорошее описание одной из наиболее удачных систем этого типа – компьютера Superdome фирмы Hewlett-Packard –

можно найти в книге [3]. Неприятным свойством SMP систем является то, что их стоимость растет быстрее, чем производительность при увеличении числа процессоров в системе. Кроме того, из-за задержек при обращении к общей памяти неизбежно взаимное торможение при параллельном выполнении даже независимых программ.

1.3. Системы с массовым параллелизмом (MPP)

Проблемы, присущие многопроцессорным системам с общей памятью, простым и естественным образом устраняются в системах с массовым параллелизмом. Компьютеры этого типа представляют собой многопроцессорные системы с распределенной памятью, в которых с помощью некоторой коммуникационной среды объединяются однородные вычислительные узлы (рис. 1.2).

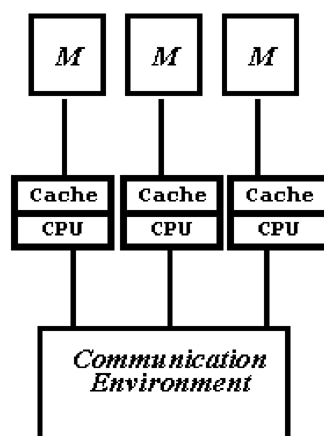

Рис. 1.2. Архитектура систем с распределенной памятью.

Каждый из узлов состоит из одного или нескольких процессоров, собственной оперативной памяти, коммуникационного оборудования, подсистемы ввода/вывода, т.е. обладает всем необходимым для независимого функционирования. При этом на каждом узле может функционировать либо полноценная операционная система (как в системе RS/6000 SP2), либо урезанный вариант, поддерживающий только

базовые функции ядра, а полноценная ОС работает на специальном управляющем компьютере (как в системах Cray T3E, nCUBE2).

Процессоры в таких системах имеют прямой доступ только к своей локальной памяти. Доступ к памяти других узлов реализуется обычно с помощью механизма передачи сообщений. Такая архитектура вычислительной системы устраняет одновременно как проблему конфликтов при обращении к памяти, так и проблему когерентности кэш-памяти. Это дает возможность практически неограниченного наращивания числа процессоров в системе, увеличивая тем самым ее производительность. Успешно функционируют MPP системы с сотнями и тысячами процессоров (ASCI White – 8192, Blue Mountain – 6144). Производительность наиболее мощных систем достигает 10 триллионов оп/сек (10 Tflops). Важным свойством MPP систем является их высокая степень масштабируемости. В зависимости от вычислительных потребностей для достижения необходимой производительности требуется просто собрать систему с нужным числом узлов.

На практике все, конечно, гораздо сложнее. Устранение одних проблем, как это обычно бывает, порождает другие. Для MPP систем на первый план выходит проблема эффективности коммуникационной среды. Легко сказать: “Давайте соберем систему из 1000 узлов”. Но каким образом соединить в единое целое такое множество узлов? Самым простым и наиболее эффективным было бы соединение каждого процессора с каждым. Но тогда на каждом узле потребовалось бы 999 коммуникационных каналов, желательно двунаправленных. Очевидно, что это нереально. Различные производители MPP систем использовали разные топологии. В компьютерах Intel Paragon процессоры образовывали прямоугольную двумерную сетку. Для этого в каждом узле достаточно четырех коммуникационных каналов. В компьютерах Cray T3D/T3E использовалась топология трехмерного тора. Соответственно, в узлах этого компьютера было шесть коммуникационных каналов. Фирма

nCUBE использовала в своих компьютерах топологию n-мерного гиперкуба. Подробнее на этой топологии мы остановимся в главе 4 при изучении суперкомпьютера nCUBE2. Каждая из рассмотренных топологий имеет свои преимущества и недостатки. Отметим, что при обмене данными между процессорами, не являющимися ближайшими соседями, происходит трансляция данных через промежуточные узлы. Очевидно, что в узлах должны быть предусмотрены какие-то аппаратные средства, которые освобождали бы центральный процессор от участия в трансляции данных. В последнее время для соединения вычислительных узлов чаще используется иерархическая система высокоскоростных коммутаторов, как это впервые было реализовано в компьютерах IBM SP2. Такая топология дает возможность прямого обмена данными между любыми узлами, без участия в этом промежуточных узлов.

Системы с распределенной памятью идеально подходят для параллельного выполнения независимых программ, поскольку при этом каждая программа выполняется на своем узле и никаким образом не влияет на выполнение других программ. Однако при разработке параллельных программ приходится учитывать более сложную, чем в SMP системах, организацию памяти. Оперативная память в MPP системах имеет 3-х уровневую структуру:

- кэш-память процессора;
- локальная оперативная память узла;
- оперативная память других узлов.

При этом отсутствует возможность прямого доступа к данным, расположенным в других узлах. Для их использования эти данные должны быть предварительно переданы в тот узел, который в данный момент в них нуждается. Это значительно усложняет программирование. Кроме того, обмены данными между узлами выполняются значительно медленнее, чем обработка данных в локальной оперативной памяти узлов. Поэтому

написание эффективных параллельных программ для таких компьютеров представляет собой более сложную задачу, чем для SMP систем.

1.4. Кластерные системы

Кластерные технологии стали логическим продолжением развития идей, заложенных в архитектуре MPP систем. Если процессорный модуль в MPP системе представляет собой законченную вычислительную систему, то следующий шаг напрашивается сам собой: почему бы в качестве таких вычислительных узлов не использовать обычные серийно выпускаемые компьютеры. Развитие коммуникационных технологий, а именно, появление высокоскоростного сетевого оборудования и специального программного обеспечения, такого как система MPI (см. часть 2), реализующего механизм передачи сообщений над стандартными сетевыми протоколами, сделали кластерные технологии общедоступными. Сегодня не составляет большого труда создать небольшую кластерную систему, объединив вычислительные мощности компьютеров отдельной лаборатории или учебного класса.

Привлекательной чертой кластерных технологий является то, что они позволяют для достижения необходимой производительности объединять в единые вычислительные системы компьютеры самого разного типа, начиная от персональных компьютеров и заканчивая мощными суперкомпьютерами. Широкое распространение кластерные технологии получили как средство создания систем суперкомпьютерного класса из составных частей массового производства, что значительно удешевляет стоимость вычислительной системы. В частности, одним из первых был реализован проект COCOA [4], в котором на базе 25 двухпроцессорных персональных компьютеров общей стоимостью порядка $100000 была создана система с производительностью, эквивалентной 48-процессорному Cray T3D стоимостью несколько миллионов долларов США.

Конечно, о полной эквивалентности этих систем говорить не приходится. Как указывалось в предыдущем разделе, производительность систем с распределенной памятью очень сильно зависит от производительности коммуникационной среды. Коммуникационную среду можно достаточно полно охарактеризовать двумя параметрами: *латентностью* – временем задержки при посылке сообщения и *пропускной способностью* – скоростью передачи информации. Так вот для компьютера Cray T3D эти параметры составляют соответственно 1 мкс и 480 Мб/сек, а для кластера, в котором в качестве коммуникационной среды использована сеть Fast Ethernet, 100 мкс и 10 Мб/сек. Это отчасти объясняет очень высокую стоимость суперкомпьютеров. При таких параметрах, как у рассматриваемого кластера, найдется не так много задач, которые могут эффективно решаться на достаточно большом числе процессоров.

Если говорить кратко, то *кластер* – это связанный набор полноценных компьютеров, используемый в качестве единого вычислительного ресурса. Преимущества кластерной системы перед набором независимых компьютеров очевидны. Во-первых, разработано множество диспетчерских систем пакетной обработки заданий, позволяющих послать задание на обработку кластеру в целом, а не какому-то отдельному компьютеру. Эти диспетчерские системы автоматически распределяют задания по свободным вычислительным узлам или буферизуют их при отсутствии таковых, что позволяет обеспечить более равномерную и эффективную загрузку компьютеров. Во-вторых, появляется возможность совместного использования вычислительных ресурсов нескольких компьютеров для решения одной задачи.

Для создания кластеров обычно используются либо простые однопроцессорные персональные компьютеры, либо двух- или четырехпроцессорные SMP-серверы. При этом не накладывается никаких ограничений на состав и архитектуру узлов. Каждый из узлов может

функционировать под управлением своей собственной операционной системы. Чаще всего используются стандартные ОС: Linux, FreeBSD, Solaris, Tru64 Unix, Windows NT. В тех случаях, когда узлы кластера неоднородны, то говорят о гетерогенных кластерах.

При создании кластеров можно выделить два подхода. Первый подход применяется при создании небольших кластерных систем. В кластер объединяются полнофункциональные компьютеры, которые продолжают работать и как самостоятельные единицы, например, компьютеры учебного класса или рабочие станции лаборатории. Второй подход применяется в тех случаях, когда целенаправленно создается мощный вычислительный ресурс. Тогда системные блоки компьютеров компактно размещаются в специальных стойках, а для управления системой и для запуска задач выделяется один или несколько полнофункциональных компьютеров, называемых *хост-компьютерами*. В этом случае нет необходимости снабжать компьютеры вычислительных узлов графическими картами, мониторами, дисковыми накопителями и другим периферийным оборудованием, что значительно удешевляет стоимость системы.

Разработано множество технологий соединения компьютеров в кластер. Наиболее широко в данное время используется технология Fast Ethernet. Это обусловлено простотой ее использования и низкой стоимостью коммуникационного оборудования. Однако за это приходится расплачиваться заведомо недостаточной скоростью обменов. В самом деле, это оборудование обеспечивает максимальную скорость обмена между узлами 10 Мб/сек, тогда как скорость обмена с оперативной памятью составляет 250 Мб/сек и выше. Разработчики пакета подпрограмм ScaLAPACK, предназначенного для решения задач линейной алгебры на многопроцессорных системах, в которых велика доля коммуникационных операций, формулируют следующим образом требование к многопроцессорной системе: “Скорость межпроцессорных обменов между

двумя узлами, измеренная в Мб/сек, должна быть не менее 1/10 пиковой производительности вычислительного узла, измеренной в Mflops” [5]. Таким образом, если в качестве вычислительных узлов использовать компьютеры класса Pentium III 500 Мгц (пиковая производительность 500 Mflops), то аппаратура Fast Ethernet обеспечивает только 1/5 от требуемой скорости. Частично это положение может поправить переход на технологии Gigabit Ethernet.

Ряд фирм предлагают специализированные кластерные решения на основе более скоростных сетей, таких как SCI фирмы Scali Computer (~100 Мб/сек) и Mirynet (~120 Мб/сек). Активно включились в поддержку кластерных технологий и фирмы-производители высокопроизводительных рабочих станций (SUN, HP, Silicon Graphics).

1.5. Классификация вычислительных систем

Большое разнообразие вычислительных систем породило естественное желание ввести для них какую-то классификацию. Эта классификация должна однозначно относить ту или иную вычислительную систему к некоторому классу, который, в свою очередь, должен достаточно полно ее характеризовать. Таких попыток предпринималось множество. Одна из первых классификаций, ссылки на которую наиболее часто встречаются в литературе, была предложена М. Флинном в конце 60-х годов прошлого века. Она базируется на понятиях двух потоков: команд и данных. На основе числа этих потоков выделяется четыре класса архитектур.

1. SISD (Single Instruction Single Data) – единственный поток команд и единственный поток данных. По сути дела это классическая машина фон Неймана. К этому классу относятся все однопроцессорные системы.
2. SIMD (Single Instruction Multiple Data) – единственный поток команд и множественный поток данных. Типичными представителями

являются матричные компьютеры, в которых все процессорные элементы выполняют одну и ту же программу, применяемую к своим (различным для каждого ПЭ) локальным данным. Некоторые авторы к этому классу относят и векторно-конвейерные компьютеры, если каждый элемент вектора рассматривать как отдельный элемент потока данных.

3. MISD (Multiple Instruction Single Date) – множественный поток команд и единственный поток данных. М. Флинн не смог привести ни одного примера реально существующей системы, работающей на этом принципе. Некоторые авторы в качестве представителей такой архитектуры называют векторно-конвейерные компьютеры, однако такая точка зрения не получила широкой поддержки.
4. MIMD (Multiple Instruction Multiple Date) – множественный поток команд и множественный поток данных. К этому классу относятся практически все современные многопроцессорные системы.

Поскольку в этой классификации все современные многопроцессорные системы принадлежат одному классу, то вряд ли эта классификация представляет сегодня какую-либо практическую ценность. Тем не менее, мы привели ее потому, что используемые в ней термины достаточно часто упоминаются в литературе по параллельным вычислениям.

Глава 2.
КРАТКАЯ ХАРАКТЕРИСТИКА СРЕДСТВ ПРОГРАММИРОВАНИЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Эффективность использования компьютеров в решающей степени зависит от состава и качества программного обеспечения, установленного на них. В первую очередь это касается программного обеспечения, предназначенного для разработки прикладных программ. Так, например, недостаточная развитость таких средств для систем MPP типа долгое

время являлась сдерживающим фактором для их широкого использования. В настоящее время ситуация изменилась, и благодаря кластерным технологиям MPP системы стали самой распространенной и доступной разновидностью высокопроизводительных вычислительных систем. В данной главе мы кратко рассмотрим средства параллельного программирования для многопроцессорных систем.

Как отмечалось ранее, основной характеристикой при классификации многопроцессорных систем является наличие общей (SMP системы) или распределенной (MPP системы) памяти. Это различие является важнейшим фактором, определяющим способы параллельного программирования и, соответственно, структуру программного обеспечения.

2.1. Системы с общей памятью

К системам этого типа относятся компьютеры с SMP архитектурой, различные разновидности NUMA систем и мультипроцессорные векторно-конвейерные компьютеры. Характерным словом для этих компьютеров является “единый”: единая оперативная память, единая операционная система, единая подсистема ввода-вывода. Только процессоры образуют множество. Единая UNIX-подобная операционная система, управляющая работой всего компьютера, функционирует в виде множества процессов. Каждая пользовательская программа также запускается как отдельный процесс. Операционная система сама каким-то образом распределяет процессы по процессорам. В принципе, для распараллеливания программ можно использовать механизм порождения процессов. Однако этот механизм не очень удобен, поскольку каждый процесс функционирует в своем адресном пространстве, и основное достоинство этих систем – общая память – не может быть использован простым и естественным образом. Для распараллеливания программ используется механизм порождения нитей (threads) – легковесных процессов, для которых не

создается отдельного адресного пространства, но которые на многопроцессорных системах также распределяются по процессорам. В языке программирования C возможно прямое использование этого механизма для распараллеливания программ посредством вызова соответствующих системных функций, а в компиляторах с языка FORTRAN этот механизм используется либо для автоматического распараллеливания, либо в режиме задания распараллеливающих директив компилятору (такой подход поддерживают и компиляторы с языка C).

Все производители симметричных мультипроцессорных систем в той или иной мере поддерживают стандарт POSIX Pthread и включают в программное обеспечение распараллеливающие компиляторы для популярных языков программирования или предоставляют набор директив компилятору для распараллеливания программ. В частности, многие поставщики компьютеров SMP архитектуры (Sun, HP, SGI) в своих компиляторах предоставляют специальные директивы для распараллеливания циклов. Однако эти наборы директив, во-первых, весьма ограничены и, во-вторых, несовместимы между собой. В результате этого разработчикам приходится распараллеливать прикладные программы отдельно для каждой платформы.

В последние годы все более популярной становится система программирования OpenMP [3, 6], являющаяся во многом обобщением и расширением этих наборов директив. Интерфейс OpenMP задуман как стандарт для программирования в модели общей памяти. В OpenMP входят спецификации набора директив компилятору, процедур и переменных среды. По сути дела, он реализует идею "инкрементального распараллеливания", позаимствованную из языка HPF (High Performance Fortran – Fortran для высокопроизводительных вычислений) (см. раздел 2.2). Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы. При этом система программирования OpenMP предоставляет разработчику

большие возможности по контролю над поведением параллельного приложения. Вся программа разбивается на последовательные и параллельные области. Все последовательные области выполняет главная нить, порождаемая при запуске программы, а при входе в параллельную область главная нить порождает дополнительные нити. Предполагается, что OpenMP-программа без какой-либо модификации должна работать как на многопроцессорных системах, так и на однопроцессорных. В последнем случае директивы OpenMP просто игнорируются. Следует отметить, что наличие общей памяти не препятствует использованию технологий программирования, разработанных для систем с распределенной памятью. Многие производители SMP систем предоставляют также такие технологии программирования, как MPI и PVM. В этом случае в качестве коммуникационной среды выступает разделяемая память.

2.2. Системы с распределенной памятью

В системах этого типа на каждом вычислительном узле функционирует собственные копии операционной системы, под управлением которых выполняются независимые программы. Это могут быть как действительно независимые программы, так и параллельные ветви одной программы. В этом случае единственно возможным механизмом взаимодействия между ними является механизм передачи сообщений.

Стремление добиться максимальной производительности заставляет разработчиков при реализации механизма передачи сообщений учитывать особенности архитектуры многопроцессорной системы. Это способствует написанию более эффективных, но ориентированных на конкретный компьютер программ. Вместе с тем независимыми разработчиками программного обеспечения было предложено множество реализаций механизма передачи сообщений, независимых от конкретной платформы. Наиболее известными из них являются EXPRESS компании Parasoft и

коммуникационная библиотека PVM (Parallel Virtual Machine), разработанная в Oak Ridge National Laboratory.

В 1994 г. был принят стандарт механизма передачи сообщений MPI (Message Passing Interface) [7]. Он готовился с 1992 по 1994 гг. группой Message Passing Interface Forum, в которую вошли представители более чем 40 организаций из Америки и Европы. Основная цель, которую ставили перед собой разработчики MPI – это обеспечение полной независимости приложений, написанных с использованием MPI, от архитектуры многопроцессорной системы, без какой-либо существенной потери производительности. По замыслу авторов это должно было стать мощным стимулом для разработки прикладного программного обеспечения и стандартизованных библиотек подпрограмм для многопроцессорных систем с распределенной памятью. Подтверждением того, что эта цель была достигнута, служит тот факт, что в настоящее время этот стандарт поддерживается практически всеми производителями многопроцессорных систем. Реализации MPI успешно работают не только на классических MPP системах, но также на SMP системах и на сетях рабочих станций (в том числе и неоднородных).

MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Поддерживаются интерфейсы для языков C и FORTRAN. В последнее время добавлена поддержка языка C++. Библиотека включает в себя множество функций передачи сообщений типа точка-точка, развитый набор функций для выполнения коллективных операций и управления процессами параллельного приложения. Основное отличие MPI от предшественников в том, что явно вводятся понятия групп процессов, с которыми можно оперировать как с конечными множествами, а также областей связи и коммуникаторов, описывающих эти области связи. Это предоставляет программисту очень гибкие средства для написания эффективных параллельных программ. В настоящее время MPI является

основной технологией программирования для многопроцессорных систем с распределенной памятью. Достаточно подробно MPI будет описан в части 2.

Несмотря на значительные успехи в развитии технологии программирования с использованием механизма передачи сообщений, трудоемкость программирования с использованием этой технологии все-таки слишком велика.

Альтернативный подход предоставляет парадигма параллельной обработки данных, которая реализована в языке высокого уровня HPF [8]. От программиста требуется только задать распределение данных по процессорам, а компилятор автоматически генерирует вызовы функций синхронизации и передачи сообщений (неудачное расположение данных может вызвать существенное увеличение накладных расходов). Для распараллеливания циклов используются либо специальные конструкции языка (оператор FORALL), либо директивы компилятору, задаваемые в виде псевдокомментариев ($HPF INDEPENDENT). Язык HPF реализует идею инкрементального распараллеливания и модель общей памяти на системах с распределенной памятью. Эти два обстоятельства и определяют простоту программирования и соответственно привлекательность этой технологии. Одна и та же программа, без какой-либо модификации, должна эффективно работать как на однопроцессорных системах, так и на многопроцессорных вычислительных системах.

Программы на языке HPF существенно короче функционально идентичных программ, использующих прямые вызовы функций обмена сообщениями. По-видимому, языки этого типа будут активно развиваться и постепенно вытеснят из широкого обращения пакеты передачи сообщений. Последним будет отводиться роль, которая в современном программировании отводится ассемблеру: к ним будут прибегать лишь для разработки языков высокого уровня и при написании библиотечных подпрограмм, от которых требуется максимальная эффективность.

В середине 90-х годов, когда появился HPF, с ним связывались большие надежды, однако трудности с его реализацией пока что не позволили создать достаточно эффективных компиляторов. Пожалуй, наиболее удачными были проекты, в которых компиляция HPF-программ выполняется в два этапа. На первом этапе HPF-программа преобразуется в стандартную Фортран-программу, дополненную вызовами коммуникационных подпрограмм. А на втором этапе происходит ее компиляция стандартными компиляторами. Здесь следует отметить систему компиляции Adaptor [9], разработанную немецким Институтом алгоритмов и научных вычислений, и систему разработки параллельных программ DVM, созданную в Институте прикладной математики им. М.В. Келдыша РАН. Система DVM поддерживает не только язык Фортран, но и C [10]. Наш опыт работы с системой Adaptor показал, что в большинстве случаев эффективность параллельных HPF-программ значительно ниже, чем MPI-программ. Тем не менее, в некоторых простых случаях Adaptor позволяет распараллелить обычную программу добавлением в нее всего нескольких строк.

Следует отметить область, где механизму передачи сообщений нет альтернативы – это обслуживание функционального параллелизма. Если каждый узел выполняет свой собственный алгоритм, существенно отличающийся от того, что делает соседний процессор, а взаимодействие между ними имеет нерегулярный характер, то ничего другого, кроме механизма передачи сообщений, предложить невозможно.

Описанные в данной главе технологии, конечно же, не исчерпывают весь список – приведены наиболее универсальные и широко используемые в настоящее время. Более подробный обзор современных технологий параллельного программирования можно найти в книге [3].

Глава 3.
ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ 33НА MPP СИСТЕМАХ

Решение на компьютере вычислительной задачи для выбранного алгоритма решения предполагает выполнение некоторого фиксированного объема арифметических операций. Ускорить решение задачи можно одним из трех способов:

1. использовать более производительную вычислительную систему с более быстрым процессором и более скоростной системной шиной;
2. оптимизировать программу, например, в плане более эффективного использования скоростной кэш-памяти;
3. распределить вычислительную работу между несколькими процессорами, т.е. перейти на параллельные технологии.

3.1. Параллельное программирование на MPP системах

Разработка параллельных программ является весьма трудоемким процессом, особенно для систем MPP типа, поэтому, прежде чем приступать к этой работе, важно правильно оценить как ожидаемый эффект от распараллеливания, так и трудоемкость выполнения этой работы. Очевидно, что без распараллеливания не обойтись при программировании алгоритмов решения тех задач, которые в принципе не могут быть решены на однопроцессорных системах. Это может проявиться в двух случаях: либо когда для решения задачи требуется слишком много времени, либо когда для программы недостаточно оперативной памяти на однопроцессорной системе. Для небольших задач зачастую оказывается, что параллельная версия работает медленнее, чем однопроцессорная. Такая ситуация наблюдается, например, при решении на nCUBE2 систем линейных алгебраических уравнений, содержащих менее 100 неизвестных. Заметный эффект от распараллеливания начинает наблюдаться при решении систем с 1000 и более неизвестными. На кластерных системах

ситуация еще хуже. Разработчики уже упоминавшегося ранее пакета ScaLAPACK для многопроцессорных систем с приемлемым соотношением между производительностью узла и скоростью обмена (сформулированного в разделе 1.4) дают следующую формулу для количества процессоров, которое рекомендуется использовать при решении задач линейной алгебры:

$P = M \times N / 10^6$, где $M \times N$ – размерность матрицы.

Или, другими словами, количество процессоров должно быть таково, чтобы на процессор приходился блок матрицы размером примерно 1000×1000. Эта формула, конечно, носит рекомендательный характер, но, тем не менее, наглядно иллюстрирует, для задач какого масштаба разрабатывался пакет ScaLAPACK. Рост эффективности распараллеливания при увеличении размера решаемой системы уравнений объясняется очень просто: при увеличении размерности решаемой системы уравнений объем вычислительной работы растет пропорционально n^3, а объем обменов между процессорами пропорционально n^2. Это снижает относительную долю коммуникационных затрат при увеличении размерности системы уравнений. Однако, как мы увидим далее, не только коммуникационные издержки влияют на эффективность параллельного приложения.

Параллельные технологии на MPP системах допускают две модели программирования (похожие на классификацию М. Флинна):

1. SPMD (Single Program Multiple Date) – на всех процессорах выполняются копии одной программы, обрабатывающие разные блоки данных;
2. MPMD (Multiple Program Multiple Date) – на процессорах выполняются разные программы, обрабатывающие разные данные.

Второй вариант иногда называют функциональным распараллеливанием. Такой подход, в частности, используется в системах обработки видеоинформации, когда множество квантов данных должны

проходить несколько этапов обработки. В этом случае вполне оправданной будет конвейерная организация вычислений, при которой каждый этап обработки выполняется на отдельном процессоре. Однако такой подход имеет весьма ограниченное применение, поскольку организовать достаточно длинный конвейер, да еще с равномерной загрузкой всех процессоров, весьма сложно.

Наиболее распространенным режимом работы на системах с распределенной памятью является загрузка в некоторое число процессоров одной и той же копии программы. Рассмотрим вопрос, каким образом при этом можно достичь большей скорости решения задачи по сравнению с однопроцессорной системой.

Разработка параллельной программы подразумевает разбиение задачи на P подзадач, каждая из которых решается на отдельном процессоре. Таким образом, упрощенную схему параллельной программы, использующей механизм передачи сообщений, можно представить следующим образом:

```
IF (proc_id.EQ.0)
CALL task1
END IF
IF (proc_id.EQ.1)
CALL task2
END IF
…..
result = reduce(result_task1, result_task2, …)
END
```

Здесь proc_id – идентификатор процессора, а функция reduce формирует некий глобальный результат на основе локальных результатов, полученных на каждом процессоре. В этом случае одна и та же копия программы будет выполняться на P процессорах, но каждый процессор будет решать только свою подзадачу. Если разбиение на подзадачи достаточно равномерное, а накладные расходы на обмены не слишком велики, то можно ожидать близкого к P коэффициента ускорения решения задачи.

Отдельного обсуждения требует понятие *подзадача*. В параллельном программировании это понятие имеет весьма широкий смысл. В MPMD модели под подзадачей понимается некоторый функционально выделенный фрагмент программы. В SPMD модели под подзадачей чаще понимается обработка некоторого блока данных. На практике процедура распараллеливания чаще всего применяется к циклам. Тогда в качестве отдельных подзадач могут выступать экземпляры тела цикла, выполняемые для различных значений переменной цикла. Рассмотрим простейший пример:

```
DO I = 1,1000
C(I)  = C(I) + A(I+1)
END DO
```

В этом примере можно выделить 1000 независимых подзадач вычисления компонентов вектора C, каждая из которых, в принципе, может быть выполнена на отдельном процессоре. Предположим, что в распоряжении программиста имеется 10-ти процессорная система, тогда в качестве независимой подзадачи можно оформить вычисление 100 элементов вектора C. При этом до выполнения вычислений необходимо принять решение о способе размещения этих массивов в памяти процессоров. Здесь возможны два варианта:

1. Все массивы целиком хранятся в каждом процессоре, тогда процедура распараллеливания сводится к вычислению стартового и конечного значений переменной цикла для каждого процессора. В каждом процессоре будет храниться своя копия всего массива, в которой будет модифицирована только часть элементов. В конце вычислений, возможно, потребуется сборка модифицированных частей со всех процессоров.
2. Все или часть массивов распределены по процессорам, т.е. в каждом процессоре хранится 1/P часть массива. Тогда может потребоваться алгоритм установления связи индексов локального массива в некотором процессоре с глобальными индексами всего массива,

например, если значение элемента массива является некоторой функцией индекса. Если в процессе вычислений окажется, что какие-то требующиеся компоненты массива отсутствуют в данном процессоре, то потребуется их пересылка из других процессоров.

Отметим, что вопрос распределения данных по процессорам и связь этого распределения с эффективностью параллельной программы является основным вопросом параллельного программирования. Хранение копий всех массивов во всех процессорах во многих случаях уменьшает накладные расходы на пересылки данных, однако не дает выигрыша в плане объема решаемой задачи и создает сложности синхронизации копий массива при независимом изменении элементов этого массива различными процессорами. Распределение массивов по процессорам позволяет решать значительно более объемные задачи (их то как раз и имеет смысл распараллеливать), но тогда на первый план выступает проблема минимизации пересылок данных.

Рассмотренный выше пример достаточно хорошо укладывается в схему методологического подхода к решению задачи на многопроцессорной системе, который излагается Фостером [11]. Автор выделяет 4 этапа разработки параллельного алгоритма:

1. разбиение задачи на минимальные независимые подзадачи (partitioning);
2. установление связей между подзадачами (communication);
3. объединение подзадач с целью минимизации коммуникаций (agglomeration);
4. распределение укрупненных подзадач по процессорам таким образом, чтобы обеспечить равномерную загрузку процессоров (mapping).

Эта схема, конечно, не более чем описание философии параллельного программирования, которая лишь подчеркивает отсутствие какого-либо формализованного подхода в параллельном программировании для

MPP систем. Если 1-й и 2-й пункты имеют более или менее однозначное решение, то решение задач 3-го и 4-го пунктов основывается главным образом на интуиции программиста. Чтобы проиллюстрировать это обстоятельство, рассмотрим следующую задачу. Предположим, требуется исследовать поведение определителя матрицы в зависимости от некоторого параметра. Один из подходов состоит в том, чтобы написать параллельную версию подпрограммы вычисления определителя и вычислить его значения для исследуемого интервала значений параметра. Однако, если размер матрицы относительно невелик, то может оказаться, что значительные усилия на разработку параллельной подпрограммы вычисления определителя не дадут сколь либо существенного выигрыша в скорости работы программы. В этом случае, скорее всего, более продуктивным подходом будет использование для нахождения определителя стандартной оптимизированной однопроцессорной подпрограммы, а по процессорам разложить исследуемый диапазон изменений параметра.

В заключение рассмотрим, какими свойствами должна обладать многопроцессорная система MPP типа для исполнения на ней параллельных программ. Минимально необходимый набор требуемых для этого средств удивительно мал:

1. Процессоры в системе должны иметь уникальные идентификаторы (номера).
2. Должна существовать функция идентификации процессором самого себя.
3. Должны существовать функции обмена между двумя процессорами: посылка сообщения одним процессором и прием сообщения другим процессором.

Парадигма передачи сообщений подразумевает асимметрию функций передачи и приема сообщений. Инициатива инициализации обмена принадлежит передающей стороне. Принимающий процессор

может принять только то, что ему было послано. Различные реализации механизма передачи сообщений для облегчения разработки параллельных программ вводят те или иные расширения минимально необходимого набора функций.

3.2. Эффективность параллельных программ

В идеале решение задачи на P процессорах должно выполняться в P раз быстрее, чем на одном процессоре, или/и должно позволить решить задачу с объемами данных, в P раз большими. На самом деле такое ускорение практически никогда не достигается. Причина этого хорошо иллюстрируется законом Амдала [12]:

$$S \le 1/(f + (1-f)/P) \qquad (3.1)$$

где S – ускорение работы программы на P процессорах, а f – доля непараллельного кода в программе.

Эта формула справедлива и при программировании в модели общей памяти, и в модели передачи сообщений. Несколько разный смысл вкладывается в понятие *доля непараллельного кода*. Для SMP систем (модель общей памяти) эту долю образуют те операторы, которые выполняет только главная нить программы. Для MPP систем (механизм передачи сообщений) непараллельная часть кода образуется за счет операторов, выполнение которых дублируется всеми процессорами. Оценить эту величину из анализа текста программы практически невозможно. Такую оценку могут дать только реальные просчеты на различном числе процессоров. Из формулы (3.1) следует, что P-кратное ускорение может быть достигнуто, только когда доля непараллельного кода равна 0. Очевидно, что добиться этого практически невозможно. Очень наглядно действие закона Амдала демонстрирует таблица 3.1.

Таблица 3.1. Ускорение работы программы в зависимости от доли непараллельного кода.

Число процессоров	Доля последовательных вычислений %				
	50	25	10	5	2
	Ускорение работы программы				
2	1.33	1.60	1.82	1.90	1.96
4	1.60	2.28	3.07	3.48	3.77
8	1.78	2.91	4.71	5.93	7.02
16	1.88	3.36	6.40	9.14	12.31
32	1.94	3.66	7.80	12.55	19.75
512	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

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

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

распараллеливание заведомо неэффективных программ. Эту ситуацию достаточно наглядно проиллюстрирует следующий раздел.

3.3. Использование высокопроизводительных технологий

Рассмотрим проблемы, возникающие при разработке высокоэффективных программ для современных вычислительных систем на примере элементарной задачи перемножения двух квадратных матриц. Поскольку эта задача чрезвычайно проста, то она особенно наглядно демонстрирует, что разработка высокоэффективных приложений представляет собой весьма нетривиальную задачу. С другой стороны, для этой задачи достаточно просто отслеживать производительность компьютера на реальном приложении. Для оценки этой величины мы будем измерять время выполнения фиксированного числа операций, которые необходимо выполнить для решения задачи. Для перемножения двух квадратных матриц размерности N нужно выполнить (2*N - 1)*N*N арифметических операций.

Фиксирование просто времени выполнения программы не совсем удобно, поскольку программа может выполняться при различных исходных данных (при различных размерностях матриц), и тогда очень сложно сопоставлять результаты тестов. Кроме того, само по себе время выполнения программы мало что говорит об эффективности ее выполнения. Более интересно знать, с какой производительностью работает вычислительная система на этой программе.

Конечно, это не совсем корректный тест, поскольку используются только операции умножения и сложения, но его результаты весьма поучительны. Будем выполнять перемножение достаточно больших матриц (1000×1000) и сравнивать полученную производительность с той, которую декларируют производители компьютеров. Тестирование будем выполнять на 3-х различных вычислительных системах: двухпроцессорной

системе SUN Ultra60, Linux-кластере с узлами Pentium III 500 Mhz и двухпроцессорной системе Compaq Alpha DS20E.

Для компьютера Alpha производительность одного процессора оценивается в 1000 Mflops, для SUN Ultra60 – 800 Mflops, для Pentium III 500 Mhz – 500 Mflops. Будем называть эти величины *пиковой* или *потенциальной производительностью* и посмотрим, в какой мере мы можем достичь этих показателей на реальном приложении.

Для решения нашей задачи любой программист написал бы на языке Фортран77 что-то подобное приведенному ниже:

```
      program matmult
      integer i, j, k, n, nm
      parameter (n=1000)
      real*8 a(n,n), b(n,n), c(n,n), s
      real*8 time, dsecnd, op, mf
      op = (2.d0*n-1)*n*n
c  Блок начальной инициализации
      do 1 i = 1,n
      do 1 j = 1,n
      a(i,j) = dble(i)
      b(i,j) = 1./dble(j)
    1 continue
c  Вычислительный блок
      time = dsecnd()
      do 2 i = 1,n
      do 2 j = 1,n
      s =0.0D0
      do 3 k = 1,n
    3 s = s + a(i,k)*b(k,j)
      c(i,j) = s
    2  continue
c Блок печати
      time = dsecnd() - time
      mf = op/(time*1000000.0)
      write(*,10) c(1,1),c(1,n),c(n,1),c(n,n)
   10 format(2x,2f16.6)
      write(*,*) ' time calculation: ', time,
     *'  mflops: ',mf
      end
c Функция таймер
      double precision function dsecnd()
      real tarray(2)
      dsecnd = etime(tarray)
      return
      end
```

Откомпилируем программу в стандартном режиме:

```
f77 -o matmult matmult.f
```

и посмотрим результат на разных системах.

Таблица 3.2. Время решения и производительность при компиляции в стандартном режиме.

Решение на 1-ом процессоре	Ultra60	Linux	Alpha
время решения (сек.)	261.42	153.21	108.41
производительность (Mflops)	7.65	13.05	18.44

Результат, мягко говоря, обескураживающий. Ни о каких сотнях и тысячах Mflops речь не идет и близко. Производительность в 40 и более раз ниже, чем декларируют производители вычислительных систем. Здесь мы сознательно при компиляции не включали оптимизацию, чтобы проверить, насколько компиляторы могут повысить производительность программы при автоматической оптимизации.

Откомпилируем программу в режиме оптимизации (Ultra60 и Alpha: **fast -O4**, Linux: **-O**).

Таблица 3.3. Время решения и производительность при компиляции с оптимизацией.

Решение на 1-ом процессоре	Ultra60	Linux	Alpha
время решения (сек.)	134.76	58.84	90.84
производительность (Mflops)	14.83	33.97	22.00

Как и следовало ожидать, автоматическая оптимизация в различной степени ускорила решение задачи (в соответствии с обычной практикой в 2-3 раза), но все равно производительность удручающе мала. Причем наибольшую производительность демонстрирует как раз не самая быстрая машина (Pentium III под Linux).

Обратим внимание на то, что в операторе, помеченном меткой 3, выполняется выборка элементов строки из отстоящих далеко друг от друга элементов массива (массивы в Фортране размещаются по столбцам). Это не позволяет буферизовать их в быстрой кэш-памяти. Перепишем вычислительную часть программы следующим образом:

```
    do 2 i = 1,n
    do m = 1,n
    row(m) = a(i,m)
    end do
    do 2 j = 1,n
    c(i,j) =0.0d0
    do 3 k = 1,n
3   c(i,j) = c(i,j) + row(k)*b(k,j)
2   continue
```

т.е. мы завели промежуточный массив, в который предварительно выбираем строку матрицы A. Результат в следующей таблице.

Таблица 3.4. Время решения и производительность при предварительной выборке строки матрицы.

Решение на 1-ом процессоре	Ultra60	Linux	Alpha
время решения (сек.)	33.76	54.41	11.16
производительность (Mflops)	59.21	36.74	179.13

Результаты заметно улучшились для компьютера Ultra60 и особенно для Alpha, которые обладают достаточно большой кэш-памятью. Это подтверждает наше предположение о важности эффективного использования кэш-памяти.

На эффективное использование кэш-памяти нацелены поставляемые с высокопроизводительными системами оптимизированные математические библиотеки BLAS. До недавнего времени это обстоятельство было одним из основных аргументов в конкурентной борьбе фирм-производителей высокопроизводительных систем с компьютерами "желтой" сборки. В настоящее время ситуация изменилась. Написана и бесплатно распространяется самонастраивающаяся библиотека BLAS (ATLAS), которая может устанавливаться на любом компьютере, под любой операционной системой.

В частности, в библиотеке BLAS есть готовая процедура перемножения матриц. Заменим в программе весь вычислительный блок на вызов соответствующей процедуры:

```
CALL  DGEMM('N', 'N', N, N, N, ONE, A, N, B, N, ZERO, C, N)
```

Приведем (табл. 3.5) результаты для программы, в которой для перемножения матриц используется подпрограмма DGEMM из библиотеки ATLAS. Заметим, что эффективность этой подпрограммы не уступает, а в некоторых случаях и превосходит эффективность подпрограммы из стандартных библиотек, поставляемых с программным обеспечением компьютеров (Sun Performance Library для Ultra60 и Common Extended Math Library для Alpha).

Таблица 3.5. Время решения и производительность при использовании подпрограммы DGEMM из библиотеки ATLAS.

Решение на 1-ом процессоре	Ultra60	Linux	Alpha
время решения (сек.)	2.72	5.36	2.24
производительность (Mflops)	734.8	372.9	894.0

Итак, в конце концов, мы получили результаты для производительности, близкие к тем, которые декларируются производителями вычислительных систем. По сравнению с первым вариантом расчета (табл. 3.2) ускорение работы программы для систем Ultra60, Pentium III и Alpha составило соответственно 96, 31 и 48 раз!

Поскольку все тестируемые системы являются мультипроцессорными, то можно попытаться еще ускорить решение нашей задачи за счет использования параллельных технологий. Рассмотрим два варианта. В первом случае выполним распараллеливание самого быстрого однопроцессорного варианта, не использующего библиотечных подпрограмм. Данные для этой однопроцессорной программы приведены в таблице 3.4. Текст параллельной версии программы приведен в конце 2-й части данной книги. Результаты тестирования представлены в таблице 3.6.

Таблица 3.6. Производительность параллельной версии программы, не использующей библиотечных подпрограмм (в Mflops).

Число процессоров	Ultra60	Linux	Alpha
1	59.5	36.2	179.1
2	378.6	66.9	357.9
4	-	130.6	-

Эти данные показывают, что эффективность распараллеливания достаточно высока и производительность программы хорошо растет при увеличении числа процессоров, однако она значительно ниже той, которую позволяет получить однопроцессорная программа с использованием высокопроизводительных библиотек. Резкий скачок производительности на Ultra60 объясняется тем, что после распараллеливания данные как-то очень удачно расположились в кэш-памяти.

Максимального ускорения работы программы можно добиться при использовании параллельной версии подпрограммы перемножения матриц из пакета ScaLAPACK совместно с библиотекой ATLAS. Приведем в таблице 3.7 данные по производительности при выполнении этой программы на 2-х процессорах.

Таблица 3.7. Время решения и производительность при использовании параллельной версии подпрограммы перемножения матриц из библиотеки ScaLAPACK совместно с библиотекой ATLAS.

Решение на 2-х процессорах	Ultra60	Linux	Alpha
время решения (сек.)	1.62	3.7	1.30
производительность (Mflops)	1227.4	541.6	1539.1

Итак, в окончательном варианте мы смогли решить задачу на компьютере Ultra60 за 1.62 сек., или в **161** раз быстрее по сравнению с первоначальным расчетом, а на компьютере Alpha за 1.3 сек., или в **83** раза быстрее. Несколько замечаний следует сказать по поводу Linux-кластера. Двухпроцессорный вариант на нем оказался менее эффективным, чем на других системах, и, по-видимому, нет смысла решать эту задачу на большем числе процессоров. Это связано с медленной коммуникационной средой. Доля накладных расходов на пересылки данных слишком велика по сравнению с вычислительными затратами. Однако, как мы отмечали ранее, эта доля уменьшается при увеличении размеров матрицы. В самом деле, при увеличении параметра N до 4000 с приемлемой эффективностью задача решается даже на 4-х процессорах. Суммарная производительность

на 4-х процессорах составила 1231 Mflops, или по 307 Mflops на процессор, что, конечно, весьма неплохо.

Возможно, материал этого раздела не имеет прямого отношения к проблемам параллельного программирования, однако он достаточно наглядно демонстрирует, что не следует забывать об эффективности каждой ветви параллельной программы. Производители вычислительных систем, как правило, не слишком кривят душой, когда дают данные о пиковой производительности своих систем, но при этом они умалчивают о том, насколько трудно достичь этой производительности. Еще раз напомним, что все эти проблемы связаны со значительным отставанием скорости работы основной памяти от скорости работы процессоров.

Глава 4.
МНОГОПРОЦЕССОРНАЯ ВЫЧИСЛИТЕЛЬНАЯ СИСТЕМА nCUBE2

4.1. Общее описание вычислительной системы

Типичным представителем многопроцессорной системы с массовым параллелизмом (MPP) является суперкомпьютер nCUBE2, состоящий из мультипроцессора nCUBE2 и хост-компьютера, управляющего его работой. Мультипроцессор состоит из набора процессорных модулей (узлов), объединенных в гиперкубовую структуру. В такой структуре процессоры размещаются в вершинах N-мерного куба (*гиперкуба*), а коммуникационные каналы, соединяющие процессоры, расположены вдоль ребер гиперкуба. Общее число процессоров в гиперкубе размерности N равно 2^N. На рис. 4.1 приведены гиперкубовые структуры для различного числа процессоров.

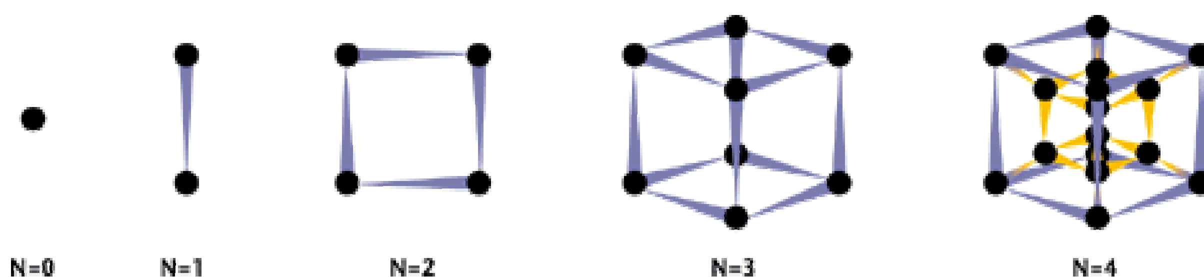

Рис. 4.1. Гиперкубовые структуры для различного числа процессоров.

Гиперкубовая архитектура является одной из наиболее эффективных топологий соединения вычислительных узлов. Основным показателем эффективности топологии многопроцессорной системы является количество шагов, требуемое для пересылки данных между двумя наиболее удаленными друг от друга процессорами. В гиперкубовой архитектуре максимальное расстояние (число шагов) между узлами равно размерности гиперкуба. Например, в системе с 64 процессорами сообщение всегда достигнет адресата не более, чем за 6 шагов. Для сравнения заметим, что в системе с топологией двумерной сетки для передачи данных между наиболее удаленными процессорами требуется 14 шагов. Кроме того, при увеличении количества процессоров в два раза максимальное расстояние между процессорами увеличивается всего на 1 шаг. Очевидно, что для образования такой архитектуры на вычислительных узлах необходимо иметь достаточное количество коммуникационных каналов. В процессорных модулях nCUBE2 имеется 13 таких каналов, что позволяет собирать системы, состоящие из 8192 процессоров.

Физическая нумерация процессоров построена таким образом, что номера соседних узлов в двоичной записи отличаются только одним битом. Номер этого бита однозначно определяет номер коммуникационного канала, соединяющего эти процессоры. Это позволяет эффективно реализовать аппаратные коммутации между любой парой процессоров. *Подкубом* в гиперкубовой архитектуре называют подмножество узлов, которые, в свою очередь, образуют гиперкуб

меньшей размерности. На рис. 4.2 жирными линиями выделены подкубы размерности 1 и 2. Каждая из 8-ми вершин куба образует подкуб размерности 0; четыре подкуба размерности 1 образуют совокупности узлов (0,1), (2,3), (4,5), (6,7); два подкуба размерности 2 образуют совокупности узлов (0,1,3,2) и (4,5,7,6); один подкуб размерности 3 образует вся совокупность 8-ми узлов. Для параллельной программы всегда выделяется набор узлов, образующих подкуб нужной размерности. Это означает, что программа не может быть загружена в набор узлов (1,3,7,5) при заказе 4-х процессоров, поскольку они не принадлежат одному подкубу.

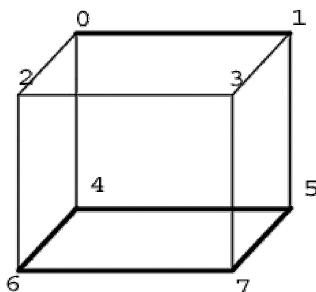

Рис. 4.2. Подкубы размерности 1 и 2 в трехмерном гиперкубе.

Каждый узел в массиве процессоров nCUBE2 состоит из 64-битного центрального процессора, коммуникационного процессора и оперативной памяти. Коммуникационный процессор отвечает за пересылку данных между узлами, освобождая центральный процессор от выполнения рутинных операций по приему, отправке и маршрутизации потока данных. Ниже приведены технические характеристики вычислительного комплекса nCUBE2, установленного в РГУ:

- число процессоров 64
- оперативная память на один процессор (Мб) 32
- число процессоров ввода/вывода 8
- число каналов ввода/вывода 6
- объем дисковых накопителей (Гб) 20
- суммарная пиковая производительность (Mflops) 192

Доступ к вычислительным ресурсам nCUBE2 получают пользователи, зарегистрированные на хост-компьютере, роль которого

выполняет рабочая станция SGI 4D/35 (Silicon Graphics), работающая под управлением операционной системы IRIX 4.0.5. С помощью хост-компьютера выполняется начальная инициализация системы, ее тестирование и подготовка программ для их выполнения на nCUBE2. В программное обеспечение хост-компьютера входит серверная программа (proxy-сервер), позволяющая организовать прямой доступ к вычислительным ресурсам nCUBE2 с *хост-компьютеров второго уровня*, в качестве которых могут выступать рабочие станции SUN. Для этого на них должно быть установлено программное обеспечение хост-компьютера.

На хост-компьютерах устанавливается среда параллельного программирования (Parallel Software Environment – PSE). PSE поставляется в трех вариантах: для операционных систем IRIX 4.0.5, SunOS и Solaris.

4.2. Структура программного обеспечения nCUBE2

Все программное обеспечение вычислительной системы nCUBE2 находится в файловой системе хост-компьютера и содержит следующие компоненты [13].

1. Средства, обеспечивающие выполнение параллельных программ на nCUBE2:
   - операционная система nCX – компактное (~128Kb) оптимизированное UNIX-подобное микроядро, которое загружается в каждый процессор nCUBE2 и в процессоры ввода-вывода во время инициализации системы (по команде nboot);
   - драйвер взаимодействия nCUBE2 с хост-компьютером, функционирующий на интерфейсном модуле хост-компьютера;
   - утилиты системного администрирования, запускаемые на хост-компьютере для начальной инициализации системы, тестирования, контроля состояния и остановки системы;

- набор драйверных программ, загружаемых в процессоры ввода/вывода для обслуживания периферийных устройств;
- набор стандартных UNIX-утилит, исполняемых на nCUBE2 (ls, cp, mkdir и т.д.).

2. Средства подготовки и запуска параллельных программ на nCUBE2:
   - кросс-компиляторы с языков C и FORTRAN77, вызываемые утилитой ncc;
   - загрузчик параллельных программ xnc;
   - профилировщики параллельных программ xtool, ctool, etool, nprof;
   - отладчики параллельных программ ndb и ngdb;
   - набор системных вызовов для загрузки и запуска параллельных slave- (ведомых) программ из master- (головных) программ, исполняющихся либо на хост-компьютере, либо на мультипроцессоре nCUBE2 (семейство функций rexec)
3. Набор библиотек:
   - библиотека параллельных подпрограмм;
   - математические библиотеки, в том числе BLAS;
   - библиотека интерфейса хост-компьютера с nCUBE2.

4.3. Работа на многопроцессорной системе nCUBE2

Технологический цикл разработки параллельных прикладных программ включает:

- подготовку текста программы на языках C или FORTRAN77;
- компиляцию программы;
- запуск на исполнение;
- отладку параллельной программы;
- профилирование программы с целью ее оптимизации.

Подготовка текстов программ производится на хост-компьютере с помощью стандартных редакторов UNIX систем (vi, mc или nedit и xmedit при работе в X-window среде). Текст программы может также готовиться на персональном компьютере с последующей передачей на хост-компьютер с помощью средств ftp. В этом случае передачу файла следует производить в текстовом формате для преобразования из DOS формата в UNIX формат. Если коммуникационная программа такого преобразования не выполняет, то следует воспользоваться утилитами преобразования dos2unix (на SUN), to_unix (на SGI).

Компиляция параллельных программ осуществляется на хост-компьютере с помощью команды ncc, которая представляет собой интерфейс к компилирующим системам с языков C и FORTRAN77. Вызов той или иной системы происходит в зависимости от расширения имени исходного файла ('.c' для программ на языке C и '.f' для программ на языке FORTRAN). Двоичные коды хост-компьютера и многопроцессорной системы nCUBE2 несовместимы между собой.

Синтаксис команды ncc:

```
ncc [[options] sourcefiles [-l libraries]]
```

Примечание: в квадратных скобках здесь и далее будут помечаться необязательные компоненты командной строки.

Наиболее часто используемые опции ncc (далеко не полный перечень):

-c

Запретить фазу редактирования связей и сохранить все полученные объектные файлы.

-g

Генерировать при компиляции дополнительную информацию для отладчика.

-help

Вывести список опций команды ncc. То же при запуске ncc без аргументов.

-L dir

Добавить директорию dir в список директорий, в которых редактор связей nld ищет библиотеки.

-l name

Подключить библиотеку libname.a. Порядок поиска в библиотеках зависит от положения в командной строке других библиотек и объектных файлов.

-O

Оптимизировать объектный код. После окончания отладки программы использовать обязательно. Выигрыш в производительности может быть очень существенный (в 2-3 раза).

-o file

Присвоить исполнимому модулю указанное имя file вместо используемого по умолчанию имени a.out.

-I dir

Добавить директорию dir к стандартному пути поиска include-файлов. В случае нескольких опций –I директории просматриваются в порядке их следования.

-p

Генерировать дополнительную информацию для профилировщика nprof.

-bounds

Проверять границы массивов. При выходе индексов массива из границ программа останавливается и вызывается отладчик.

-d num

Задать размерность подкуба, который по умолчанию будет выделяться программе при запуске, если его размерность не будет явно указана в командной строке. По умолчанию программы компилируются для работы на 0-мерном кубе, то есть на единственном узле.

При отсутствии аргументов в команде ncc выдается подсказка по ее использованию.

Для компиляции программы myprog.c в режиме оптимизации должна быть набрана команда:

```
ncc –O –o myprog  myprog.c
```

В результате будет создан исполнимый файл с именем myprog. Если на приглашение командного интерпретатора хост-компьютера набрать имя созданной программы

```
myprog
```

то она автоматически загрузится в первый свободный процессор nCUBE2.

Команда

ncc –O –d 4 –o myprog myprog.c

создаст исполнимый модуль, который автоматически будет загружаться в первый свободный подкуб размерности 4, т.е. в подкуб из 16 процессоров. Более гибкое управление загрузкой выполняется командой xnc.

Запуск программ на многопроцессорной системе nCUBE2. Программное обеспечение nCUBE2 предоставляет три различных варианта загрузки программ в многопроцессорную систему nCUBE2.

1. Загрузка программы из командного интерпретатора хост-компьютера с помощью команды xnc.
2. Загрузка slave-программы (ведомой) из master-программы (головной), исполняющейся на хост-компьютере, с помощью подпрограмм специальной библиотеки хост-компьютера libncube.a:
   - функция nopen открывает подкуб требуемой размерности
   - функция nodeset специфицирует список процессоров для загрузки
   - функция rexec загружает указанную программу в nCUBE2.

   Примечание: Помимо этих функций в состав библиотеки входят функции, обеспечивающие обмен информацией между master- и slave-программами.
3. Загрузка slave-программы из master-программы, выполняющейся на nCUBE2, с помощью подпрограмм Run-time библиотеки:
   - функция nodeset специфицирует список процессоров для загрузки
   - функция rexec загружает указанную программу в nCUBE2.

Наиболее типичным вариантом запуска программ на многопроцессорной системе nCUBE2 является запуск ее из командного интерпретатора хост-компьютера с использованием загрузчика xnc. В этом

случае вне зависимости от того, для какого числа процессоров была откомпилирована программа, одна и та же копия программы будет загружена в требуемое число процессоров.

Например, команда

    xnc -d 5 prog

загружает программу prog в 5-мерный подкуб (32 процессора).

Синтаксис команды xnc:

    xnc [options] [program [args]]

где
program – имя исполнимого файла;
args – список аргументов, передаваемых программе.

Наиболее важные опции команды xnc:

-d dim
: Выделить подкуб размерности dim. Количество выделяемых процессоров равно 2^{dim} (по умолчанию 0).

-F
: Блокировать вызов отладчика при ошибке во время выполнения.

-g
: Запустить задачу в фоновом режиме.

-l loadfile
: Использовать информацию для загрузчика из файла loadfile. Этот механизм позволяет загружать в различные процессоры подкуба разные программы (реализация режима программирования Multiple Programs Multiple Data – MPMD). Каждую строку файла загрузчик xnc выполняет как отдельную загрузку, но синхронизирует все программы перед их выполнением.

-Ncomm size
: Задать размер коммуникационного буфера size байт; по умолчанию 65,536. Эта опция аннулирует значение Ncomm, установленное командой ncc. ВНИМАНИЕ: это очень важная опция – если размер передаваемого сообщения больше размера буфера, то возможно зависание программы или выход по ошибке.

-Nfile count
: Задать максимальное число файлов count, которые могут быть открыты одновременно (по умолчанию 8). Эта опция аннулирует значение Nfile, установленное в ncc.

-T timeout

Устанавливает время выполнения программы timeout секунд. Если запущенная программа не завершится в течение указанного времени, процесс xnc прекратит ее работу автоматически.

При отсутствии каких-либо аргументов в команде xnc выдается подсказка по использованию этой команды.

Команда xnc загружает исполнимый модуль в первый свободный подкуб запрошенной размерности и передает запросы ввода/вывода системе ввода/вывода хост-компьютера. Следовательно, программе может быть выделено 1, 2, 4, 8, 16 и т.д. процессоров. Для адресации процессоров прикладная программа использует логические номера процессоров в пределах подкуба, а не физические аппаратные адреса. В пределах подкуба процессоры нумеруются с 0. Например, любой 2-мерный подкуб имеет четыре процессора с логическими номерами 0, 1, 2, 3. При этом прикладная программа ограничена выделенным пространством узлов, т.е. не может взаимодействовать с узлами за пределами выделенного подкуба. При отсутствии свободного подкуба запрошенной размерности запрос на запуск программы просто отвергается. После завершения программы процесс xnc автоматически освобождает процессоры, после чего они могут быть выделены для других программ.

Программы, требующие длительного времени для своего выполнения, могут быть запущены в фоновом режиме с перенаправлением ввода/вывода:

```
xnc –d 4 myprog [< inputfile ] > outputfile &
```

где

inputfile – файл с входной информацией;
outputfile – файл для выходной информации;
& – символ, означающий исполнение задания в фоновом режиме.

Вопросы, связанные с загрузкой выполняемых модулей из других программ, т.е. организацию режима master-slave, в данной книге мы рассматривать не будем.

Отладка параллельных программ. Среда параллельного программирования включает два отладчика:

- ndb, интерактивный символьный отладчик, позволяющий отлаживать программы, написанные на языках высокого уровня C и FORTRAN;
- ngdb, реализация для nCUBE2 свободно распространяемого GNU-отладчика GDB, обеспечивает отладку исходных программ, написанных на языке C.

Оба отладчика позволяют:

- проверять значения переменных, регистров процессоров, содержимого стеков;
- отображать данные в различных форматах (целые, восьмеричные, ASCII);
- отображать карту памяти исполняемой программы;
- отображать исходный текст;
- устанавливать точки останова;
- исполнять программу в пошаговом режиме;
- выполнять другие стандартные операции отладки.

Примечание: для работы с отладчиком программа должна быть скомпилирована с опцией -g.

Отладчик ndb вызывается автоматически, если программа при выполнении на nCUBE2 завершается аварийно. После вызова отладчика появляется приглашение на ввод команд. Наиболее полезной командой для определения текущего состояния программы является команда where, которая показывает точку останова, несколько верхних уровней стека вызванных подпрограмм и их аргументы. Для просмотра значений переменных служит команда print. Например, можно выдать на терминал значения переменных X и Y, набрав:

```
ndb> print X, Y
X = 12
Y = 0.145
```

ВНИМАНИЕ: отладчик не работает при запуске программ с хост-компьютеров второго уровня. Для работы с ним программа должна быть запущена с главного хост-компьютера sgi1.rnd.runnet.ru, причем из оболочки csh или sh (не tcsh или bash).

Профилирование параллельных программ позволяет получить детальные сведения о процессе исполнения программы, анализируя которые можно существенно повысить ее производительность. Утилиты профилирования:

- xtool – профилирует выполнение программы, подсчитывая время, которое программа тратит на исполнение каждой из подпрограмм на каждом процессоре.
- ctool – профилирует сообщения программы, следит за трафиком межпроцессорных сообщений и подсчитывает накладные расходы программы на различные виды коммуникационных операций.
- etool – профилирует определенные пользователем события, запоминая время наступления события, тип события, значение указанной переменной программы на момент наступления события.

В процессе профилирования исполняемой программы собранные данные сохраняются в файлах exprof.prx, exprof.prc, exprof.pre.

Для активизации профилировщиков xtool, ctool и etool надо переменной окружения EXPROF_SWITCHES присвоить символьное значение, являющееся любой комбинацией соответственно букв **x, c** и **e**. Например, после исполнения команды

```
setenv EXPROF_SWITCHES xc
```

при запуске любой программы на nCUBE2 будет происходить сбор информации для профилировщиков в файлах exprof.prx и exprof.prc. Эту информацию можно будет проанализировать, запустив один из выбранных профилировщиков:

```
xtool myprog
ctool myprog
```

4.4. Получение информации о системе и управление процессами

Информацию о конфигурации и текущем состоянии nCUBE2 выдают различные хост-резидентные утилиты.

В данном разделе кратко описываются следующие процедуры:

- просмотр активных пользователей в системе nCUBE2;
- просмотр запущенных процессов;
- просмотр количества доступных процессоров в системе nCUBE2;
- удаление процессов.

Для получения более подробной информации о конкретной утилите служит команда:

```
nman    <имя команды>
```

Просмотр активных пользователей в системе nCUBE2 выполняется с помощью команды ***nwho***. Рассмотрим следующий пример:

```
nwho

ncube    I/O node ffff    Dec 09 18:22  sgi1
ncube    I/O node ffff    Dec 09 18:23  sgi1
ncube    I/O node ffff    Dec 09 18:22  sgi1

ncube    I/O node ffff    Dec 09 18:22  sgi1

ncube    I/O node ffff    Dec 09 18:22  sgi1
milton   [0000, 0003]     Dec 10 17:03  sgi1
carlson  [0016, 0031]     Dec 10 17:11  sgi1
```

В данном примере команда nwho показала пять процессов системного пользователя ncube на процессорах ввода/вывода, процесс пользователя milton, исполняемый на 4-х процессорах, и процесс пользователя carlson на 16-ти процессорах.

Команда просмотра запущенных процессов nps выдает аналогичную информацию в расширенном формате, в частности, показывает номер xnc-процесса на хост-компьютере (2-й столбец). Эта информация может потребоваться для удаления запущенного процесса.

nps -au

USER	PID	CHAN	SPID	STAT	NODES	NMIN	NMAX	TMIN	TMAX	START	FROM
ncube	338	3	-	0	0	0	-1	0	65535	Dec 9	sgi1
ncube	312	4	-	0	0	0	-1	0	32767	Dec 9	sgi1
ncube	370	3	-	0	0	0	-1	0	65535	Dec 9	sgi1
ncube	375	3	-	0	0	0	-1	0	65535	Dec 9	sgi1
ncube	376	3	-	0	0	0	-1	0	65535	Dec 9	sgi1
milton	4983	6	-	1	4	0	3	0	65535	17:03	sgi1
carlson	4992	6	-	1	16	16	31	0	65535	17:21	sgi1

5 I/O(s) allocated

Команда nps без параметров показывает только процессы пользователя, выполнившего эту команду.

Просмотр количества доступных процессоров в системе выполняется с помощью команды ***ncube*.** Например, если команда

ncube -n

выдала сообщение

4 node(s) in use out of 64

то это означает, что в данный момент занято 4 процессора из 64-х.

В случае зависания nCUBE2-программы ***прервать ее выполнение*** можно, уничтожив родительский xnc-процесс на хост-компьютере с помощью команды системы UNIX:

kill -9 <идентификатор процесса>

Идентификатор этого ***процесса*** можно посмотреть с помощью команды

nps -au.

Все запущенные пользователем на nCUBE2 ***процессы уничтожаются*** командой **nkill**.

4.5. Средства параллельного программирования на nCUBE2

На многопроцессорной системе nCUBE2 механизм передачи сообщений реализован в виде расширения Run-time библиотеки (подключаемой по умолчанию), в которую включен ряд дополнительных

функций, вызываемых из программ, написанных на языках C и FORTRAN. Список этих функций весьма невелик, но, тем не менее, предоставляет программисту достаточно широкие возможности для создания программ сложной структуры. Возможно программирование как в режиме SPMD, так и в режиме MPMD.

Рассмотрим наиболее важные из этих функций. При обсуждении параметров процедур символами IN будем указывать входные параметры процедур, символами OUT выходные, а INOUT – входные параметры, модифицируемые процедурой.

Подпрограмма идентификации процессором самого себя whoami

Все параметры этой подпрограммы являются выходными. Возвращаются: идентификатор (логический номер) процессора nCUBE2, номер процесса на nCUBE2, номер xnc-процесса на хост-компьютере, размерность выделенного задаче подкуба (общее число процессоров).

C:
```
void whoami (int *node_ID, int *proc, int *host, int *dims)
```
FORTRAN:
```
subroutine whoami (node_ID, proc, host, dims)
integer node_ID, proc, host, dims
```

Здесь:

OUT node_ID – номер процессора в заказанном подкубе, нумерация с 0;
OUT proc – целая переменная, в которую упакован номер процессора и номер процесса;
OUT host – номер процесса на хост-компьютере, используется как адрес для обмена с хост-компьютером;
OUT dims – размерность выделенного программе подкуба.

Функция nwrite посылает сообщение с идентификатором type длиной nbyte из массива buffer процессору dest.

C:
```
int nwrite (char *buffer, int nbytes, int dest, int type, int *flag)
```
FORTRAN:
```
integer function nwrite (buffer, nbytes, dest, type, flag)
dimension buffer (*)
integer nbytes, dest, type, flag
```

IN buffer – адрес первого байта сообщения;
IN nbytes – число передаваемых байт;
IN dest – относительный номер процессора-получателя (адрес 0xffff означает посылку сообщения всем процессорам – broadcast);
IN type – идентификатор сообщения;
flag – не используется.

Функция возвращает код ошибки или 0 в случае успешного завершения.

Функция nread принимает из системного буфера сообщение с идентификатором type от отправителя source длиной nbyte и помещает его в адресное пространство процесса, начиная с адреса buffer. Если используется широковещательный запрос, то на выходе функции параметр source содержит адрес отправителя, а параметр type – идентификатор принятого сообщения.

```
C:
int nread (char *buffer, int nbytes, int *source, int *type, int *flag)
FORTRAN:
integer function nread (buffer, nbytes, source, type, flag)
dimension buffer(*)
integer nbytes, source, type, flag
```

Здесь:

OUT buffer – адрес, куда будет помещаться сообщение;
IN nbytes – число принимаемых байт;
INOUT source – относительный номер процессора, от которого ожидается сообщение (адрес -1 означает прием от любого процессора, на выходе source содержит адрес источника, чье сообщение было принято);
INOUT type – идентификатор ожидаемого сообщения (если -1, то любое сообщение, находящееся в системном буфере, на выходе переменная type будет содержать идентификатор принятого сообщения);
flag – параметр не используется.

Функция возвращает число принятых байт.

Подпрограмма nrange позволяет ограничить диапазон ожидаемых сообщений в случае использования расширенного запроса в операциях чтения (type = -1).

C:

```
void nrange (int low, int high)
```

FORTRAN:

```
subroutine nrange (low, high)
integer  low, high
```

IN low – нижняя граница диапазона (по умолчанию 0);
IN high – верхняя граница диапазона (по умолчанию 32,767).

Функция ntest проверяет наличие в буфере ожидаемого сообщения и возвращает длину сообщения в байтах, если оно получено. В противном случае возвращается отрицательное число.

C:

```
int ntest (int *source, int *type)
```

FORTRAN:

```
integer function ntest (source, type)
integer source, type
```

INOUT source – номер процессора, от которого ожидается сообщение (-1 от любого процессора, в этом случае source на выходе содержит адрес отправителя);

INOUT type – идентификатор ожидаемого сообщения (-1 любое сообщение, в этом случае type на выходе содержит идентификатор принятого сообщения).

Кроме этих базовых функций, PSE поддерживает буферизованный режим обменов:

nwritep – передача сообщения по указателю на буфер;
nreadp – прием сообщения из буфера;
ngetp – резервирует память под буфер и возвращает указатель на него.

C:

```
void* ngetp (unsigned nbytes)
```

IN nbytes – размер буфера в байтах.

Для организации режима MPMD – исполнения разных программ в подкубе с возможностью обмена данными между ними – используются функции:

nodeset – выделение процессоров для новой программы;
rexec – загрузка новой программы в процессоры.

В данной книге этот режим работы рассматриваться не будет, поскольку он слишком специфичен для конкретной платформы.

Помимо базового набора представленных выше функций, в программное обеспечение nCUBE2 входит библиотека элементарных параллельных подпрограмм libnpara.a (при компиляции подключается с помощью опции компилятора -lnpara). Поскольку в подпрограммах библиотек nCUBE2 не специфицируется тип пересылаемых данных, то для каждого типа представлена отдельная функция. Префикс в имени функции соответствует:

i – целый тип 4 байта;
l – целый тип 8 байт;
r – вещественный тип 4 байта;
d – вещественный тип двойной точности 8 байт.

При использовании библиотеки npara к тексту программ должны подключаться include-файлы:

C:

```
#include <ncube/npara_prt.h>
```

FORTRAN:

```
include 'ncube/npara_prt.hf'
```

Таблица 4.1. Список подпрограмм параллельной библиотеки nCUBE.

Назначение	Подпрограммы
Широковещательная рассылка сообщения	nbroadcast
Суммирование скалярных переменных	isum, lsum, rsum, dsum
Суммирование векторных переменных	nisumn, nlsumn, nrsumn, ndsumn
Скалярный максимум	imax, lmax, rmax, dmax
Векторный максимум	nimaxn, nlmaxn, nrmaxn, ndmaxn

Скалярный минимум	imin, lmin, rmin, dmin
Векторный минимум	niminn, nlminn, nrminn, ndminn
Транспонирование двумерных матриц	trans2d8p, trans2d16p
Транспонирование трехмерных матриц	trans3d8p, trans3d16p
Двухмерное быстрое преобразование Фурье	cfft2dp, dcfft2dp
Трехмерное быстрое преобразование Фурье	cfft3dp, dcfft3dp
Упорядочение работы узлов	seqstart, seqend, ntseqstart, ntseqend
Разбиение данных	part1d
Отображение узлов на сетку	nodetogrid, ngridtonode

В качестве примера дадим описание двух функций, которые наиболее часто используются в приложениях.

Функция рассылки сообщения всем процессорам подкуба nbroadcast

Функция позволяет послать всем процессорам данные, расположенные в адресном пространстве узла bnode, начиная с адреса buf. Длина сообщения length байт, идентификатор сообщения type. Принимаемое сообщение размещается в ту же самую переменную buf.

C:
```
        int nbroadcast(buf, length, bnode, type, mask);
        void *buf;
        int length, bnode, type, mask;
```
FORTRAN:
```
        integer*4 nbroadcast(buf, length, bnode, type, mask)
        integer*4 buf(*)
        integer*4 length, bnode, type, mask
```

INOUT	buf	– указатель на буфер, содержащий сообщение;
IN	length	– целое, указывающее длину buf в байтах;
IN	bnode	– относительный номер узла, который выполняет рассылку;
IN	msgtype	– идентификатор сообщения;
IN	mask	– битовая маска каналов, которая указывает, какие узлы текущего подкуба получат сообщение. Если mask равно -1, сообщение получат все узлы.

Функция вычисления глобальной суммы dsum

Функция возвращает сумму локальных значений переменной mypart из процессоров, выборка которых осуществляется согласно маске mask. Результирующее значение суммы получит процессор target.

C:
```
    double dsum (mypart, target, msgtyp, mask)
    double mypart;
    int target, msgtyp, mask;
```
FORTRAN:
```
    real*8 function dsum (mypart, target, msgtyp, mask)
    real*8 mypart
    integer*4 target, msgtyp, mask
```

IN mydata – переменная, содержащая суммируемое значение;
IN target – номер узла, который получит результирующую сумму (если target = -1, результат получат все узлы);
IN msgtype – идентификатор сообщения;
IN mask – битовая маска каналов – указывает, какие узлы текущего подкуба будут участвовать в суммировании. Младший бит маски соответствует каналу 0, следующий бит – каналу 1 и т.д. Биты маски, которые не соответствуют используемым в подкубе каналам, игнорируются. Если mask = -1, участвуют все узлы.

4.6. Библиотека подпрограмм хост-компьютера для взаимодействия с параллельными программами nCUBE2

В большинстве случаев нет необходимости организовывать специальное взаимодействие между хост-компьютером и вычислительными узлами. Все запросы ввода/вывода автоматически транслируются хост-процессом xnc. Однако в программном обеспечении nCUBE2 предусмотрена возможность прямого взаимодействия процессов, выполняющихся на nCUBE2, с процессами хост-компьютера. Это позволяет распределять выполнение задачи между ними. Например, вычислительную часть выполнять на nCUBE2, а графическую обработку результатов динамически выполнять на хост-компьютере. В этом случае головная программа (master) запускается на хост-компьютере, из нее выполняется загрузка и запуск ведомой (slave) вычислительной программы

на nCUBE2. Тогда между этими программами возможен прямой обмен информацией, реализованный по аналогии с обменом данными между процессорами. Для обеспечения этого режима в программное обеспечение хост-компьютера добавлена библиотека libncube.a. В ее состав, в частности, входят функции:

nopen – запрос подкуба на nCUBE2;
nodeset – создание дескриптора для задаваемого набора процессоров;
rexec – загрузка программы в процессоры nCUBE2;
nread – чтение сообщения на хост-компьютере из параллельной программы;
nwrite – посылка сообщения от хост-компьютера параллельной программе;
nclose – освобождение запрошенного подкуба.

В данной работе мы не будем детально рассматривать эти функции, поскольку режим master-slave на сегодня утратил практический смысл. В самом деле, если объемы вычислений таковы, что они допускают интерактивную обработку результатов, то такие задачи разумнее решать на современных графических рабочих станциях.

4.7. Пример параллельной программы с использованием средств PSE

В заключение главы рассмотрим классический пример программы вычисления числа π на многопроцессорной системе nCUBE2 с использованием стандартных средств разработки параллельных программ PSE. Для расчета используем формулу:

$$\pi = 4 * \int_0^1 dx/(1 + x * x) \qquad (4.1)$$

Интегрирование будем выполнять методом трапеций. Для получения точности 10^{-8} необходимо интервал разбить на 10^6 частей. Для начала приведем текст обычной последовательной программы и посмотрим, каким образом ее нужно модифицировать, чтобы получить параллельную версию.

```
c   numerical integration to calculate pi (sequential program)
      program calc_pi
      integer i, n
      double precision w, sum
      double precision v
      integer np
      real*8 time, mflops, time1, dsecnd
c  Вводим число точек разбиения интервала
      print *, 'Input number of stripes : '
      read *, n
      np = 1
c  Включаем таймер
      time1 = dsecnd()

      w = 1.0 / n
      sum = 0.0d0
c  Основной цикл интегрирования
      do i = 1, n
        v = (i - 0.5d0) * w
        v = 4.0d0 / (1.0d0 + v * v)
        sum = sum + v
      end do
c  Фиксируем время, затраченное на вычисления
      time   = dsecnd() - time1
c  Подсчитываем производительность компьютера в Mflops
      mflops = 6 * n / (1000000.0 * time)
      print *, 'pi is approximated with ', sum *w
      print *, 'time   = ', time, ' seconds'
      print *, 'mflops = ', mflops, ' on ', np, ' processors'
      print *, 'mflops = ', mflops/np, ' for one processor'
      end
```

Это классическая легко распараллеливающаяся задача. В самом деле, операторы внутри цикла образуют независимые подзадачи, поэтому мы можем распределить выполнение этих подзадач на некоторый набор процессоров. Аддитивность операции суммирования позволяет разбить ее выполнение на вычисление частичных сумм на каждом процессоре с последующим суммированием этих частичных сумм. Параллельная версия этой программы с использованием средств PSE выглядит следующим образом:

```
c  жирным шрифтом выделены изменения в программе
c  numerical integration to calculate pi (parallel program)
      program calc_pi
      include 'ncube/npara_prt.hf'   ! include-файл libnpara.a
      integer i, n
      double precision w, gsum, sum
      double precision v
```

```
      integer np, myid, ierr, proc, j, dim, msgtype, mask
      real*8 time, mflops, time1, dsecnd
c  Вызов функции идентификации процессора myid и опроса
c  размерности заказанного подкуба dim
      call whoami(myid, proc, j, dim)
c  Операцию чтения с клавиатуры выполняет только 0-й процессор
      if ( myid .eq. 0 ) then
      print *, 'Input number of stripes : '
      read *, n
      time1 = dsecnd()
      endif
c  Установка переменных для функции nbroadcast
      msgtype = 1
      mask = -1
c  Рассылка параметра n всем процессорам
      ierr = nbroadcast(n, 4, 0, msgtype, mask)
c  Подсчет заказанного числа процессоров
      np = 2**dim

      w = 1.0 / n
      sum = 0.0d0
c  Вычисление частичной суммы на  каждом процессоре
      do i = myid+1, n, np
        v = (i - 0.5d0 ) * w
        v = 4.0d0 / (1.0d0 + v * v)
        sum = sum + v
      end do
c  Установка переменных для функции dsum
      msgtype = 2
      mask = -1
      node = 0
c  Суммирование частичных сумм с сохранением результата на 0-м
c  процессоре
      gsum = dsum(sum, node, msgtype, mask)
c  Вывод информации производит только 0-ой процессор
      if (myid .eq. 0) then
      time   = dsecnd() - time1
      mflops = 6 * n / (1000000.0 * time)
      print *, 'pi is approximated with ', gsum *w
      print *, 'time   = ', time, ' seconds'
      print *, 'mflops = ', mflops, ' on ', np, ' processors'
      print *, 'mflops = ', mflops/np, ' for one processor'
      endif
      end
```

Средства разработки параллельных программ, входящие в состав PSE, являются типичными для многопроцессорных систем. Те или иные аналоги рассмотренных выше функций можно найти в составе программного обеспечения любой многопроцессорной системы. Как правило, производители многопроцессорных систем включают в

программное обеспечение различные варианты коммуникационных библиотек.

Общепризнанным стандартом такой коммуникационной библиотеки сегодня по праву считается MPI, поскольку о его поддержке объявили практически все фирмы-производители многопроцессорных систем и разработчики программного обеспечения для них. Подробное рассмотрение функциональности этой библиотеки приводится в части 2 настоящей книги. Существует несколько бесплатно распространяемых реализаций этой коммуникационной библиотеки:

MPICH – разработка Argonne National Laboratory;
LAM – разработка Ohio Supercomputer Center (входит в дистрибутив Linux);
CHIMP/MPI – разработка Edinburgh Parallel Computing Centre.

На многопроцессорной системе nCUBE2 установлена свободно распространяемая реализация этой библиотеки MPICH, которую мы настоятельно рекомендуем в качестве основного средства разработки параллельных программ. Использование этой коммуникационной библиотеки позволяет:

- разрабатывать платформенно-независимые программы;
- использовать существующие стандартные библиотеки подпрограмм.

Опыт работы с этой библиотекой на nCUBE2 показал, что, во-первых, практически не происходит потери производительности, и, во-вторых, не требуется никакой модификации программы при переносе на другую многопроцессорную систему (Linux-кластер, 2-х процессорную Alpha DS20E). Технология работы с параллельной программой на nCUBE2 остается той же самой – единственное отличие состоит в том, что при компиляции программы необходимо подключить MPI библиотеку:

```
ncc -O -o myprog myprog.c –lmpi
```

Библиотека MPICH является также основным средством параллельного программирования на высокопроизводительном вычислительном кластере.

Глава 5.
ВЫСОКОПРОИЗВОДИТЕЛЬНЫЙ ВЫЧИСЛИТЕЛЬНЫЙ КЛАСТЕР

5.1. Архитектура вычислительного кластера

В разделе 1.4 отмечалось, что наиболее доступный способ создания вычислительных ресурсов суперкомпьютерного класса предоставляют кластерные технологии. В простейшем случае с помощью сетевых коммуникационных устройств объединяются однотипные компьютеры, и в их программное обеспечение добавляется коммуникационная библиотека типа MPI. Это позволяет использовать набор компьютеров как единый вычислительный ресурс для запуска параллельных программ. Однако вряд ли такую систему можно назвать полноценным кластером. Очень скоро обнаруживается, что для эффективной эксплуатации такой системы совершенно необходима некоторая диспетчерская система, которая бы распределяла задания пользователей по вычислительным узлам и блокировала бы запуск других заданий на занятых узлах. Существует достаточно большое число таких систем, как коммерческих, так и бесплатно распространяемых. Причем большинство из них не требует идентичности вычислительных узлов. В РГУ создан гетерогенный вычислительный кластер на базе диспетчерской системы Open PBS [14].

В состав высокопроизводительного вычислительного кластера в настоящее время входят:

1. **Compaq Alpha DS20E** – 2-х процессорная SMP система с общей памятью объемом 1 Гб и объемом дискового пространства 2×18 Гб. Пиковая производительность одного процессора 1 Gflops. На однопроцессорном тесте LINPACK достигается производительность 190 Mflops без использования специальных библиотек и 800 Mflops при использовании оптимизированной библиотеки CXML. На параллельной MPI-версии теста LINPACK получена производительность 1.6 Gflops на двух процессорах.

2. **SUN Ultra 60** – 2-х процессорная SMP система с общей памятью объемом 1 Гб и объемом дискового пространства 18 Гб. Пиковая производительность одного процессора 800 Mflops. На однопроцессорном тесте LINPACK достигается производительность 85 Mflops без использования специальных библиотек и 470 Mflops при использовании оптимизированных библиотек LAPACK+ATLAS. На параллельной MPI-версии теста LINPACK получена производительность 0.8 Gflops на двух процессорах. Компьютер выполняет также функции сетевой информационной службы (NIS) для авторизации пользователей и файлового сервера (NFS), экспортирующего домашние директории пользователей на все другие вычислительные системы (nCUBE2, Linux-кластер, Alpha DS20E). Используется также как хост-компьютер для работы с nCUBE2.
3. **Linux-кластер** – вычислительная система из 10 узлов, соединенных сетью Fast Ethernet через коммутатор Cisco Catalyst 2924. Каждый из узлов представляет собой компьютер Pentium III 500 Мгц с 256 Мб оперативной памяти и жестким диском емкостью 10 Гб. Возможно исполнение как однопроцессорных, так и параллельных программ. В качестве основного средства параллельного программирования используется библиотека MPI. Пиковая производительность каждого процессора 500 Mflops. На однопроцессорном тесте LINPACK достигается производительность 44 Mflops без использования специальных библиотек и 300 Mflops при использовании оптимизированных библиотек MKL+ATLAS. На параллельной MPI-версии теста LINPACK получена производительность 2.5 Gflops для всего кластера в целом.

Система сконфигурирована таким образом, что на каждом из 14 процессоров может выполняться не более одного счетного процесса. Таким образом, одновременно может обрабатываться не более 14 обычных

однопроцессорных программ. Для параллельных программ не поддерживается механизм выполнения одной программы на процессорах разной архитектуры. Пользователь должен заранее определиться, на какой из 3-х перечисленных выше архитектур должна быть выполнена его программа, и поставить ее в соответствующую очередь. Таким образом, на системах Alpha и SUN одна программа может использовать не более двух процессоров, а на Linux-кластере – не более 10. При отсутствии необходимых ресурсов в момент запуска задания оно ставится в очередь до освобождения требуемых ресурсов.

На компьютерах Linux-кластера установлена свободно распространяемая операционная система Linux RedHat 6.2, а на компьютерах Alpha и SUN – лицензионные ОС фирм-производителей: Tru64 Unix 4.0F и Solaris 2.7 соответственно. Прикладное программное обеспечение всех вычислительных систем в максимально возможной степени унифицировано для обеспечения переносимости программ на уровне исходных текстов. Это было достигнуто установкой на всех системах коммуникационной библиотеки MPI, системы компиляции программ с языка HPF, базовой библиотеки линейной алгебры ATLAS, однопроцессорной и параллельной версий библиотеки LAPACK, параллельной версии библиотеки для решения систем линейных алгебраических уравнений с разреженными матрицами Aztec (рассматривается в части 3). Это позволило пользователям компилировать исходные тексты программ на любой из перечисленных вычислительных систем без какой-либо их модификации.

В качестве коммуникационной среды на Linux-кластере используется 100-мегабитная сеть Fast Ethernet с коммутатором Cisco Catalyst 2900. Это обеспечивает пиковую скорость передачи данных 10 Мб/сек между любыми парами узлов. На 2-х процессорных системах Alpha и SUN в качестве коммуникационной среды MPI используется разделяемая память, что обеспечивает значительно большую скорость обмена между

параллельными процессами, а именно, 260 Мб/сек и 80 Мб/сек соответственно.

Домашние директории пользователей расположены на компьютере SUN Ultra 60, выполняющего функции NFS сервера, и с которого они экспортируются на все машины высокопроизводительного вычислительного кластера. Учитывая, что этот компьютер является также хост-компьютером многопроцессорной системы nCUBE2, это позволяет работать с любой высокопроизводительной системой без пересылки и дублирования файлов исходных текстов программ. Одна и та же программа без какой-либо модификации может быть откомпилирована для любой из высокопроизводительных систем. Разумеется, компиляция должна выполняться на той системе, для которой изготавливается исполнимый модуль. Для nCUBE2 рекомендуется выполнять компиляцию на компьютере SUN, а для Linux-кластера – на любом из компьютеров кластера (rsucl0, rsucl1, ... , rsucl9).

Для компиляции программ можно использовать стандартные команды вызова компиляторов, но при этом необходимо учитывать, что компилятор должен найти пути к коммуникационной библиотеке и необходимым include-файлам. На Linux-кластере и компьютере SUN для компиляции программ используются специальные команды:

mpif77 -O -o program program.f – для программ на языке FORTRAN77
mpicc -O -o program program.c – для программ на языке C,

которые представляют собой командные файлы, выполняющие все необходимые настройки для поиска требуемых include-файлов и библиотеки MPI.

Для облегчения процесса компиляции были написаны универсальные файлы Makefile, в которых пользователям для перекомпиляции программы достаточно только изменить название архитектуры, для которой должен быть создан исполнимый файл. Ниже представлен образец универсального Makefile для компиляции

фортрановских программ. В данном случае подразумевается, что исполнимая программа будет собираться из головного исходного файла testaz.f и двух файлов с подпрограммами sub1.f и sub2.f. Помимо коммуникационной библиотеки подключается математическая библиотека Aztec. Первая строка указывает, что компиляция будет выполняться для Linux-кластера (возможные значения переменной MACHINE: SUN, ALPHA, NCUBE, LINUX).

```
MACHINE = LINUX
PROG    =  testaz
OBJ     =  $(PROG).o  sub1.o  sub2.o

FC_SUN          = mpif77
FC_ALPHA        = f77
FC_NCUBE        = ncc
FC_LINUX        = mpif77

FFLAGS_SUN              = -fast -O4 -I/usr/local/include
FFLAGS_ALPHA    = -fast -O4 -I/usr/local/include
FFLAGS_NCUBE    = -O
FFLAGS_LINUX    = -O -I/usr/local/include

LDFLAGS_SUN         = -fast -O4
LDFLAGS_ALPHA       = -fast -O4
LDFLAGS_NCUBE               = -O -Ncomm 3000000
LDFLAGS_LINUX               = -O

LIB_SUN         = -laztec -lmpich -lsocket -lnsl
LIB_ALPHA               = -laztec -lmpi
LIB_NCUBE               = -laztec -lmpi -lblasn -lf -lm
LIB_LINUX       = -laztec -lmpi

F77         = $(FC_${MACHINE})
FFLAGS   = $(FFLAGS_${MACHINE})
LIBS        = $(LIB_${MACHINE})
LDFLAGS = $(LDFLAGS_${MACHINE})

all: exe
exe: $(TESTOBJ)
      ${F77}  ${LDFLAGS} -o $(PROG)_${MACHINE} ${OBJ}
${LIBS}
clean :
      rm -f *.o
.f.o : ; $(F77) -c ${FFLAGS} $*.f
.c.o : ; $(CC) -c $(CCFLAGS) $(CDEFS) $*.c
```

После запуска команды make на одной из машин кластера будет создан исполнимый файл с именем testaz_LINUX. Если переменной MACHINE присвоить значение NCUBE и запустить команду make на компьютере SUN, то будет создан исполнимый файл testaz_NCUBE для многопроцессорной системы nCUBE2. По аналогии легко составить Makefile для компиляции программ, написанных на языках C и C++.

Запуск параллельных MPI-программ на исполнение выполняется с помощью команды:

```
mpirun -np N program
```

где N – заказываемое число процессоров (компьютеров).

ВНИМАНИЕ: Не допускается прямой запуск программ на выполнение с помощью команды mpirun. Эта команда может использоваться только в командных файлах диспетчерской системы пакетной обработки заданий OpenPBS, установленной на высокопроизводительном вычислительном кластере.

5.2. Система пакетной обработки заданий

Основное назначение системы пакетной обработки заданий состоит в запуске программы на исполнение на тех узлах кластера, которые в данный момент не заняты обработкой других заданий, и в буферизации задания, если в данный момент отсутствуют свободные ресурсы для его выполнения. Большинство подобных систем предоставляют и множество других полезных услуг. В полной мере это относится и к рассматриваемой системе OpenPBS.

Важнейшим достоинством системы OpenPBS является достаточно удобная и эффективная поддержка вычислительных узлов разной конфигурации и архитектуры. Система OpenPBS обеспечивает управление выполнением заданий на широком наборе конфигураций вычислительных узлов:

- на рабочих станциях с режимом разделения времени между задачами;
- на многопроцессорных системах как SMP, так и MPP архитектур;
- на кластерных системах с одним или несколькими процессорами на вычислительных узлах;
- на произвольных комбинациях перечисленных выше систем.

Кроме этого, следует отметить простоту и удобство работы с диспетчерской системой как для администратора системы, так и для конечных пользователей.

Для администратора диспетчерской системы предоставляются широкие возможности динамического изменения параметров системы, таких как создание и уничтожение очередей, подключение и отключение вычислительных узлов, установку и изменение предельных лимитов для пользователей, установку лимитов по умолчанию и т.д. Каждому вычислительному узлу может быть присвоен некоторый набор свойств, и тогда задание может быть послано на выполнение на те узлы, которые удовлетворяют необходимому набору свойств. Поэтому представлялось разумным разделить вычислительные узлы по их архитектуре и для каждой из них создать отдельную очередь с именами ALPHA, SUN и LINUX. Многопроцессорная система nCUBE2 используется в основном в учебном процессе и для отладки программ и поэтому диспетчерской системой не обслуживается. К достоинствам системы PBS следует отнести наличие удобных средств оперативного контроля состояния очередей, состояния вычислительных узлов, а также наличие системы регистрации выполненных заданий.

В настоящее время в системе OpenPBS установлены два предельных лимита и их значения по умолчанию:

- максимальное число заказываемых процессоров для архитектур ALPHA и SUN – 2, для кластера – 10 (значение по умолчанию 1);

- максимальное время решения – 168 часов (значение по умолчанию 1 час).

Если в заказе превышен предельный лимит, то задание отвергается. По истечении заказанного времени решения задача автоматически снимается со счета.

Для конечных пользователей работа через систему OpenPBS сводится к запуску программ с помощью специальной команды qsub, в качестве аргументов которой передаются имя очереди, в которую должно быть поставлено задание, и имя запускающего командного файла (скрипта):

```
qsub -q  ARCH  script_name
```

где

ARCH – имя очереди, в которую ставится задание (возможные значения ALPHA, SUN, LINUX)

script_name – имя командного файла, который может быть создан с помощью любого текстового редактора.

В этом командном файле помимо команды запуска программы указываются требуемые задаче ресурсы: количество и архитектура процессоров, время решения задачи, переменные окружения и др. В простейшем случае для запуска однопроцессорной программы на Linux-кластере нужно сформировать файл (например, с именем lpbs1) следующего содержания:

```
#!/bin/sh
###PBS script to run simple program
#PBS -l nodes=1:LINUX
cd $HOME
cd prog/parallel/aztec
./progname
```

Конструкции #PBS распознаются командой qsub и устанавливают лимиты для задачи. В данном случае будет запущена однопроцессорная программа с именем progname на любом свободном узле Linux-кластера. Максимальное время решения задачи 1 час устанавливается по умолчанию. При отсутствии свободных узлов требуемой архитектуры

задание будет поставлено в очередь. В данном случае запуск программы progname должен быть произведен командой:

qsub -q LINUX lpbs1

Для запуска параллельной программы на 4-х процессорах Linux-кластера используется тот же самый формат команды, но содержимое скрипта должно быть другим (скрипт lpbs4)

```
#!/bin/sh
###PBS script to run parallel program
#PBS -l walltime=03:00:00
#PBS -l nodes=4:LINUX
cd $HOME
cd prog/parallel/aztec
mpirun -np 4 progname
```

Здесь заказано время счета 3 часа на 4-х узлах Linux-кластера.

Запуск выполняется командой:

qsub -q LINUX lpbs4

Несколько по-другому выполняется заказ процессоров на 2-х процессорных SMP системах Alpha и SUN. Они трактуются как один узел с двумя процессорами, поэтому для запуска 2-х процессорной задачи на архитектуре SUN скрипт должен иметь вид:

```
#!/bin/sh
###PBS script to run parallel program on SUN
#PBS -l walltime=10:00
#PBS -l nodes=1:ppn=2:SUN
cd $HOME
cd prog/parallel/aztec
mpirun -np 2 progname
```

Здесь строка #PBS -l nodes=1:ppn=2:SUN указывает, что используется 1 узел с двумя процессорами архитектуры SUN. Аналогичный скрипт требуется и для компьютера Alpha, с заменой названия архитектуры SUN на ALPHA. Для однопроцессорных программ скрипт должен иметь вид:

```
#!/bin/sh
###PBS script to run simple program
#PBS -l walltime=30:00
#PBS -l nodes=1:ppn=1:ALPHA
```

```
cd $HOME
cd prog/parallel/aztec
./progname
```

Замечание: На компьютере Alpha, если программа использует библиотеку MPI, то для запуска ее даже на одном процессоре следует использовать команду dmpirun. Т.е. скрипт должен иметь вид:

```
#!/bin/sh
###PBS script to run  MPI program on ALPHA
#PBS -l walltime=30:00
#PBS -l nodes=1:ppn=1:ALPHA
cd $HOME
cd prog/parallel/aztec
dmpirun -np 1 progname
```

Команда dmpirun – это системная команда для запуска MPI-программ в ОС Tru64 Unix.

Специальный вид должен иметь скрипт для запуска параллельных SMP программ, использующих механизм многопоточности (multithreading). Для двухпроцессорной системы Sun Ultra 60 скрипт выглядит следующим образом:

```
#!/bin/sh
#PBS -l walltime=1:00:00
#PBS -l nodes=1:ppn=2:SUN
#PBS -v PARALLEL=2
cd /export/home/victor/prog/bench/misc/sun
./mxmr2
```

При запуске программы через команду qsub заданию присваивается уникальный целочисленный идентификатор, который представляет собой номер запущенного задания. По этому номеру можно отслеживать прохождение задания, снять задание со счета или из очереди или переместить его в очереди относительно своих других заданий. Для этого помимо команды qsub пользователю предоставляется целый набор вспомогательных команд. Приведем их список (табл. 5.1) без подробного описания (его можно посмотреть либо с помощью команды man, либо на WWW сервере http://rsusu1.rnd.runnet.ru/opbs/contents.html):

Таблица 5.1. Список команд системы пакетной обработки заданий OpenPBS.

qdel	удаление задания;
qhold	поставить запрет на исполнение задания;
qmove	переместить задание;
qmsg	послать сообщение заданию;
qrls	убрать запрет на исполнение, установленный командой qhold;
qselect	выборка заданий;
qsig	посылка сигнала (в смысле ОС UNIX) заданию;
qstat	выдача состояния очередей (наиболее полезны команды qstat –a и qstat -q);
qsub	постановка задания в очередь;
pestat	выдача состояния всех вычислительных узлов;
xpbs	графический интерфейс для работы с системой OpenPBS;
xpbsmon	графическая программа выдачи состояния вычислительных ресурсов.

Пакетная обработка заданий предполагает, что программа не должна быть интерактивной, т.е. программа не должна содержать ввода с клавиатуры. Ввод информации в программу возможен только из файлов. Как правило, для этого в программе используются специальные операторы чтения из файлов, но можно также использовать стандартные операторы в сочетании с механизмом перенаправления ввода из файла. В этом случае команда запуска программы в PBS-скрипте должна иметь вид:

```
mpirun –np 4 progname < datfile
```

Что касается вывода информации, то она буферизуется в специальном системном буфере и переписывается в рабочий каталог пользователя по окончании решения задачи. Имя выходного файла формируется автоматически следующим образом:

<имя скрипта>.o<номер задания>,

а в файл

<имя скрипта>.e<номер задания>

будет записываться стандартный канал диагностики (ошибок).

Имя выходного файла можно изменить с помощью специальной опции команды qsub.

Тем не менее, существует возможность просмотра результатов выполнения программы, запущенной в пакетном режиме, в процессе ее решения. Для этого можно использовать механизм перенаправления выдачи в файл. В PBS скрипте в строке запуска программы записывается конструкция:

```
./progname > out.dat
```

Здесь out.dat – имя файла, в который будет перенаправляться информация, выдаваемая на терминал. Тогда после запуска программы в пакетном режиме можно набрать на терминале команду:

```
tail -f out.dat ,
```

которая будет выдавать на терминал информацию по мере ее записи в файл. Прервать просмотр без всякого ущерба для задачи можно, нажав клавиши Ctrl C (но не Ctrl Z). Точно так же в любой момент можно возобновить просмотр. Кроме того, периодически просматривать накопленные результаты можно будет с помощью команды cat.

Система пакетной обработки заданий OpenPBS не препятствует выдаче на экран в интерактивном режиме и графической информации. Более того, графическая информация может быть выдана на экран любого компьютера (не обязательно того, с которого запущено задание). Однако для этого должны быть соблюдены два условия:

1. На компьютере, на экран которого будет выдаваться графическая информация, должен быть запущен X-сервер в момент начала выполнения программы.
2. Должен быть разрешен доступ к X-серверу с того компьютера, на котором выполняется задание.

При несоблюдении этих условий выполнение программы будет прекращено. Для того, чтобы организовать такой режим работы, в программу должна быть передана переменная окружения DISPLAY, указывающая адрес экрана, куда должна выдаваться графическая

информация. Делается это добавлением в PBS скрипт специальной строки, например:

```
#PBS -v DISPLAY=rsusu2.cc.rsu.ru:0.0
```

Здесь rsusu2.cc.rsu.ru – имя компьютера, на экран которого будет выдаваться графическая информация.

Некоторые пакеты и библиотеки графических подпрограмм позволяют выдавать результаты в графические файлы (GIF, PS и т.д.). Разумеется, такой режим возможен только для статических изображений, однако это избавляет от необходимости соблюдения условий 1 и 2. В целях унификации программного обеспечения графические пакеты по возможности устанавливаются на все вычислительные системы, однако в наиболее полном объеме они поддерживаются на компьютере SUN (rsusu2.rnd.runnet.ru).

ЗАКЛЮЧЕНИЕ К ЧАСТИ 1

Итак, в первой части настоящей книги рассмотрены вопросы, касающиеся архитектуры современных многопроцессорных систем и средств разработки параллельных программ. Основное внимание уделено описанию многопроцессорной системы nCUBE2. Это связано с двумя обстоятельствами: во-первых, nCUBE2 представляет собой пример классической многопроцессорной системы MPP архитектуры, и, во-вторых, на сегодняшний день в РГУ эта система остается основным средством для освоения технологий параллельного программирования. Несмотря на то, что элементную базу этого компьютера трудно назвать современной (как и обусловленную этой базой производительность), его программное обеспечение вполне соответствует уровню сегодняшнего дня (с учетом того, что оно дополнено такими современными и универсальными пакетами как MPI, HPF, библиотеками параллельных подпрограмм ScaLAPACK, Aztec и др.). Учитывая высокую степень надежности этой многопроцессорной системы и удобство работы с ней,

безусловно, она представляет собой вполне современное средство для подготовки и отладки параллельных программ, которые затем могут быть перекомпилированы для любой многопроцессорной системы.

Особое место в параллельном программировании в настоящее время занимает среда параллельного программирования MPI. С учетом этого обстоятельства в следующей части представлено достаточно подробное описание этой коммуникационной библиотеки.

Часть 2.
СРЕДА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ MPI

Коммуникационная библиотека MPI стала общепризнанным стандартом в параллельном программировании с использованием механизма передачи сообщений. Полное и строгое описание среды программирования MPI можно найти в авторском описании разработчиков [15, 16]. К сожалению, до сих пор нет перевода этого документа на русский язык. Предлагаемое вниманию читателя описание MPI не является полным, однако содержит значительно больше материала, чем принято представлять во введениях в MPI. Цель данного пособия состоит в том, чтобы, во-первых, ознакомить читателя с функциональными возможностями этой коммуникационной библиотеки и, во-вторых, рассмотреть набор подпрограмм, достаточный для программирования любых алгоритмов. Примеры параллельных программ с использованием коммуникационной библиотеки MPI, приведенные в конце данной части, протестированы на различных многопроцессорных системах (nCUBE2, Linux-кластере, 2-х процессорной системе Alpha DS20E).

Глава 6.
ОБЩАЯ ОРГАНИЗАЦИЯ MPI

MPI-программа представляет собой набор независимых процессов, каждый из которых выполняет свою собственную программу (не обязательно одну и ту же), написанную на языке C или FORTRAN. Появились реализации MPI для C++, однако разработчики стандарта MPI за них ответственности не несут. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных процедур. Как правило, каждый процесс выполняется в своем собственном адресном пространстве, однако допускается и режим разделения памяти.

MPI не специфицирует модель выполнения процесса – это может быть как последовательный процесс, так и многопотоковый. MPI не предоставляет никаких средств для распределения процессов по вычислительным узлам и для запуска их на исполнение. Эти функции возлагаются либо на операционную систему, либо на программиста. В частности, на nCUBE2 используется стандартная команда xnc, а на кластерах – специальный командный файл (*скрипт*) mpirun, который предполагает, что исполнимые модули уже каким-то образом распределены по компьютерам кластера. Описываемый в данной книге стандарт MPI 1.1 не содержит механизмов динамического создания и уничтожения процессов во время выполнения программы. MPI не накладывает каких-либо ограничений на то, как процессы будут распределены по процессорам, в частности, возможен запуск MPI-программы с несколькими процессами на обычной однопроцессорной системе.

Для идентификации наборов процессов вводится понятие *группы*, объединяющей все или какую-то часть процессов. Каждая группа образует *область связи*, с которой связывается специальный объект – *коммуникатор* области связи. Процессы внутри группы нумеруются целым числом в диапазоне 0..groupsize-1. Все коммуникационные операции с некоторым коммуникатором будут выполняться только внутри области связи, описываемой этим коммуникатором. При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор MPI_COMM_WORLD. В большинстве случаев на каждом процессоре запускается один отдельный процесс, и тогда термины процесс и процессор становятся синонимами, а величина groupsize становится равной NPROCS – числу процессоров, выделенных задаче. В дальнейшем обсуждении мы будем понимать именно такую ситуацию и не будем очень уж строго следить за терминологией.

Итак, если сформулировать коротко, MPI – это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Это достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

- функции инициализации и закрытия MPI-процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Набор функций библиотеки MPI далеко выходит за рамки набора функций, минимально необходимого для поддержки механизма передачи сообщений, описанного в первой части. Однако сложность этой библиотеки не должна пугать пользователей, поскольку, в конечном итоге, все это множество функций предназначено для облегчения разработки эффективных параллельных программ. В конце концов, пользователю принадлежит право самому решать, какие средства из предоставляемого арсенала использовать, а какие нет. В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI-функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций [11].

Каждая из MPI функций характеризуется способом выполнения:

1. *Локальная функция* – выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.
2. *Нелокальная функция* – для ее завершения требуется выполнение MPI-процедуры другим процессом.
3. *Глобальная функция* – процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.

4. *Блокирующая функция* – возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.
5. *Неблокирующая функция* – возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Использование библиотеки MPI имеет некоторые отличия в языках C и FORTRAN.

В языке C все процедуры являются функциями, и большинство из них возвращает код ошибки. При использовании имен подпрограмм и именованных констант необходимо строго соблюдать регистр символов. Массивы индексируются с 0. Логические переменные представляются типом int (true соответствует 1, а false – 0). Определение всех именованных констант, прототипов функций и определение типов выполняется подключением файла mpi.h. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках. В таблице 6.1 приведено соответствие предопределенных в MPI типов стандартным типам языка C.

Таблица 6.1. Соответствие между MPI-типами и типами языка C.

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

В языке FORTRAN большинство MPI-процедур являются подпрограммами (вызываются с помощью оператора CALL), а код ошибки возвращают через дополнительный последний параметр процедуры. Несколько процедур, оформленных в виде функций, код ошибки не возвращают. Не требуется строгого соблюдения регистра символов в именах подпрограмм и именованных констант. Массивы индексируются с 1. Объекты MPI, которые в языке C являются структурами, в языке FORTRAN представляются массивами целого типа. Определение всех именованных констант и определение типов выполняется подключением файла mpif.h. В таблице 6.2 приведено соответствие предопределенных в MPI типов стандартным типам языка FORTRAN.

Таблица 6.2. Соответствие между MPI-типами и типами языка FORTRAN.

Тип MPI	**Тип языка FORTRAN**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

В таблицах 6.1 и 6.2 перечислен обязательный минимум поддерживаемых стандартных типов, однако, если в базовой системе представлены и другие типы, то их поддержку будет осуществлять и MPI, например, если в системе есть поддержка комплексных переменных двойной точности DOUBLE COMPLEX, то будет присутствовать тип MPI_DOUBLE_COMPLEX. Типы MPI_BYTE и MPI_PACKED используются для передачи двоичной информации без какого-либо преобразования. Кроме того, программисту предоставляются средства создания собственных типов на базе стандартных (глава 10).

Изучение MPI начнем с рассмотрения базового набора из 6 функций, образующих минимально полный набор, достаточный для написания простейших программ.

Глава 7.
БАЗОВЫЕ ФУНКЦИИ MPI

Любая прикладная MPI-программа (приложение) должна начинаться с вызова функции инициализации MPI – функции MPI_Init. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. Кроме этого создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса.

Синтаксис ***функции инициализации MPI_Init*** значительно отличается в языках C и FORTRAN:

C:

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

```
FORTRAN:
	MPI_INIT(IERROR)
	INTEGER IERROR
```

В программах на C каждому процессу при инициализации передаются аргументы функции main, полученные из командной строки. В программах на языке FORTRAN параметр IERROR является выходным и возвращает код ошибки.

Функция завершения MPI-программ MPI_Finalize

```
C:
	int MPI_Finalize(void)
FORTRAN:
	MPI_FINALIZE(IERROR)
	INTEGER IERROR
```

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Функция определения числа процессов в области связи MPI_Comm_size

```
C:
	int MPI_Comm_size(MPI_Comm comm, int *size)
FORTRAN:
	MPI_COMM_SIZE(COMM, SIZE, IERROR)
	INTEGER COMM, SIZE, IERROR
```

IN comm – коммуникатор;
OUT size – число процессов в области связи коммуникатора comm.

Функция возвращает количество процессов в области связи коммуникатора comm.

До создания явным образом групп и связанных с ними коммуникаторов (глава 6) единственно возможными значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI. Подпрограмма является локальной.

Функция определения номера процесса MPI_Comm_rank

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

```
FORTRAN:
      MPI_COMM_RANK(COMM, RANK, IERROR)
      INTEGER COMM, RANK, IERROR
```

IN comm – коммуникатор;
OUT rank – номер процесса, вызвавшего функцию.

Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции). Подпрограмма является локальной.

В минимальный набор следует включить также две функции передачи и приема сообщений.

Функция передачи сообщения MPI_Send

```
C:
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
FORTRAN:
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
         IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

IN buf – адрес начала расположения пересылаемых данных;
IN count – число пересылаемых элементов;
IN datatype – тип посылаемых элементов;
IN dest – номер процесса-получателя в группе, связанной с коммуникатором comm;
IN tag – идентификатор сообщения (аналог типа сообщения функций nread и nwrite PSE nCUBE2);
IN comm – коммуникатор области связи.

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf – это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Функция приема сообщения MPI_Recv

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

```
FORTRAN:
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
            STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

OUT buf – адрес начала расположения принимаемого сообщения;
IN count – максимальное число принимаемых элементов;
IN datatype – тип элементов принимаемого сообщения;
IN source – номер процесса-отправителя;
IN tag – идентификатор сообщения;
IN comm – коммуникатор области связи;
OUT status – атрибуты принятого сообщения.

Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

Более детально об операциях обмена сообщениями мы поговорим в следующей главе, а в заключение этой главы рассмотрим функцию, которая не входит в очерченный нами минимум, но которая важна для разработки эффективных программ. Речь идет о функции отсчета времени – таймере. С одной стороны, такие функции имеются в составе всех операционных систем, но, с другой стороны, существует полнейший произвол в их реализации. Опыт работы с различными операционными системами показывает, что при переносе приложений с одной платформы на другую первое (а иногда и единственное), что приходится переделывать – это обращения к функциям учета времени. Поэтому разработчики MPI, добиваясь полной независимости приложений от операционной среды, определили и свои функции отсчета времени.

Функция отсчета времени (таймер) MPI_Wtime

```
C:
      double MPI_Wtime(void)
FORTRAN:
      DOUBLE PRECISION MPI_WTIME()
```

Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета). Гарантируется, что эта

точка отсчета не будет изменена в течение жизни процесса. Для хронометража участка программы вызов функции делается в начале и конце участка и определяется разница между показаниями таймера.

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
   … хронометрируемый участок …
   endtime   = MPI_Wtime();
   printf("Выполнение заняло %f секунд\n",endtime-starttime);
}
```

Функция MPI_Wtick, имеющая точно такой же синтаксис, возвращает разрешение таймера (минимальное значение кванта времени).

Глава 8.
КОММУНИКАЦИОННЫЕ ОПЕРАЦИИ ТИПА ТОЧКА-ТОЧКА

8.1. Обзор коммуникационных операций типа точка-точка

К операциям этого типа относятся две представленные в предыдущем разделе коммуникационные процедуры. В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов. Описанные в предыдущем разделе функции реализуют *стандартный режим с блокировкой*.

Блокирующие функции подразумевают выход из них только после полного окончания операции, т.е. вызывающий процесс блокируется, пока операция не будет завершена. Для функции посылки сообщения это означает, что все пересылаемые данные помещены в буфер (для разных реализаций MPI это может быть либо какой-то промежуточный системный буфер, либо непосредственно буфер получателя). Для функции приема сообщения блокируется выполнение других операций, пока все

данные из буфера не будут помещены в адресное пространство принимающего процесса.

Неблокирующие функции подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Как для блокирующих, так и неблокирующих операций MPI поддерживает четыре режима выполнения. Эти режимы касаются только функций передачи данных, поэтому для блокирующих и неблокирующих операций имеется по четыре функции посылки сообщения. В таблице 8.1 перечислены имена базовых коммуникационных функций типа точка-точка, имеющихся в библиотеке MPI.

Таблица 8.1. Список коммуникационных функций типа точка-точка.

Режимы выполнения	**С блокировкой**	**Без блокировки**
Стандартная посылка	MPI_Send	MPI_**I**send
Синхронная посылка	MPI_**S**send	MPI_**Is**send
Буферизованная посылка	MPI_**B**send	MPI_**Ib**send
Согласованная посылка	MPI_**R**send	MPI_**Ir**send
Прием информации	MPI_Recv	MPI_**I**recv

Из таблицы хорошо виден принцип формирования имен функций. К именам базовых функций Send/Recv добавляются различные префиксы.

Префикс S (synchronous) – означает синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. Функция нелокальная.

Префикс B (buffered) – означает буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.

Префикс R (ready) – согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция нелокальная.

Префикс I (immediate) – относится к неблокирующим операциям.

Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом. Функции передачи, находящиеся в одном столбце, имеют совершенно одинаковый синтаксис и отличаются только внутренней реализацией. Поэтому в дальнейшем будем рассматривать только стандартный режим, который в обязательном порядке поддерживают все реализации MPI.

8.2. Блокирующие коммуникационные операции

Синтаксис базовых коммуникационных функций MPI_Send и MPI_Recv был приведен в главе 7, поэтому здесь мы рассмотрим только семантику этих операций.

В стандартном режиме выполнение операции обмена включает три этапа.

1. Передающая сторона формирует пакет сообщения, в который помимо передаваемой информации упаковываются адрес отправителя (source), адрес получателя (dest), идентификатор сообщения (tag) и коммуникатор (comm). Этот пакет передается отправителем в системный буфер, и на этом функция посылки сообщения заканчивается.
2. Сообщение системными средствами передается адресату.
3. Принимающий процессор извлекает сообщение из системного буфера, когда у него появится потребность в этих данных. Содержательная часть сообщения помещается в адресное пространство принимающего процесса (параметр buf), а служебная – в параметр status.

Поскольку операция выполняется в асинхронном режиме, адресная часть принятого сообщения состоит из трех полей:

- коммуникатора (comm), поскольку каждый процесс может одновременно входить в несколько областей связи;
- номера отправителя в этой области связи (source);
- идентификатора сообщения (tag), который используется для взаимной привязки конкретной пары операций посылки и приема сообщений.

Параметр count (количество принимаемых элементов сообщения) в процедуре приема сообщения должен быть не меньше, чем длина принимаемого сообщения. При этом реально будет приниматься столько элементов, сколько находится в буфере. Такая реализация операции чтения связана с тем, что MPI допускает использование расширенных запросов:

- для идентификаторов сообщений (MPI_ANY_TAG – читать сообщение с любым идентификатором);
- для адресов отправителя (MPI_ANY_SOURCE – читать сообщение от любого отправителя).

Не допускается расширенных запросов для коммуникаторов. Расширенные запросы возможны только в операциях чтения. Интересно отметить, что таким же образом организованы операции обмена в PSE nCUBE2 [13]. В этом отражается фундаментальное свойство механизма передачи сообщений: асимметрия операций передачи и приема сообщений, связанная с тем, что инициатива в организации обмена принадлежит передающей стороне.

Таким образом, после чтения сообщения некоторые параметры могут оказаться неизвестными, а именно: число считанных элементов, идентификатор сообщения и адрес отправителя. Эту информацию можно получить с помощью параметра status. Переменные status должны быть явно объявлены в MPI-программе. В языке C status – это структура типа MPI_Status с тремя полями MPI_SOURCE, MPI_TAG, MPI_ERROR. В языке FORTRAN status – массив типа INTEGER размера

MPI_STATUS_SIZE. Константы MPI_SOURCE, MPI_TAG и MPI_ERROR определяют индексы элементов. Назначение полей переменной status представлено в таблице 8.2.

Таблица 8.2. Назначение полей переменной status.

Поля status	**C**	**FORTRAN**
Процесс-отправитель	status.MPI_SOURCE	status(MPI_SOURCE)
Идентификатор сообщения	status.MPI_TAG	status(MPI_TAG)
Код ошибки	status.MPI_ERROR	status(MPI_ERROR)

Как видно из таблицы 8.2, количество считанных элементов в переменную status не заносится.

Для определения числа фактически полученных элементов сообщения необходимо использовать специальную ***функцию MPI_Get_count.***

```
C:
int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype,
                   int *count)
FORTRAN:
MPI_GET_COUNT (STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT,
IERROR
```

IN status – атрибуты принятого сообщения;
IN datatype – тип элементов принятого сообщения;
OUT count – число полученных элементов.

Подпрограмма MPI_Get_count может быть вызвана либо после чтения сообщения (функциями MPI_Recv, MPI_Irecv), либо после опроса факта поступления сообщения (функциями MPI_Probe, MPI_Iprobe). Операция чтения безвозвратно уничтожает информацию в буфере приема. При этом попытка считать сообщение с параметром count меньше, чем число элементов в буфере, приводит к потере сообщения.

Определить параметры полученного сообщения без его чтения можно с помощью ***функции MPI_Probe.***

C:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status
               *status)
```

FORTRAN:

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR
```

IN source – номер процесса-отправителя;
IN tag – идентификатор сообщения;
IN comm – коммуникатор;
OUT status – атрибуты опрошенного сообщения.

Подпрограмма MPI_Probe выполняется с блокировкой, поэтому завершится она лишь тогда, когда сообщение с подходящим идентификатором и номером процесса-отправителя будет доступно для получения. Атрибуты этого сообщения возвращаются в переменной status. Следующий за MPI_Probe вызов MPI_Recv с теми же атрибутами сообщения (номером процесса-отправителя, идентификатором сообщения и коммуникатором) поместит в буфер приема именно то сообщение, наличие которого было опрошено подпрограммой MPI_Probe.

При использовании блокирующего режима передачи сообщений существует потенциальная опасность возникновения тупиковых ситуаций, в которых операции обмена данными блокируют друг друга. Приведем пример некорректной программы, которая будет зависать при любых условиях.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
   CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
   CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

В этом примере оба процесса (0-й и 1-й) входят в режим взаимного ожидания сообщения друг от друга. Такие тупиковые ситуации будут

возникать всегда при образовании циклических цепочек блокирующих операций чтения.

Приведем вариант правильной программы.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

Другие комбинации операций SEND/RECV могут работать или не работать в зависимости от реализации MPI (буферизованный обмен или нет).

В ситуациях, когда требуется выполнить ***взаимный обмен данными между процессами***, безопаснее использовать ***совмещенную операцию MPI_Sendrecv.***

C:

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
                 sendtype, int dest, int sendtag, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int source,
                 MPI_Datatype recvtag, MPI_Comm comm,
                 MPI_Status *status)
```

FORTRAN:

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST,
SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE,
RECVTAG, COMM, STATUS, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG,
RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR
```

IN	sendbuf	– адрес начала расположения посылаемого сообщения;
IN	sendcount	– число посылаемых элементов;
IN	sendtype	– тип посылаемых элементов;
IN	dest	– номер процесса-получателя;
IN	sendtag	– идентификатор посылаемого сообщения;
OUT	recvbuf	– адрес начала расположения принимаемого сообщения;
IN	recvcount	– максимальное число принимаемых элементов;
IN	recvtype	– тип элементов принимаемого сообщения;
IN	source	– номер процесса-отправителя;

IN recvtag – идентификатор принимаемого сообщения;
IN comm – коммуникатор области связи;
OUT status – атрибуты принятого сообщения.

Функция MPI_Sendrecv совмещает выполнение операций передачи и приема. Обе операции используют один и тот же коммуникатор, но идентификаторы сообщений могут различаться. Расположение в адресном пространстве процесса принимаемых и передаваемых данных не должно пересекаться. Пересылаемые данные могут быть различного типа и иметь разную длину.

В тех случаях, когда необходим ***обмен данными одного типа с замещением посылаемых данных на принимаемые***, удобнее пользоваться ***функцией MPI_Sendrecv_replace.***

С:

```
MPI_Sendrecv_replace(void* buf, intcount, MPI_Datatype datatype,
                     int dest, int sendtag, int source, int recvtag,
                     MPI_Comm comm, MPI_Status *status)
```

FORTRAN:

```
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST,
SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE,
RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

INOUT buf – адрес начала расположения посылаемого и принимаемого сообщения;
IN count – число передаваемых элементов;
IN datatype – тип передаваемых элементов;
IN dest – номер процесса-получателя;
IN sendtag – идентификатор посылаемого сообщения;
IN source – номер процесса-отправителя;
IN recvtag – идентификатор принимаемого сообщения;
IN comm – коммуникатор области связи;
OUT status – атрибуты принятого сообщения.

В данной операции посылаемые данные из массива buf замещаются принимаемыми данными.

В качестве адресатов source и dest в операциях пересылки данных можно использовать специальный адрес MPI_PROC_NULL.

Коммуникационные операции с таким адресом ничего не делают. Применение этого адреса бывает удобным вместо использования логических конструкций для анализа условий посылать/читать сообщение или нет. Этот прием будет использован нами далее в одном из примеров, а именно, в программе решения уравнения Лапласа методом Якоби.

8.3. Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверку завершения операции.

Неблокирующие операции используют специальный скрытый (opaque) объект "запрос обмена" (request) для связи между функциями обмена и функциями опроса их завершения. Для прикладных программ доступ к этому объекту возможен только через вызовы MPI-функций. Если операция обмена завершена, подпрограмма проверки снимает "запрос обмена", устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free.

Функция передачи сообщения без блокировки MPI_Isend

```
C:
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm, MPI_Request *request)
FORTRAN:
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
IERROR
```

IN buf – адрес начала расположения передаваемых данных;
IN count – число посылаемых элементов;

IN datatype – тип посылаемых элементов;
IN dest – номер процесса-получателя;
IN tag – идентификатор сообщения;
IN comm – коммуникатор;
OUT request – “запрос обмена”.

Возврат из подпрограммы происходит немедленно (immediate), без ожидания окончания передачи данных. Этим объясняется префикс I в именах функций. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен “запрос обмена”. Это можно сделать с помощью подпрограмм MPI_Wait или MPI_Test, передав им параметр request.

Функция приема сообщения без блокировки MPI_Irecv

C:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request *request)
```

FORTRAN:

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

OUT buf – адрес для принимаемых данных;
IN count – максимальное число принимаемых элементов;
IN datatype – тип элементов принимаемого сообщения;
IN source – номер процесса-отправителя;
IN tag – идентификатор сообщения;
IN comm – коммуникатор;
OUT request – “запрос обмена”.

Возврат из подпрограммы происходит немедленно, без ожидания окончания приема данных. Определить момент окончания приема можно с помощью подпрограмм MPI_Wait или MPI_Test с соответствующим параметром request.

Как и в блокирующих операциях часто возникает необходимость опроса параметров полученного сообщения без его фактического чтения. Это делается с помощью функции MPI_Iprobe.

Неблокирующая функция чтения параметров полученного сообщения MPI_Iprobe

C:
```
int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag
                    MPI_Status *status)
```
FORTRAN:
```
MPI_IPROBE (SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
IERROR
```

IN source – номер процесса-отправителя;
IN tag – идентификатор сообщения;
IN comm – коммуникатор;
OUT flag – признак завершенности операции;
OUT status – атрибуты опрошенного сообщения.

Если flag=true, то операция завершилась, и в переменной status находятся атрибуты этого сообщения.

Воспользоваться результатом неблокирующей коммуникационной операции или повторно использовать ее параметры можно только после ее полного завершения. Имеется два типа функций завершения неблокирующих операций (ожидание завершения и проверки завершения):

1. Операции семейства WAIT блокируют работу процесса до полного завершения операции.
2. Операции семейства TEST возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет. Они не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

Функция ожидания завершения неблокирующей операции MPI_Wait

C:
```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```
FORTRAN:
```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

INOUT request – “запрос обмена”;
OUT status – атрибуты сообщения.

Это нелокальная блокирующая операция. Возврат происходит после завершения операции, связанной с запросом request. В параметре status возвращается информация о законченной операции.

Функция проверки завершения неблокирующей операции MPI_Test

C:

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

FORTRAN:

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

INOUT request – “запрос обмена”;
OUT flag – признак завершенности проверяемой операции;
OUT status – атрибуты сообщения, если операция завершилась.

Это локальная неблокирующая операция. Если связанная с запросом request операция завершена, возвращается flag = true, а status содержит информацию о завершенной операции. Если проверяемая операция не завершена, возвращается flag = false, а значение status в этом случае не определено.

Рассмотрим пример использования неблокирующих операций и функции MPI_Wait.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
   **** Выполнение вычислений во время передачи сообщения ****
   CALL MPI_WAIT(request, status, ierr)
ELSE
   CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
   **** Выполнение вычислений во время приема сообщения ****
   CALL MPI_WAIT(request, status, ierr)
END IF
```

Функция снятия запроса без ожидания завершения неблокирующей операции MPI_Request_free

```
C:
int MPI_Request_free(MPI_Request *request)
FORTRAN:
MPI_REQUEST_FREE(REQUEST, IERROR)
INTEGER REQUEST, IERROR
```

INOUT request – “запрос обмена”.

Параметр request устанавливается в значение MPI_REQUEST_NULL. Связанная с этим запросом операция не прерывается, однако проверить ее завершение с помощью MPI_Wait или MPI_Test уже нельзя. Для прерывания коммуникационной операции следует использовать функцию MPI_Cancel(MPI_Request *request).

В MPI имеется набор подпрограмм для одновременной проверки на завершение нескольких операций. Без подробного обсуждения приведем их перечень (таблица 8.3).

Таблица 8.3. Функции коллективного завершения неблокирующих операций.

Выполняемая проверка	**Функции ожидания (блокирующие)**	**Функции проверки (неблокирующие)**
Завершились все операции	MPI_Waitall	MPI_Testall
Завершилась по крайней мере одна операция	MPI_Waitany	MPI_Testany
Завершилась одна из списка проверяемых	MPI_Waitsome	MPI_Testsome

Кроме того, MPI позволяет для неблокирующих операций формировать целые пакеты запросов на коммуникационные операции MPI_Send_init и MPI_Recv_init, которые запускаются функциями MPI_Start или MPI_Startall. Проверка на завершение выполнения производится обычными средствами с помощью функций семейства WAIT и TEST.

Глава 9.
КОЛЛЕКТИВНЫЕ ОПЕРАЦИИ

9.1. Обзор коллективных операций

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

- Синхронизацию всех процессов с помощью барьеров (MPI_Barrier).
- Коллективные коммуникационные операции, в число которых входят:
    - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);

- сборка (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
- сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
- разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
- совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами, в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).

- Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
  - с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
  - с рассылкой результата всем процессам (MPI_Allreduce);
  - совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
  - префиксная редукция (MPI_Scan).

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;

- "векторный" вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

Отличительные особенности коллективных операций:

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Примечание: В данной главе часто будут использоваться понятия *буфер обмена*, *буфер передачи*, *буфер приема*. Не следует понимать эти понятия в буквальном смысле как некую специальную область памяти, куда помещаются данные перед вызовом коммуникационной функции. На самом деле, это, как правило, используемые в программе обычные массивы, которые непосредственно могут участвовать в коммуникационных операциях. В вызовах подпрограмм передается адрес начала непрерывной области памяти, которая будет участвовать в операции обмена.

Изучение коллективных операций начнем с рассмотрения двух функций, стоящих особняком: MPI_Barrier и MPI_Bcast.

Функция синхронизации процессов MPI_Barrier блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами одновременно (все процессы “преодолевают барьер” одновременно).

```
C:
        int MPI_Barrier(MPI_Comm comm)
FORTRAN:
        MPI_BARRIER(COMM, IERROR)
        INTEGER COMM, IERROR
IN      comm        – коммуникатор.
```

Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Неявную синхронизацию процессов выполняет любая коллективная функция.

Широковещательная рассылка данных выполняется с помощью ***функции MPI_Bcast***. Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.

```
C:
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
                    int root, MPI_Comm comm)
FORTRAN:
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

```
INOUT     buffer      – адрес начала расположения в памяти
                        рассылаемых данных;
IN        count       – число посылаемых элементов;
IN        datatype    – тип посылаемых элементов;
IN        root        – номер процесса-отправителя;
IN        comm        – коммуникатор.
```

После завершения подпрограммы каждый процесс в области связи коммуникатора comm, включая и самого отправителя, получит копию сообщения от процесса-отправителя root. На рис. 9.1 представлена графическая интерпретация операции Bcast.

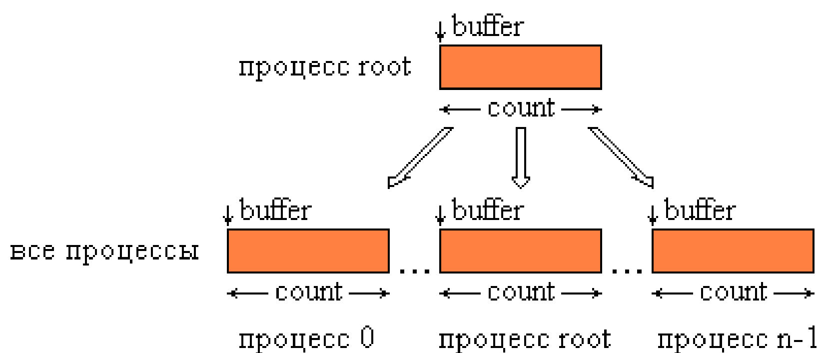

Рис 9.1. Графическая интерпретация операции Bcast.

Пример использования функции MPI_Bcast

```
IF ( MYID .EQ. 0 ) THEN
PRINT *, 'ВВЕДИТЕ ПАРАМЕТР N : '
READ *, N
END IF
CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, IERR)
```

9.2. Функции сбора блоков данных от всех процессов группы

Семейство функций сбора блоков данных от всех процессов группы состоит из четырех подпрограмм: MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv. Каждая из указанных подпрограмм расширяет функциональные возможности предыдущих.

Функция MPI_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

C:
```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
                  sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, int root, MPI_Comm comm)
```
FORTRAN:
```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR
```

IN	sendbuf	– адрес начала размещения посылаемых данных;
IN	sendcount	– число посылаемых элементов;
IN	sendtype	– тип посылаемых элементов;
OUT	recvbuf	– адрес начала буфера приема (используется только в процессе-получателе root);
IN	recvcount	– число элементов, получаемых от каждого процесса (используется только в процессе-получателе root);
IN	recvtype	– тип получаемых элементов;
IN	root	– номер процесса-получателя;
IN	comm	– коммуникатор.

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число sendcount должно равняться числу recvcount. То есть, recvcount в вызове из процесса root – это число собираемых от каждого процесса элементов, а не общее количество собранных элементов. Графическая интерпретация операции Gather представлена на Рис. 9.2.

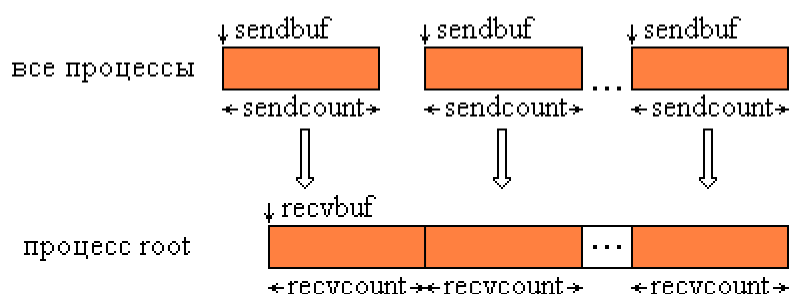

Рис. 9.2. Графическая интерпретация операции Gather.

Пример программы с использованием функции MPI_Gather

```
MPI_Comm comm;
int array[100];
int root, *rbuf;
… …
```

```
MPI_Comm_size(comm, &gsize);
rbuf = (int *) malloc( gsize * 100 * sizeof(int));
MPI_Gather(array, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.

C:

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
                  sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm)
```

FORTRAN:

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
              RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
COMM, IERROR
```

IN sendbuf – адрес начала буфера посылки;
IN sendcount – число посылаемых элементов;
IN sendtype – тип посылаемых элементов;
OUT recvbuf – адрес начала буфера приема;
IN recvcount – число элементов, получаемых от каждого процесса;
IN recvtype – тип получаемых элементов;
IN comm – коммуникатор.

Графическая интерпретация операции Allgather представлена на рис. 9.3. На этой схеме ось Y образуют процессы группы, а ось X – блоки данных.

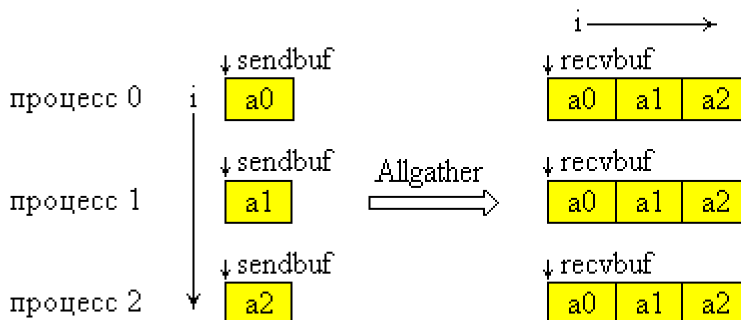

Рис 9.3. Графическая интерпретация операции Allgather.

Функция MPI_Gatherv позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов,

принимаемых от каждого процесса, задается индивидуально с помощью массива recvcounts. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений displs.

C:

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype
                    sendtype, void* rbuf, int *recvcounts, int *displs,
                    MPI_Datatype recvtype, int root, MPI_Comm
                    comm)
```

FORTRAN:

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RBUF,
RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
RECVTYPE, ROOT, COMM, IERROR
```

IN sendbuf – адрес начала буфера передачи;
IN sendcount – число посылаемых элементов;
IN sendtype – тип посылаемых элементов;
OUT rbuf – адрес начала буфера приема;
IN recvcounts – целочисленный массив (размер равен числу процессов в группе), i-й элемент массива определяет число элементов, которое должно быть получено от процесса i;
IN displs – целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение i-го блока данных относительно начала rbuf;
IN recvtype – тип получаемых элементов;
IN root – номер процесса-получателя;
IN comm – коммуникатор.

Сообщения помещаются в буфер приема процесса root в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом i, размещаются в адресном пространстве процесса root, начиная с адреса rbuf + displs[i]. Графическая интерпретация операции Gatherv представлена на рис. 9.4.

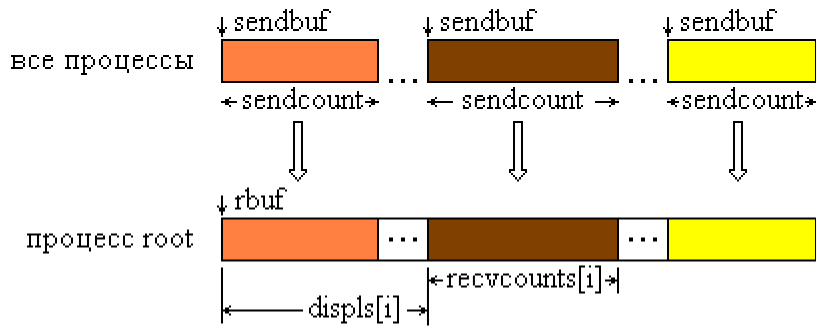

Рис. 9.4. Графическая интерпретация операции Gatherv.

Функция MPI_Allgatherv является аналогом функции MPI_Gatherv, но сборка выполняется всеми процессами группы. Поэтому в списке параметров отсутствует параметр root.

```
C:
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype
                   sendtype, void* rbuf, int *recvcounts,
                   int *displs, MPI_Datatype recvtype,
                   MPI_Comm comm)
FORTRAN:
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RBUF,
RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
RECVTYPE, COMM, IERROR
```

IN sendbuf – адрес начала буфера передачи;
IN sendcount – число посылаемых элементов;
IN sendtype – тип посылаемых элементов;
OUT rbuf – адрес начала буфера приема;
IN recvcounts – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса;
IN displs – целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение относительно начала rbuf i-го блока данных;
IN recvtype – тип получаемых элементов;
IN comm – коммуникатор.

9.3. Функции распределения блоков данных по всем процессам группы

Семейство функций распределения блоков данных по всем процессам группы состоит из двух подпрограмм: MPI_Scatter и MPI_Scatterv.

Функция MPI_Scatter разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе). Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, несущественны.

```
C:
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
                 sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm
                 comm)
FORTRAN:
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR
```

IN sendbuf – адрес начала размещения блоков распределяемых данных (используется только в процессе-отправителе root);
IN sendcount – число элементов, посылаемых каждому процессу;
IN sendtype – тип посылаемых элементов;
OUT recvbuf – адрес начала буфера приема;
IN recvcount – число получаемых элементов;
IN recvtype – тип получаемых элементов;
IN root – номер процесса-отправителя;
IN comm – коммуникатор.

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых элементов, а число посылаемых элементов sendcount должно

равняться числу принимаемых recvcount. Следует обратить внимание, что значение sendcount в вызове из процесса root – это число посылаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather. На рис. 9.5 представлена графическая интерпретация операции Scatter.

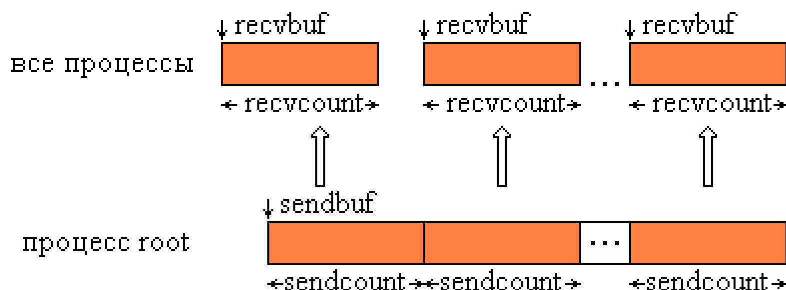

Рис. 9.5. Графическая интерпретация операции Scatter.

Пример использования функции MPI_Scatter

```
MPI_Comm comm;
int    rbuf[100], gsize;
int    root, *array;
… …
MPI_Comm_size(comm, &gsize);
array = (int *) malloc( gsize * 100 * sizeof(int));
MPI_Scatter(array, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Функция MPI_Scatterv является векторным вариантом функции MPI_Scatter, позволяющим посылать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого i-му процессу, задается в массиве смещений displs, а число посылаемых элементов – в массиве sendcounts. Эта функция является обратной по отношению к функции MPI_Gatherv.

```
C:
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                MPI_Datatype sendtype, void* recvbuf, int recvcount,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
FORTRAN:
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,
RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR
```

IN sendbuf – адрес начала буфера посылки (используется только в процессе-отправителе root);
IN sendcounts – целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу;
IN displs – целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение относительно начала sendbuf для данных, посылаемых процессу i;
IN sendtype – тип посылаемых элементов;
OUT recvbuf – адрес начала буфера приема;
IN recvcount – число получаемых элементов;
IN recvtype – тип получаемых элементов;
IN root – номер процесса-отправителя;
IN comm – коммуникатор.

На рис. 9.6 представлена графическая интерпретация операции Scatterv.

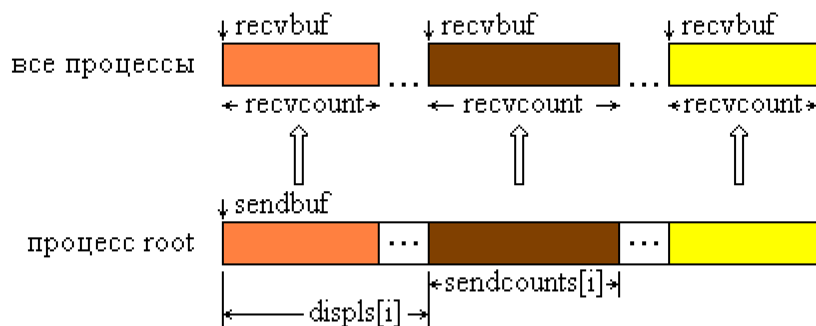

Рис. 9.6. Графическая интерпретация операции Scatterv.

9.4. Совмещенные коллективные операции

Функция MPI_Alltoall совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j-ый блок своего буфера sendbuf процессу j, который помещает его в i-ый блок своего буфера recvbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.

C:

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype
                 sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, MPI_Comm comm)
```

FORTRAN:
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

IN sendbuf – адрес начала буфера посылки;
IN sendcount – число посылаемых элементов;
IN sendtype – тип посылаемых элементов;
OUT recvbuf – адрес начала буфера приема;
IN recvcount – число элементов, получаемых от каждого процесса;
IN recvtype – тип получаемых элементов;
IN comm – коммуникатор.

Графическая интерпретация операции Alltoall представлена на рис. 9.7.

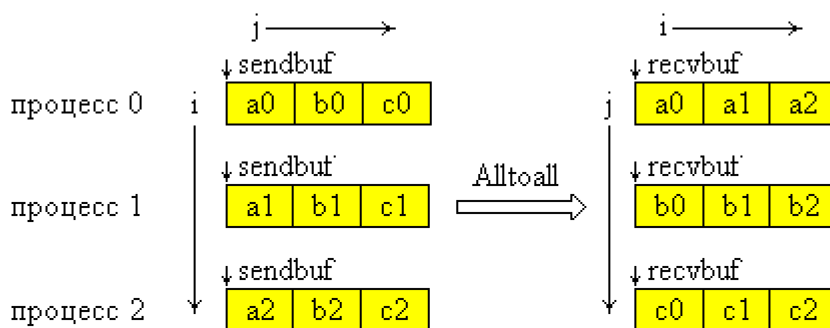

Рис. 9.7. Графическая интерпретация операции Alltoall.

Функция MPI_Alltoallv реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

9.5. Глобальные вычислительные операции над распределенными данными

В параллельном программировании математические операции над блоками данных, распределенных по процессорам, называют *глобальными операциями редукции*. В общем случае *операцией редукции* называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то в этом случае говорят о

глобальной (параллельной) редукции. Например, пусть в адресном пространстве всех процессов некоторой группы процессов имеются копии переменной var (необязательно имеющие одно и то же значение), тогда применение к ней операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной. Использование операций редукции является одним из основных средств организации распределенных вычислений.

В MPI глобальные операции редукции представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
- с сохранением результата в адресном пространстве всех процессов (MPI_Allreduce);
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. i-я компонента этого вектора является результатом редукции первых i компонент распределенного вектора (MPI_Scan);
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter).

Функция MPI_Reduce выполняется следующим образом. Операция глобальной редукции, указанная параметром op, выполняется над первыми элементами входного буфера, и результат посылается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т.д.

```
C:
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
FORTRAN:
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

IN sendbuf – адрес начала входного буфера;
OUT recvbuf – адрес начала буфера результатов (используется только в процессе-получателе root);
IN count – число элементов во входном буфере;
IN datatype – тип элементов во входном буфере;
IN op – операция, по которой выполняется редукция;
IN root – номер процесса-получателя результата операции;
IN comm – коммуникатор.

На рис. 9.8 представлена графическая интерпретация операции Reduce. На данной схеме операция '+' означает любую допустимую операцию редукции.

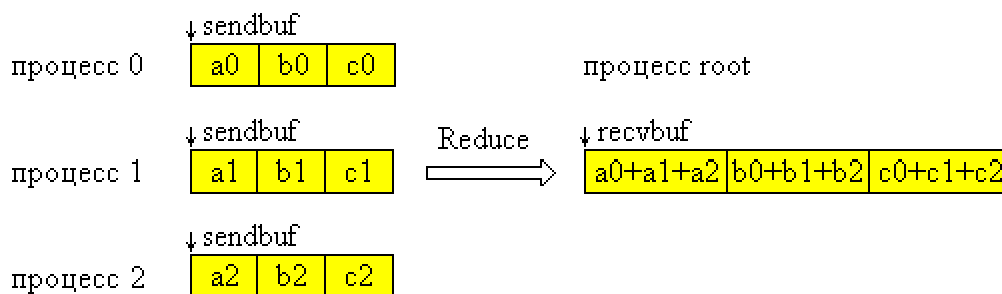

Рис. 9.8. Графическая интерпретация операции Reduce.

В качестве операции op можно использовать либо одну из предопределенных операций, либо операцию, сконструированную пользователем. Все предопределенные операции являются ассоциативными и коммутативными. Сконструированная пользователем операция, по крайней мере, должна быть ассоциативной. Порядок редукции определяется номерами процессов в группе. Тип datatype элементов должен быть совместим с операцией op. В таблице 9.1 представлен перечень предопределенных операций, которые могут быть использованы в функциях редукции MPI.

Таблица 9.1. Предопределенные операции в функциях редукции MPI.

Название	Операция	Разрешенные типы
MPI_MAX MPI_MIN	Максимум Минимум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сумма Произведение	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логическое AND Логическое OR Логическое исключающее OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Поразрядное AND Поразрядное OR Поразрядное исключающее OR	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальное значение и его индекс Минимальное значение и его индекс	Специальные типы для этих функций

В таблице используются следующие обозначения:

C integer: MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG
FORTRAN integer: MPI_INTEGER
Floating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical: MPI_LOGICAL
Complex: MPI_COMPLEX
Byte: MPI_BYTE

Операции MAXLOC и MINLOC выполняются над специальными парными типами, каждый элемент которых хранит две величины: значения, по которым ищется максимум или минимум, и индекс элемента. В MPI имеется 9 таких предопределенных типов.

C:

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	int and int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

FORTRAN:

MPI_2REAL	REAL and REAL
MPI_2DOUBLE_PRECISION	DOUBLE PRECISION and DOUBLE PRECISION
MPI_2INTEGER	INTEGER and INTEGER

Функция MPI_Allreduce сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса root. В остальном, набор параметров такой же, как и в предыдущей функции.

```
C:
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
FORTRAN:
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
                  COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

IN sendbuf – адрес начала входного буфера;
OUT recvbuf – адрес начала буфера приема;
IN count – число элементов во входном буфере;
IN datatype – тип элементов во входном буфере;
IN op – операция, по которой выполняется редукция;
IN comm – коммуникатор.

На рис. 9.9 представлена графическая интерпретация операции Allreduce.

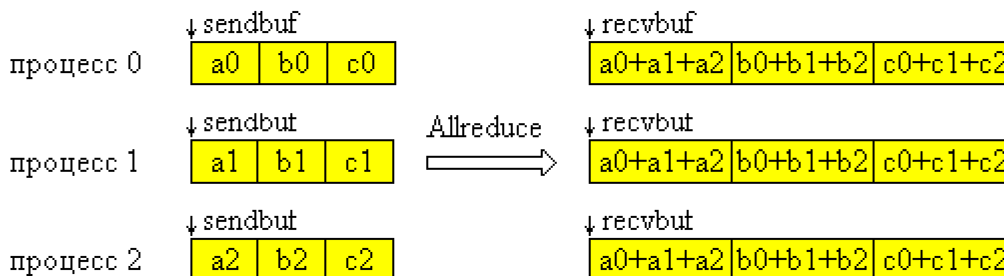

Рис. 9.9. Графическая интерпретация операции Allreduce.

Функция MPI_Reduce_scatter совмещает в себе операции редукции и распределения результата по процессам.

```
C:
MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
               MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
FORTRAN:
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS,
                              DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

IN sendbuf – адрес начала входного буфера;
OUT recvbuf – адрес начала буфера приема;

IN revcount – массив, в котором задаются размеры блоков, посылаемых процессам;
IN datatype – тип элементов во входном буфере;
IN op – операция, по которой выполняется редукция;
IN comm – коммуникатор.

Функция MPI_Reduce_scatter отличается от MPI_Allreduce тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе, i-ая часть посылается i-ому процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом. На рис. 9.10 представлена графическая интерпретация операции Reduce_scatter.

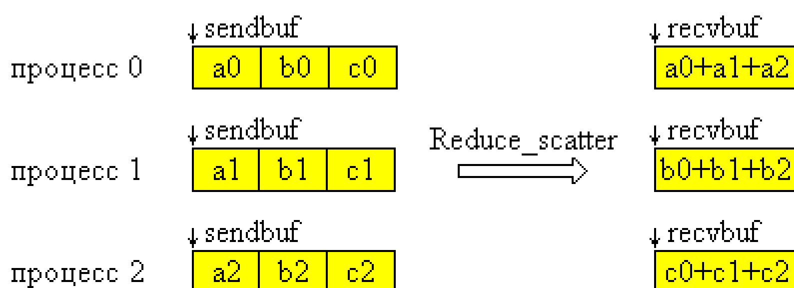

Рис. 9.10. Графическая интерпретация операции Reduce_scatter.

Функция MPI_Scan выполняет префиксную редукцию. Параметры такие же, как в MPI_Allreduce, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0, … , i включительно.

```
C:
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
FORTRAN:
MPI_MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
             COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

IN sendbuf – адрес начала входного буфера;
OUT recvbuf – адрес начала буфера приема;
IN count – число элементов во входном буфере;
IN datatype – тип элементов во входном буфере;
IN op – операция, по которой выполняется редукция;

IN comm – коммуникатор.

На рис. 9.11 представлена графическая интерпретация операции Scan.

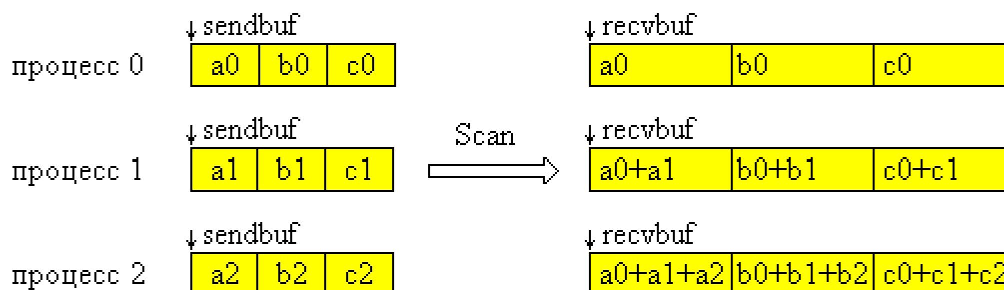

Рис. 9.11. Графическая интерпретация операции Scan.

Глава 10.

ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ И ПЕРЕДАЧА УПАКОВАННЫХ ДАННЫХ

Рассмотренные ранее коммуникационные операции позволяют посылать или получать последовательность элементов одного типа, занимающих смежные области памяти. При разработке параллельных программ иногда возникает потребность передавать данные разных типов (например, структуры) или данные, расположенные в несмежных областях памяти (части массивов, не образующих непрерывную последовательность элементов). MPI предоставляет два механизма эффективной пересылки данных в упомянутых выше случаях:

- путем создания производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI;
- пересылку упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

В большинстве случаев оба эти механизма позволяют добиться желаемого результата, но в конкретных случаях более эффективным может оказаться либо один, либо другой подход.

10.1. Производные типы данных

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме коммуникационных. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (opaque) объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как *отображение (карта) типа*:

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$$

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию. Отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена. Этот буфер будет содержать n элементов, а i-й элемент будет иметь адрес buf+desp и иметь базовый тип type. Стандартные типы MPI имеют предопределенные отображения типов. Например, MPI_INT имеет отображение {(int,0)}.

Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов
  MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.
- Новый производный тип регистрируется вызовом функции MPI_Type_commit. Только после регистрации новый производный

тип можно использовать в коммуникационных подпрограммах и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.

- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

- Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как
    адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой MPI_Type_extent).
- Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой MPI_Type_size).

Для простых типов протяженность и размер совпадают.

Функция MPI_Type_extent определяет протяженность элемента некоторого типа.

```
C:
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
FORTRAN:
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
```

IN datatype – тип данных;
OUT extent – протяженность элемента заданного типа.

Функция MPI_Type_size определяет “чистый” размер элемента некоторого типа (за вычетом пустых промежутков).

```
C:
int MPI_Type_size(MPI_Datatype datatype, int *size)
FORTRAN:
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
```

INTEGER DATATYPE, SIZE, IERROR

IN datatype – тип данных;
OUT size – размер элемента заданного типа.

Как отмечалось выше, для создания производных типов в MPI имеется набор специальных функций-конструкторов. Рассмотрим их в порядке увеличения сложности.

Самый простой ***конструктор типа MPI_Type_contiguous*** создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти.

C:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

FORTRAN:

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

IN count – число элементов базового типа;
IN oldtype – базовый тип данных;
OUT newtype – новый производный тип данных.

Графическая интерпретация работы конструктора MPI_Type_contiguous приведена на рис. 10.1.

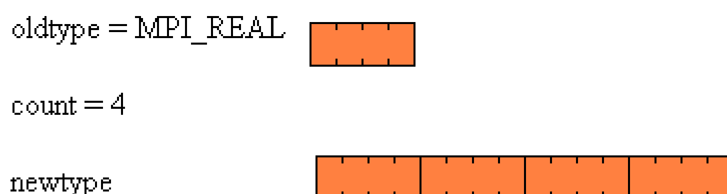

Рис. 10.1. Графическая интерпретация работы конструктора MPI_Type_contiguous.

Конструктор типа MPI_Type_vector создает тип, элемент которого представляет собой несколько равноудаленных друг от друга блоков из одинакового числа смежных элементов базового типа.

C:

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

FORTRAN:

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
NEWTYPE, IERROR)
```

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

IN count – число блоков;
IN blocklength – число элементов базового типа в каждом блоке;
IN stride – шаг между началами соседних блоков, измеренный числом элементов базового типа;
IN oldtype – базовый тип данных;
OUT newtype – новый производный тип данных.

Функция создает тип newtype, элемент которого состоит из count блоков, каждый из которых содержит одинаковое число blocklength элементов типа oldtype. Шаг stride между началом блока и началом следующего блока всюду одинаков и кратен протяженности представления базового типа. Графическая интерпретация работы конструктора MPI_Type_vector приведена на рис. 10.2.

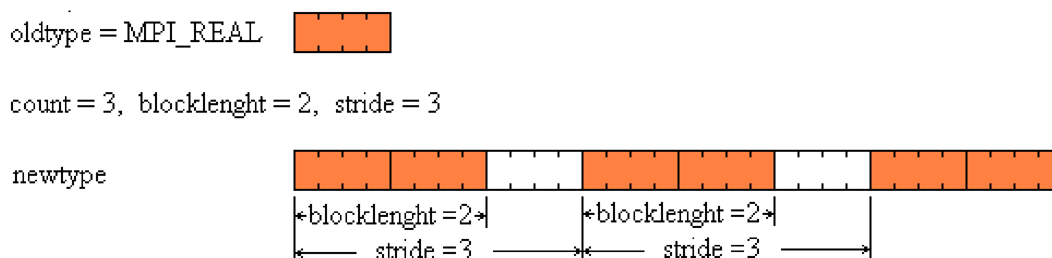

Рис. 10.2. Графическая интерпретация работы конструктора MPI_Type_vector.

Конструктор типа MPI_Type_hvector расширяет возможности конструктора MPI_Type_vector, позволяя задавать произвольный шаг между началами блоков в байтах.

C:

```
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
                     MPI_Datatype oldtype, MPI_Datatype
                     *newtype)
```

FORTRAN:

```
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE,
OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR
```

IN count – число блоков;
IN blocklength – число элементов базового типа в каждом блоке;
IN stride – шаг между началами соседних блоков в байтах;
IN oldtype – базовый тип данных;

OUT newtype – новый производный тип данных.

Графическая интерпретация работы конструктора MPI_Type_hvector приведена на рис. 10.3.

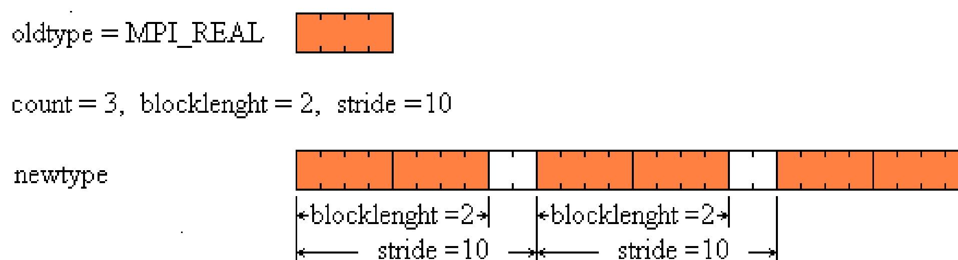

Рис. 10.3. Графическая интерпретация работы конструктора MPI_Type_hvector.

Конструктор типа MPI_Type_indexed является более универсальным конструктором по сравнению с MPI_Type_vector, так как элементы создаваемого типа состоят из произвольных по длине блоков с произвольным смещением блоков от начала размещения элемента. Смещения измеряются в элементах старого типа.

C:

```
int MPI_Type_indexed(int count, int *array_of_blocklengths,
                     int *array_of_displacements, MPI_Datatype
                     oldtype, MPI_Datatype *newtype)
```

FORTRAN:

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
```

IN count – число блоков;
IN array_of_blocklengths – массив, содержащий число элементов базового типа в каждом блоке;
IN array_of_displacements – массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа;
IN oldtype – базовый тип данных;
OUT newtype – новый производный тип данных.

Эта функция создает тип newtype, каждый элемент которого состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов базового типа и смещен от начала размещения элемента нового типа на array_of_displacements[i] элементов базового типа.

Графическая интерпретация работы конструктора MPI_Type_indexed приведена на рис. 10.4.

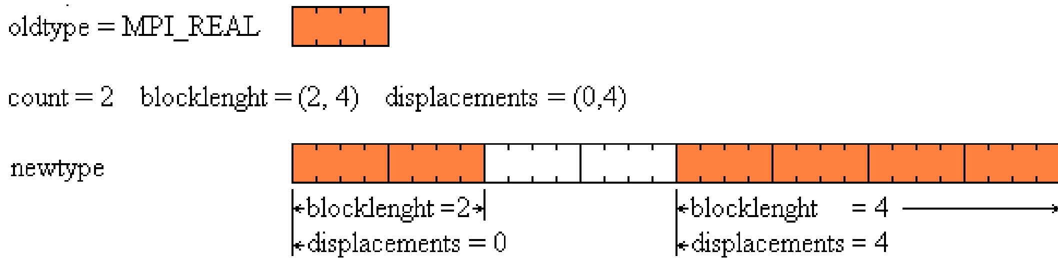

Рис. 10.4. Графическая интерпретация работы конструктора MPI_Type_indexed.

Конструктор типа MPI_Type_hindexed идентичен конструктору MPI_Type_indexed за исключением того, что смещения измеряются в байтах.

C:

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
                      MPI_Aint *array_of_displacements,
                      MPI_Datatype oldtype, MPI_Datatype *newtype)
```

FORTRAN:

```
MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
```

IN	count	– число блоков;
IN	array_of_blocklengths	– массив, содержащий число элементов базового типа в каждом блоке;
IN	array_of_displacements	– массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются в байтах;
IN	oldtype	– базовый тип данных;
OUT	newtype	– новый производный тип данных.

Элемент нового типа состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов старого типа и смещен от начала размещения элемента нового типа на array_of_displacements[i] байт. Графическая интерпретация работы конструктора MPI_Type_hindexed приведена на рис. 10.5.

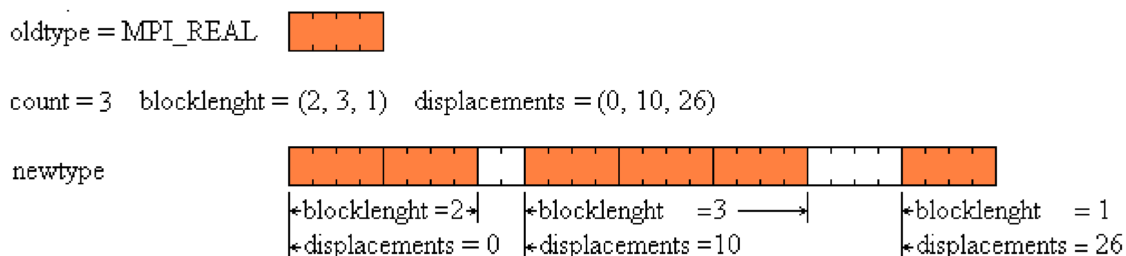

Рис. 10.5. Графическая интерпретация работы конструктора MPI_Type_hindexed.

Конструктор типа MPI_Type_struct – самый универсальный из всех конструкторов типа. Создаваемый им тип является структурой, состоящей из произвольного числа блоков, каждый из которых может содержать произвольное число элементов одного из базовых типов и может быть смещен на произвольное число байтов от начала размещения структуры.

C:

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
                    MPI_Aint *array_of_displacements, MPI_Datatype
                    *array_of_types, MPI_Datatype *newtype)
```

FORTRAN:

```
MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE,
IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*),
NEWTYPE, IERROR
```

IN	count	– число блоков;
IN	array_of_blocklength	– массив, содержащий число элементов одного из базовых типов в каждом блоке;
IN	array_of_displacements	– массив смещений каждого блока от начала размещения структуры, смещения измеряются в байтах;
IN	array_of_types	– массив, содержащий тип элементов в каждом блоке;
OUT	newtype	– новый производный тип данных.

Функция создает тип newtype, элемент которого состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов типа

array_of_types[i]. Смещение i-ого блока от начала размещения элемента нового типа измеряется в байтах и задается в array_of_displacements[i]. Графическая интерпретация работы конструктора MPI_Type_struct приведена на рис. 10.6.

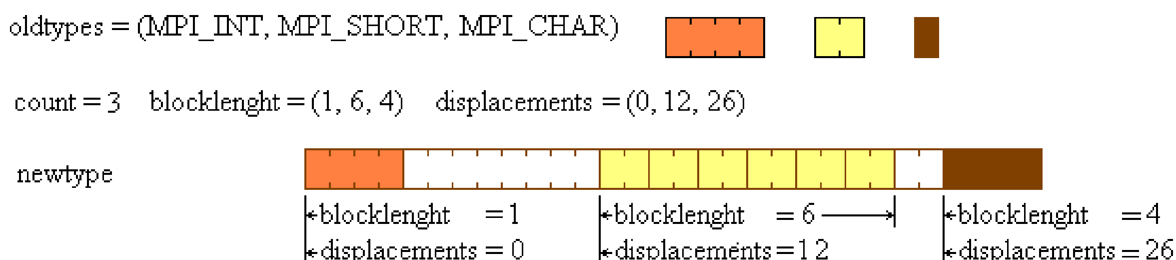

Рис. 10.6. Графическая интерпретация работы конструктора MPI_Type_struct.

Функция MPI_Type_commit регистрирует созданный производный тип. Только после регистрации новый тип может использоваться в коммуникационных операциях.

C:

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

FORTRAN:

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR
```

INOUT datatype – новый производный тип данных.

Функция MPI_Type_free уничтожает описатель производного типа.

C:

```
int MPI_Type_free(MPI_Datatype *datatype)
```

FORTRAN:

```
MPI_TYPE_FREE(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR
```

INOUT datatype – уничтожаемый производный тип.

Функция MPI_Type_free устанавливает описатель типа в состояние MPI_DATATYPE_NULL. Это не повлияет на выполняющиеся в данный момент коммуникационные операции с этим типом данных и на производные типы, которые ранее были определены через уничтоженный тип.

Для определения длины сообщения используются две функции: MPI_Get_count и MPI_Get_elements. Для сообщений из простых типов они возвращают одинаковое число. Подпрограмма MPI_Get_count возвращает число элементов типа datatype, указанного в операции получения. Если получено не целое число элементов, то она возвратит константу MPI_UNDEFINED (функция MPI_Get_count рассматривалась в главе 8, посвященной коммуникационным операциям типа точка-точка).

Функция MPI_Get_elements возвращает число элементов простых типов, содержащихся в сообщении.

```
C:
 int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype,
                          int *count)
FORTRAN:
MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT,
IERROR
```

IN status – статус сообщения;
IN datatype – тип элементов сообщения;
OUT count – число элементов простых типов, содержащихся в сообщении.

10.2. Передача упакованных данных

Функция MPI_Pack упаковывает элементы предопределенного или производного типа MPI, помещая их побайтное представление в выходной буфер.

```
C:
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void
                *outbuf, int outsize, int *position, MPI_Comm comm)
FORTRAN:
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
POSITION, COMM, IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM,
IERROR
```

INOUT inbuf – адрес начала области памяти с элементами, которые требуется упаковать;
IN incount – число упаковываемых элементов;

IN datatype – тип упаковываемых элементов;
OUT outbuf – адрес начала выходного буфера для упакованных данных;
IN outsize – размер выходного буфера в байтах;
INOUT position – текущая позиция в выходном буфере в байтах;
IN comm – коммуникатор.

Функция MPI_Pack упаковывает incount элементов типа datatype из области памяти с начальным адресом inbuf. Результат упаковки помещается в выходной буфер с начальным адресом outbuf и размером outsize байт. Параметр position указывает текущую позицию в байтах, начиная с которой будут размещаться упакованные данные. На выходе из подпрограммы значение position увеличивается на число упакованных байт, указывая на первый свободный байт. Параметр comm при последующей посылке упакованного сообщения будет использован как коммуникатор.

Функция MPI_Unpack извлекает заданное число элементов некоторого типа из побайтного представления элементов во входном массиве.

C:

```
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

FORTRAN:

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
DATATYPE, COMM, IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM,
IERROR
```

IN inbuf – адрес начала входного буфера с упакованными данными;
IN insize – размер входного буфера в байтах;
INOUT position – текущая позиция во входном буфере в байтах;
OUT outbuf – адрес начала области памяти для размещения распакованных элементов;
IN outcount – число извлекаемых элементов;
IN datatype – тип извлекаемых элементов;
IN comm – коммуникатор.

Функция MPI_Unpack извлекает outcount элементов типа datatype из побайтного представления элементов в массиве inbuf, начиная с адреса position. После возврата из функции параметр position увеличивается на размер распакованного сообщения. Результат распаковки помещается в область памяти с начальным адресом outbuf.

Для посылки элементов разного типа из нескольких областей памяти их следует предварительно запаковать в один массив, последовательно обращаясь к функции упаковки MPI_Pack. При первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера. Для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова.

Упакованный буфер пересылается любыми коммуникационными операциями с указанием типа MPI_PACKED и коммуникатора comm, который использовался при обращениях к функции MPI_Pack.

Полученное упакованное сообщение распаковывается в различные массивы или переменные. Это реализуется последовательными вызовами функции распаковки MPI_Unpack с указанием числа элементов, которое следует извлечь при каждом вызове, и с передачей значения position, возвращенного предыдущим вызовом. При первом вызове функции параметр position следует установить в 0. В общем случае, при первом обращении должно быть установлено то значение параметра position, которое было использовано при первом обращении к функции упаковки данных. Очевидно, что для правильной распаковки данных очередность извлечения данных должна быть той же самой, как и упаковки.

Функция MPI_Pack_size помогает определить размер буфера, необходимый для упаковки некоторого количества данных типа datatype.

C:

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm
                  comm, int *size)
```

FORTRAN:

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

IN incount – число элементов, подлежащих упаковке;
IN datatype – тип элементов, подлежащих упаковке;
IN comm – коммуникатор;
OUT size – размер сообщения в байтах после его упаковки.

Первые три параметра функции MPI_Pack_size такие же, как у функции MPI_Pack. После обращения к функции параметр size будет содержать размер сообщения в байтах после его упаковки.

Рассмотрим пример рассылки разнотипных данных из 0-го процесса с использованием функций MPI_Pack и MPI_Unpack.

```
char buff[100];
double x, y;
int  position, a[2];
{
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{   /* Упаковка данных*/
position = 0;
MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position,
MPI_COMM_WORLD);
MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position,
MPI_COMM_WORLD);
MPI_Pack(a, 2, MPI_INT, buff, 100, &position, MPI_COMM_WORLD);
}
   /* Рассылка упакованного сообщения */
MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);
   /* Распаковка сообщения во всех процессах */
if (myrank != 0)
position = 0;
MPI_Unpack(buff, 100, &position, &x, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, &y, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
MPI_Unpack(buff, 100, &position, a, 2, MPI_INT,
MPI_COMM_WORLD);
```

Глава 11.
РАБОТА С ГРУППАМИ И КОММУНИКАТОРАМИ

11.1. Определение основных понятий

Часто в приложениях возникает потребность ограничить область коммуникаций некоторым набором процессов, которые составляют подмножество исходного набора. Для выполнения каких-либо коллективных операций внутри этого подмножества из них должна быть сформирована своя область связи, описываемая своим коммуникатором. Для решения таких задач MPI поддерживает два взаимосвязанных механизма. Во-первых, имеется набор функций для работы с группами процессов как упорядоченными множествами, и, во-вторых, набор функций для работы с коммуникаторами для создания новых коммуникаторов как описателей новых областей связи.

Группа представляет собой упорядоченное множество процессов. Каждый процесс идентифицируется переменной целого типа. Идентификаторы процессов образуют непрерывный ряд, начинающийся с 0. В MPI вводится специальный тип данных MPI_Group и набор функций для работы с переменными и константами этого типа. Существует две предопределенных группы:

MPI_GROUP_EMPTY – группа, не содержащая ни одного процесса;
MPI_GROUP_NULL – значение, возвращаемое в случае, когда группа не может быть создана.

Созданная группа не может быть модифицирована (расширена или усечена), может быть только создана новая группа. Интересно отметить, что при инициализации MPI не создается группы, соответствующей коммуникатору MPI_COMM_WORLD. Она должна создаваться специальной функцией явным образом.

Коммуникатор представляет собой скрытый объект с некоторым набором атрибутов, а также правилами его создания, использования и уничтожения. Коммуникатор описывает некоторую область связи. Одной и

той же области связи может соответствовать несколько коммуникаторов, но даже в этом случае они не являются тождественными и не могут участвовать во взаимном обмене сообщениями. Если данные посылаются через один коммуникатор, процесс-получатель может получить их только через тот же самый коммуникатор.

В MPI существует два типа коммуникаторов:

intracommunicator – описывает область связи некоторой группы процессов;
intercommunicator – служит для связи между процессами двух различных групп.

Тип коммуникатора можно определить с помощью специальной ***функции MPI_Comm_test_inter.***

```
C:
MPI_Comm_test_inter(MPI_Comm comm, int *flag)
FORTRAN:
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
INTEGER COMM, IERROR
LOGICAL FLAG
```

IN comm – коммуникатор;
OUT flag – возвращает true, если comm – intercommunicator.

Функция возвращает значение "истина", если коммуникатор является inter коммуникатором.

При инициализации MPI создается два предопределенных коммуникатора:

MPI_COMM_WORLD – описывает область связи, содержащую все процессы;
MPI_COMM_SELF – описывает область связи, состоящую из одного процесса.

11.2. Функции работы с группами

Функция определения числа процессов в группе MPI_Group_size

```
C:
MPI_Group_size(MPI_Group group, int *size)
FORTRAN:
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
INTEGER GROUP, SIZE, IERROR
```

IN group – группа;
OUT size – число процессов в группе.

Функция возвращает число процессов в группе. Если group = MPI_GROUP_EMPTY, тогда size = 0.

Функция определения номера процесса в группе MPI_Group_rank

C:
```
MPI_Group_rank(MPI_Group group, int *rank)
```
FORTRAN:
```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
INTEGER GROUP, RANK, IERROR
```

IN group – группа;
OUT rank – номер процесса в группе.

Функция MPI_Group_rank возвращает номер в группе процесса, вызвавшего функцию. Если процесс не является членом группы, то возвращается значение MPI_UNDEFINED.

Функция установки соответствия между номерами процессов в двух группах MPI_Group_translate_ranks

C:
```
MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1,
                          MPI_Group group2, int *ranks2)
```
FORTRAN:
```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2,
RANKS2, IERROR)
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

IN group1 – группа1;
IN n – число процессов, для которых устанавливается соответствие;
IN ranks1 – массив номеров процессов из 1-й группы;
IN group2 – группа2;
OUT ranks2 – номера тех же процессов во второй группе.

Функция определяет относительные номера одних и тех же процессов в двух разных группах. Если процесс во второй группе отсутствует, то для него устанавливается значение MPI_UNDEFINED.

Для создания новых групп в MPI имеется 8 функций. Группа может быть создана либо с помощью коммуникатора, либо с помощью операций над множествами процессов других групп.

Функция создания группы с помощью коммуникатора MPI_Comm_group

```
C:
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
FORTRAN:
MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR
```

IN comm – коммуникатор;
OUT group – группа.

Функция создает группу group для множества процессов, входящих в область связи коммуникатора comm.

Следующие три функции имеют одинаковый синтаксис и создают новую группу как результат операции над множествами процессов двух групп.

```
C:
MPI_Group_union(MPI_Group group1, MPI_Group group2,
                 MPI_Group *newgroup)
MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
                 MPI_Group *newgroup)
MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                 MPI_Group *newgroup)
FORTRAN:
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP,
IERROR)
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP,
IERROR)
INTEGER GROUP1, GROUP2,NEWGROUP, IERROR
```

IN group1 – первая группа;
IN group2 – вторая группа;
OUT newgroup – новая группа.

Операции определяются следующим образом:

Union – формирует новую группу из элементов 1-й группы и из элементов 2-й группы, не входящих в 1-ю (*объединение множеств*).

Intersection – новая группа формируется из элементов 1-й группы, которые входят также и во 2-ю. Упорядочивание, как в 1-й группе (*пересечение множеств*).

Difference – новую группу образуют все элементы 1-й группы, которые не входят во 2-ю. Упорядочивание, как в 1-й группе (*дополнение множеств*).

Созданная группа может быть пустой, что эквивалентно MPI_GROUP_EMPTY.

Новые группы могут быть созданы с помощью различных выборок из существующей группы. Следующие две функции имеют одинаковый синтаксис, но являются дополнительными по отношению друг к другу.

```
C:
MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group
                    *newgroup)
MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group
                    *newgroup)
FORTRAN:
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

IN	group	– существующая группа;
IN	n	– число элементов в массиве ranks;
IN	ranks	– массив номеров процессов;
OUT	newgroup	– новая группа.

Функция MPI_Group_incl создает новую группу, которая состоит из процессов существующей группы, перечисленных в массиве ranks. Процесс с номером i в новой группе есть процесс с номером ranks[i] в существующей группе. Каждый элемент в массиве ranks должен иметь корректный номер в группе group, и среди этих элементов не должно быть совпадающих.

Функция MPI_Group_excl создает новую группу из тех процессов group, которые не перечислены в массиве ranks. Процессы упорядочиваются, как в группе group. Каждый элемент в массиве ranks должен иметь корректный номер в группе group, и среди них не должно быть совпадающих.

Две следующие функции по смыслу совпадают с предыдущими, но используют более сложное формирование выборки. Массив ranks заменяется двумерным массивом ranges, представляющим собой набор триплетов для задания диапазонов процессов.

```
C:
MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
                     MPI_Group *newgroup)
MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
                     MPI_Group *newgroup)
FORTRAN:
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

Каждый триплет имеет вид: нижняя граница, верхняя граница, шаг.

Уничтожение созданных групп выполняется ***функцией MPI_Group_free.***

```
C:
MPI_Group_free(MPI_Group *group)
FORTRAN:
MPI_GROUP_FREE(GROUP, IERROR)
INTEGER GROUP, IERROR
```

INOUT group – уничтожаемая группа.

11.3. Функции работы с коммуникаторами

В данном разделе рассматриваются функции работы с коммуникаторами. Они разделяются на функции доступа к коммуникаторам и функции создания коммуникаторов. Функции доступа являются локальными и не требуют коммуникаций, в отличие от функций создания, которые являются коллективными и могут потребовать межпроцессорных коммуникаций.

Две основные функции доступа к коммуникатору (*MPI_Comm_size* – опрос числа процессов в области связи и *MPI_Comm_rank* – опрос идентификатора, или номера, процесса в области связи) были рассмотрены

в самом начале главы 7 среди базовых функций MPI. Кроме них, имеется ***функция сравнения двух коммуникаторов MPI_Comm_compare***.

```
C:
MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int
                    *result)
FORTRAN:
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
INTEGER COMM1, COMM2, RESULT, IERROR
```

IN comm1 – первый коммуникатор;
IN comm2 – второй коммуникатор;
OUT result – результат сравнения.

Возможные значения результата сравнения:

MPI_IDENT – коммуникаторы идентичны, представляют один и тот же объект;
MPI_CONGRUENT – коммуникаторы конгруэнтны, две области связи с одними и теми же атрибутами группы;
MPI_SIMILAR – коммуникаторы подобны, группы содержат одни и те же процессы, но другое упорядочивание;
MPI_UNEQUAL – во всех других случаях.

Создание нового коммуникатора возможно с помощью одной из трех функций: MPI_Comm_dup, MPI_Comm_create, MPI_Comm_split.

Функция дублирования коммуникатора MPI_Comm_dup

```
C:
MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
FORTRAN:
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
```

IN comm – коммуникатор;
OUT newcomm – копия коммуникатора.

Функция полезна для последующего создания коммуникаторов с новыми атрибутами.

Функция создания коммуникатора MPI_Comm_create

```
C:
MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                  MPI_Comm *newcomm)
FORTRAN:
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
```

```
INTEGER COMM, GROUP, NEWCOMM, IERROR
```

IN comm – родительский коммуникатор;
IN group – группа, для которой создается коммуникатор;
OUT newcomm – новый коммуникатор.

Эта функция создает коммуникатор для группы group. Для процессов, которые не являются членами группы, возвращается значение MPI_COMM_NULL. Функция возвращает код ошибки, если группа group не является подгруппой родительского коммуникатора.

Функция расщепления коммуникатора MPI_Comm_split

```
C:
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm
                    *newcomm)
FORTRAN:
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

IN comm – родительский коммуникатор;
IN color – признак подгруппы;
IN key – управление упорядочиванием;
OUT newcomm – новый коммуникатор.

Функция расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по одной на каждое значение признака подгруппы color. Значение color должно быть неотрицательным. Каждая подгруппа содержит процессы с одним и тем же значением color. Параметр key управляет упорядочиванием внутри новых групп: меньшему значению key соответствует меньшее значение идентификатора процесса. В случае равенства параметра key для нескольких процессов упорядочивание выполняется в соответствии с порядком в родительской группе. Приведем алгоритм расщепления группы из восьми процессов на три подгруппы и его графическую интерпретацию (рис.11.1).

```
MPI_comm comm, newcomm;
int  myid, color;
……
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```

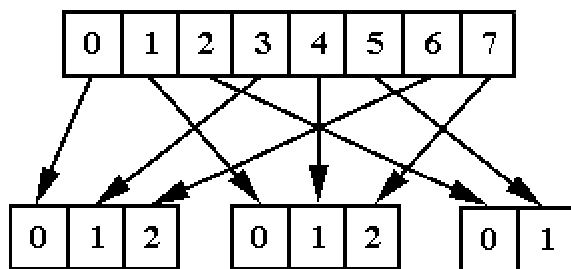

Рис. 11.1. Разбиение группы из восьми процессов на три подгруппы.

В данном примере первую подгруппу образовали процессы, номера которых делятся на 3 без остатка, вторую, для которых остаток равен 1, и третью, для которых остаток равен 2. Отметим, что после выполнения функции MPI_Comm_split значения коммуникатора newcomm в процессах разных подгрупп будут отличаться.

Функция уничтожения коммуникатора MPI_Comm_free

```
C:
MPI_Comm_free(MPI_Comm *comm)
FORTRAN:
MPI_COMM_FREE(COMM, IERROR)
INTEGER COMM, IERROR
```

IN comm – уничтожаемый коммуникатор.

Примечание: За рамками данной книги мы оставим обсуждение inter-коммуникаторов и вопросы, связанные с изменением или добавлением новых атрибутов коммуникаторов.

Глава 12.
ТОПОЛОГИЯ ПРОЦЕССОВ

12.1. Основные понятия

Топология процессов является одним из необязательных атрибутов коммуникатора. Такой атрибут может быть присвоен только intra-коммуникатору. По умолчанию предполагается линейная топология, в которой процессы пронумерованы в диапазоне от 0 до n-1, где n – число процессов в группе. Однако для многих задач линейная топология

неадекватно отражает логику коммуникационных связей между процессами. MPI предоставляет средства для создания достаточно сложных “виртуальных” топологий в виде графов, где узлы являются процессами, а грани – каналами связи между процессами. Конечно же, следует различать логическую топологию процессов, которую позволяет формировать MPI, и физическую топологию процессоров. В идеале логическая топология процессов должна учитывать как алгоритм решения задачи, так и физическую топологию процессоров. Для очень широкого круга задач наиболее адекватной топологией процессов является двумерная или трехмерная сетка. Такие структуры полностью определяются числом измерений и количеством процессов вдоль каждого координатного направления, а также способом раскладки процессов на координатную сетку. В MPI, как правило, используется row-major нумерация процессов, т.е. используется нумерация вдоль строки. На рис. 12.1 представлено соответствие между нумерациями 6-ти процессов в одномерной и двумерной (2×3) топологиях.

Row-major

0 (0,0)	1 (0,1)	2 (0,2)
3 (1,0)	4 (1,1)	5 (1,2)

Column-major

0 (0,0)	2 (0,1)	4 (0,2)
1 (1,0)	3 (1,1)	5 (1,2)

Рис. 12.1. Соотношение между идентификатором процесса (верхнее число) и координатами в двумерной сетке 2×3 (нижняя пара чисел).

12.2. Декартова топология

Обобщением линейной и матричной топологий на произвольное число измерений является декартова топология. Для создания коммуникатора с декартовой топологией используется функция MPI_Cart_create. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в

отдельности можно накладывать периодические граничные условия. Таким образом, для одномерной топологии мы можем получить или линейную структуру, или кольцо в зависимости от того, какие граничные условия будут наложены. Для двумерной топологии, соответственно, либо прямоугольник, либо цилиндр, либо тор. Заметим, что не требуется специальной поддержки гиперкубовой структуры, поскольку она представляет собой n-мерный тор с двумя процессами вдоль каждого координатного направления.

Функция создания коммуникатора с декартовой топологией MPI_Cart_create

```
C:
MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                   int *periods, int reorder, MPI_Comm *comm_cart)
FORTRAN:
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,
REORDER, COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```

IN comm_old – родительский коммуникатор;
IN ndims – число измерений;
IN dims – массив размера ndims, в котором задается число процессов вдоль каждого измерения;
IN periods – логический массив размера ndims для задания граничных условий (true – периодические, false – непериодические)
IN reorder – логическая переменная указывает, производить перенумерацию процессов (true) или нет (false);
OUT comm_cart – новый коммуникатор.

Функция является коллективной, т.е. должна запускаться на всех процессах, входящих в группу коммуникатора comm_old. При этом если какие-то процессы не попадают в новую группу, то для них возвращается результат MPI_COMM_NULL. В случае, когда размеры заказываемой сетки больше имеющегося в группе числа процессов, то функция завершается аварийно. Значение параметра reorder=false означает, что идентификаторы всех процессов в новой группе будут такими же, как в

старой группе. Если reorder=true, то MPI будет пытаться перенумеровать их с целью оптимизации коммуникаций.

Остальные функции, которые будут рассмотрены в этом разделе, имеют вспомогательный или информационный характер.

Функция определения оптимальной конфигурации сетки MPI_Dims_create

```
C:
MPI_Dims_create(int nnodes, int ndims, int *dims)
FORTRAN:
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

IN	nnodes	– общее число узлов в сетке;
IN	ndims	– число измерений;
INOUT	dims	– массив целого типа размерности ndims, в который помещается рекомендуемое число процессов вдоль каждого измерения.

На входе в процедуру в массив dims должны быть занесены целые неотрицательные числа. Если элементу массива dims[i] присвоено положительное число, то для этой размерности вычисление не производится (число процессов вдоль этого направления считается заданным). Вычисляются только те компоненты dims[i], для которых перед обращением к процедуре были присвоены значения 0. Функция стремится создать максимально равномерное распределение процессов вдоль направлений, выстраивая их по убыванию, т.е. для 12-ти процессов она построит трехмерную сетку 4×3×1. Результат работы этой процедуры может использоваться в качестве входного параметра для процедуры MPI_Cart_create.

Функция опроса числа измерений декартовой топологии MPI_Cartdim_get

```
C:
MPI_Cartdim_get(MPI_Comm comm, int *ndims)
FORTRAN:
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
```

```
INTEGER COMM, NDIMS, IERROR
```

IN comm – коммуникатор с декартовой топологией;
OUT ndims – число измерений в декартовой топологии.

Функция возвращает число измерений в декартовой топологии ndims для коммуникатора comm.

Результат может быть использован в качестве параметра для вызова ***функции MPI_Cart_get***, которая служит ***для получения более детальной информации.***

C:

```
MPI_Cart_get(MPI_Comm comm, int ndims, int *dims, int *periods,
             int *coords)
```

FORTRAN:

```
MPI_CART_GET(COMM, NDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, NDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)
```

IN comm – коммуникатор с декартовой топологией;
IN ndims – число измерений;
OUT dims – массив размера ndims, в котором возвращается число процессов вдоль каждого измерения;
OUT periods – логический массив размера ndims, в котором возвращаются наложенные граничные условия (true – периодические, false – непериодические);
OUT coords – координаты в декартовой сетке вызывающего процесса.

Две следующие функции устанавливают соответствие между идентификатором процесса и его координатами в декартовой сетке. Под идентификатором процесса понимается его номер в исходной области связи, на основе которой была создана декартова топология.

Функция получения идентификатора процесса по его координатам MPI_Cart_rank

C:

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

FORTRAN:

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
INTEGER COMM, COORDS(*), RANK, IERROR
```

IN comm – коммуникатор с декартовой топологией;

IN coords – координаты в декартовой системе;
OUT rank – идентификатор процесса.

Для измерений с периодическими граничными условиями будет выполняться приведение к основной области определения 0 <=coords(i) < dims(i).

Функция определения координат процесса по его идентификатору MPI_Cart_coords

```
C:
MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
                int *coords)
FORTRAN:
MPI_CART_COORDS(COMM, RANK, NDIMS, COORDS, IERROR)
INTEGER COMM, RANK, NDIMS, COORDS(*), IERROR
```

IN comm. – коммуникатор с декартовой топологией;
IN rank – идентификатор процесса;
IN ndims – число измерений;
OUT coords – координаты процесса в декартовой топологии.

Во многих численных алгоритмах используется операция сдвига данных вдоль каких-то направлений декартовой решетки. В MPI существует специальная функция MPI_Cart_shift, реализующая эту операцию. Точнее говоря, сдвиг данных осуществляется с помощью функции MPI_Sendrecv, а функция MPI_Cart_shift вычисляет для каждого процесса параметры для функции MPI_Sendrecv (source и dest).

Функция сдвига данных MPI_Cart_shift

```
C:
MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
               int *rank_source, int *rank_dest)
FORTRAN:
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE,
RANK_DEST, IERROR)
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST,
IERROR
```

IN comm – коммуникатор с декартовой топологией;
IN direction – номер измерения, вдоль которого выполняется сдвиг;

IN disp – величина сдвига (может быть как положительной, так и отрицательной).
OUT rank_source – номер процесса, от которого должны быть получены данные;
OUT rank_dest – номер процесса, которому должны быть посланы данные.

Номер измерения и величина сдвига не обязаны быть одинаковыми для всех процессов. В зависимости от граничных условий сдвиг может быть либо циклический, либо с учетом граничных процессов. В последнем случае для граничных процессов возвращается MPI_PROC_NULL либо для переменной rank_source, либо для rank_dest. Это значение также может быть использовано при обращении к функции MPI_sendrecv.

Другая часто используемая операция – выделение в декартовой топологии подпространств меньшей размерности и связывание с ними отдельных коммуникаторов.

Функция выделения подпространства в декартовой топологии MPI_Cart_sub

C:
```
MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
              *newcomm)
```
FORTRAN:
```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
LOGICAL REMAIN_DIMS(*)
```

IN comm – коммуникатор с декартовой топологией;
IN remain_dims – логический массив размера ndims, указывающий, входит ли i-е измерение в новую подрешетку (remain_dims[i] = true);
OUT newcomm – новый коммуникатор, описывающий подрешетку, содержащую вызывающий процесс.

Функция является коллективной. Действие функции проиллюстрируем следующим примером. Предположим, что имеется декартова решетка 2×3×4, тогда обращение к функции MPI_Cart_sub с массивом remain_dims (true, false, true) создаст три коммуникатора с топологией

2×4. Каждый из коммуникаторов будет описывать область связи, состоящую из 1/3 процессов, входивших в исходную область связи.

Кроме рассмотренных функций в MPI входит набор из 6 функций для работы с коммуникаторами с топологией графов, которые мы рассматривать не будем.

Для определения топологии коммуникатора служит ***функция MPI_Topo_test.***

```
C:
MPI_Topo_test(MPI_Comm comm, int *status)
FORTRAN:
MPI_TOPO_TEST(COMM, STATUS, IERROR)
INTEGER COMM, STATUS, IERROR
```

IN comm – коммуникатор;
OUT status – топология коммуникатора.

Функция MPI_Topo_test возвращает через переменную status топологию коммуникатора comm. Возможные значения:

MPI_GRAPH – топология графа;
MPI_CART – декартова топология;
MPI_UNDEFINED – топология не задана

Глава 13.
ПРИМЕРЫ ПРОГРАММ

13.1. Вычисление числа π

В качестве первого примера применения коммуникационной библиотеки MPI рассмотрим программу вычисления числа π (последовательная и параллельная версия программы, написанная с использованием средств программирования PSE nCUBE2, приведены в главе 4). Данный пример хорошо иллюстрирует общность подходов к разработке параллельных программ в различных средах параллельного программирования. Во многих случаях имена одних функций просто меняются на имена других идентичным им функций. В частности, если

сопоставить параллельную версию программы вычисления числа π с использованием средств PSE и предлагаемую ниже программу, то мы увидим практически один и тот же набор средств, отличающихся только своей реализацией. Однако при разработке сложных программ среда параллельного программирования MPI предоставляет более широкий спектр средств для реализации параллельных алгоритмов. Как и в примере с использованием среды параллельного программирования PSE жирным шрифтом будем выделять изменения в программе по сравнению с однопроцессорной версией.

```
      program calc_pi
      include 'mpif.h'
      integer i, n
      double precision w, gsum, sum
      double precision v
      integer np, myid, ierr
      real*8 time, mflops, time1, time2, dsecnd
c  Инициализация MPI и определение процессорной конфигурации
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, np, ierr )
c  Информацию с клавиатуры считывает 0-й процессор
      if ( myid .eq. 0 ) then
      print *, 'Введите число точек разбиения интервала : '
      read *, n
      time1 = MPI_Wtime()
      endif
c  Рассылка числа точек разбиения всем процессорам
      call MPI_BCAST(n,1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
c  Вычисление частичной суммы на процессоре
      w = 1.0 / n
      sum = 0.0d0

      do i = myid+1, n, np
        v = (i - 0.5d0 ) * w
        v = 4.0d0 / (1.0d0 + v * v)
        sum = sum + v
      end do
c  Суммирование частичных сумм с сохранением результата в 0-м
c  процессоре
      call MPI_REDUCE(sum, gsum, 1, MPI_DOUBLE_PRECISION,
     $       MPI_SUM, 0, MPI_COMM_WORLD, ierr)
c  Печать выходной информации с 0-го процессора
      if (myid .eq. 0) then
      time2 = MPI_Wtime()
      time   = time2 - time1
      mflops = 9 * n / (1000000.0 * time)
```

```
      print *, 'pi ist approximated with ', gsum *w
      print *, 'time   = ', time, ' seconds'
      print *, 'mflops = ', mflops, ' on ', np, ' processors'
      print *, 'mflops = ', mflops/np, ' for one processor'
      endif
c  Закрытие MPI
      call MPI_FINALIZE(ierr)
      end
```

13.2. Перемножение матриц

Рассмотренный в предыдущем разделе пример представляет наиболее простой для распараллеливания тип задач, в которых в процессе выполнения подзадач не требуется выполнять обмен информацией между процессорами. Такая ситуация имеет место всегда, когда переменная распараллеливаемого цикла не индексирует какие-либо массивы (типичный случай – параметрические задачи). В задачах линейной алгебры, в которых вычисления связаны с обработкой массивов, часто возникают ситуации, когда необходимые для вычисления матричные элементы отсутствуют на обрабатывающем процессоре. Тогда процесс вычисления не может быть продолжен до тех пор, пока они не будут переданы в память нуждающегося в них процессора. В качестве простейшего примера задач этого типа рассмотрим задачу перемножения матриц.

Достаточно подробно проблемы, возникающие при решении этой задачи, рассматривались в главе 3 (раздел 3.3). В основном речь шла об однопроцессорном варианте программы, но были также приведены данные по производительности для двух вариантов параллельных программ. Первый вариант был получен распараллеливанием обычной однопроцессорной программы, без использования оптимизированных библиотечных подпрограмм. В этом разделе мы рассмотрим именно эту программу. Параллельная программа, которая базируется на библиотечных подпрограммах библиотеки ScaLAPACK, будет рассмотрена в части 3.

Существует множество вариантов решения этой задачи на многопроцессорных системах. Алгоритм решения существенным образом

зависит от того, производится или нет распределение матриц по процессорам, и какая топология процессоров при этом используется. Как правило, задачи такого типа решаются либо на одномерной сетке процессоров, либо на двумерной. На рис. 13.1 представлено распределение матрицы в памяти 4-х процессоров, образующих одномерную и двумерную сетки.

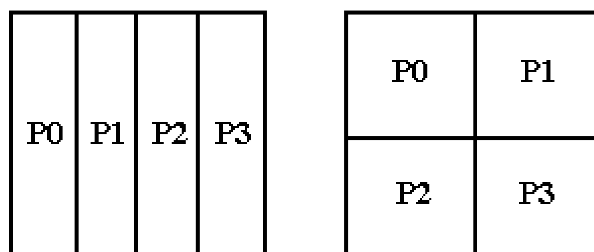

Рис. 13.1. Пример распределения матрицы на одномерную и двумерную сетки процессоров.

Первый вариант значительно проще в использовании, поскольку позволяет работать с заданным по умолчанию коммуникатором. В случае двумерной сетки потребуется описать создаваемую топологию и коммуникаторы для каждого направления сетки. Каждая из трех матриц (A, B и C) может быть распределена одним из 4-х способов:

- не распределена по процессорам;
- распределена на двумерную сетку;
- распределена по столбцам на одномерную сетку;
- распределена по строкам на одномерную сетку.

Отсюда возникает 64 возможных вариантов решения этой задачи. Большинство из этих вариантов плохо отражают специфику алгоритма и, соответственно, заведомо неэффективны. Тот или иной способ распределения матриц однозначно определяет, какие из трех циклов вычислительного блока должны быть подвержены процедуре редукции.

Ниже предлагается вариант программы решения этой задачи, который в достаточно полной мере учитывает специфику алгоритма. Поскольку для вычисления каждого матричного элемента матрицы C

необходимо выполнить скалярное произведение строки матрицы A на столбец матрицы B, то матрица A разложена на одномерную сетку процессоров по строкам, а матрица B – по столбцам. Матрица C разложена по строкам, как матрица A (рис. 13.2).

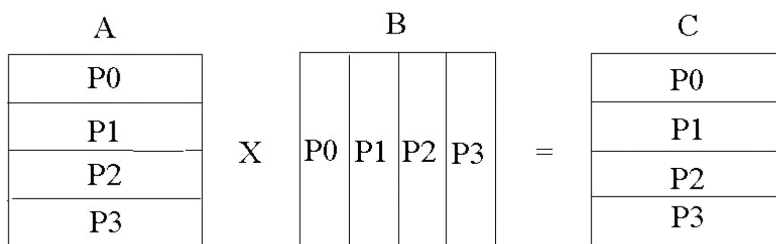

Рис. 13.2. Распределение матриц на одномерную сетку процессоров.

При таком распределении строка, необходимая для вычисления некоторого матричного элемента, гарантированно находится в данном процессоре, а столбец хотя и может отсутствовать, но целиком расположен в некотором процессоре. Поэтому алгоритм решения задачи должен предусматривать определение, в каком процессоре находится нужный столбец матрицы B, и пересылку его в тот процессор, который в нем нуждается в данный момент. На самом деле, каждый столбец матрицы B участвует в вычислении всего столбца матрицы C, и поэтому его следует рассылать во все процессоры.

```
      PROGRAM PMATMULT
      INCLUDE 'mpif.h'
C  Параметры:
C  NM    – полная размерность матриц;
C  NPMIN – минимальное число процессоров для решения задачи
C  NPS   – размерность локальной части матриц
      PARAMETER (NM = 500, NPMIN=4, NPS=NM/NPMIN+1)
      REAL*8 A(NPS,NM), B(NM,NPS), C(NPS,NM), COL(NM)
      REAL*8 TIME
C  В массивах NB и NS информация о декомпозиции матриц:
C  NB  – число строк матрицы в каждом процессоре;
C  NS  – номер строки, начиная с которого хранится матрица в данном
C        процессоре;
C  Предполагается, что процессоров не больше 64.
      INTEGER  NB(0:63), NS(0:63)
C  Инициализация MPI
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,IAM,IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NPROCS,IERR)
```

```
      IF (IAM.EQ.0) WRITE(*,*) 'NM = ',NM,' NPROCS = ',NPROCS
C  Вычисление параметров декомпозиции матриц.
C  Алгоритм реализует максимально равномерное распределение
      NB1 = NM/NPROCS
      NB2 = MOD(NM,NPROCS)
      DO I = 0,NPROCS-1
      NB(I) = NB1
      END DO
      DO I = 0,NB2-1
      NB(I)= NB(I)+1
      END DO

      NS(0)=0
      DO I = 1,NPROCS-1
      NS(I)= NS(I-1) + NB(I-1)
      END DO

C  Заполнение матрицы A,  значения матричных элементов некоторой
C  строки равны глобальному индексу этой строки. Здесь IAM – номер
C  процессора
      DO J = 1,NM
      DO I = 1,NB(IAM)
      A(I,J) = DBLE(I+NS(IAM))
      END DO
      END DO
C  Заполнение матрицы B(I,J)=1/J (подразумеваются глобальные индексы)
      DO I = 1,NM
      DO J = 1,NB(IAM)
      B(I,J) =1./DBLE(J+NS(IAM))
      END DO
      END DO
C  Включение таймера
      TIME = MPI_WTIME()
C  Блок вычисления
C  Циклы по строкам и по столбцам переставлены местами и цикл по
C  столбцам разбит на две части: по процессорам J1 и по элементам
C  внутри процессора J2; это сделано для того, чтобы не вычислять,
C  в каком процессоре находится данный столбец.
C  Переменная J выполняет сквозную нумерацию столбцов.

C  Цикл по столбцам
      J = 0
      DO J1 = 0,NPROCS-1
      DO J2 = 1,NB(J1)
      J = J + 1
C  Процессор, хранящий очередной столбец, рассылает его всем
C  остальным процессорам
      IF (IAM.EQ.J1) THEN
      DO N = 1,NM
      COL(N) = B(N,J2)
      END DO
      END IF
```

```
      CALL MPI_BCAST(COL,NM,MPI_DOUBLE_PRECISION,J1,
     *MPI_COMM_WORLD,IERR)
C  Цикл по строкам (именно он укорочен)
      DO I = 1,NB(IAM)
      C(I,J) = 0.0
C  Внутренний цикл
      DO K = 1,NM
      C(I,J) = C(I,J) + A(I,K)*COL(K)
      END DO
      END DO
      END DO
      END DO
      TIME = MPI_WTIME() -TIME
C  Печать контрольных угловых матричных элементов матрицы C из тех
C  процессоров, где они хранятся
      IF (IAM.EQ.0) WRITE(*,*) IAM,C(1,1),C(1,NM)
      IF (IAM.EQ.NPROCS-1)
     *WRITE(*,*) IAM,C(NB(NPROCS-1),1),C(NB(NPROCS-1),NM)
      IF (IAM.EQ.0) WRITE(*,*) ' TIME CALCULATION: ', TIME
      CALL MPI_FINALIZE(IERR)
      END
```

В отличие от программы вычисления числа π, в этой программе практически невозможно выделить изменения по сравнению с однопроцессорным вариантом. По сути дела – это совершенно новая программа, имеющая очень мало общего с прототипом. Распараллеливание в данной программе заключается в том, что каждый процессор вычисляет свой блок матрицы C, который составляет приблизительно 1/NPROCS часть полной матрицы. Нетрудно заметить, что пересылки данных не потребовались бы, если бы матрица B не распределялась по процессорам, а целиком хранилась в каждом процессоре. В некоторых случаях такая асимметрия в распределении матриц бывает очень выгодна.

13.3. Решение краевой задачи методом Якоби

Еще одна область задач, для которых достаточно хорошо разработана технология параллельного программирования – это краевые задачи. В этом случае используется техника декомпозиции по процессорам расчетной области, как правило, с перекрытием подобластей. На рис. 13.3 представлено такое разложение исходной расчетной области на 4 процессора с топологией одномерной сетки. Заштрихованные области на

каждом процессоре обозначают те точки, в которых расчет не производится, но они необходимы для выполнения расчета в пограничных точках. В них должна быть предварительно помещена информация с пограничного слоя соседнего процессора.

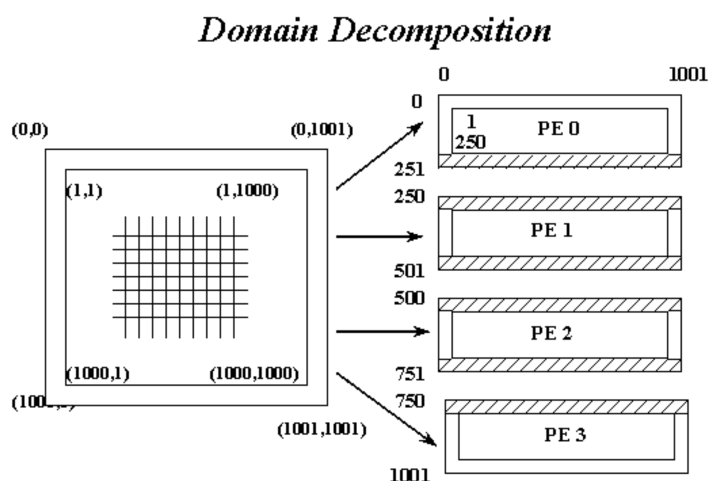

Рис. 13.3. Пример декомпозиции двумерной расчетной области.

Приведем пример программы решения уравнения Лапласа методом Якоби на двумерной регулярной сетке. В программе декомпозиция области выполнена не по строкам, как на рисунке, а по столбцам. Так удобнее при программировании на языке FORTRAN, на С удобнее разбиение производить по строкам (это определяется способом размещения матриц в памяти компьютера).

```
C  Программа решения уравнения Лапласа методом Якоби
C  с использованием функции MPI_Sendrecv и NULL процессов
      PROGRAM JACOBI
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INTEGER n,m,npmin,nps,itmax
C  Параметры:
C   n        – количество точек области в каждом направлении;
C   npmin    – минимальное число процессоров для решения задачи
C   nps      – число столбцов локальной части матрицы. Этот параметр
C              введен в целях экономии памяти.
C   itmax    – максимальное число итераций, если сходимость не будет
C              достигнута
      PARAMETER (n = 400,npmin=1,nps=n/npmin+1, itmax = 1000)
      REAL*8 A(0:n+1,0:nps+1), B(n,nps)
      REAL*8 diffnorm, gdiffnorm, eps, time
      INTEGER left, right, i, j, k, itcnt, status(0:3), tag
      INTEGER IAM, NPROCS, ierr
      LOGICAL converged
```

```
C  Определение числа процессоров, выделенных задаче (NPROCS), и
C  номера текущего процессора (IAM)
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS, ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, IAM, ierr)
C  Установка критерия достижения сходимости
      eps = 0.01
C  Вычисление числа столбцов, обрабатываемых процессором
      m = n/NPROCS
      IF (IAM.LT.(n-NPROCS*m)) THEN
      m = m+1
      END IF
      time = MPI_Wtime()
C  Задание начальных и граничных значений
      do j = 0,m+1
      do i = 0,n+1
        a(i,j) = 0.0
      end do
      end do

      do j = 0,m+1
      A(0,j) = 1.0
      A(n+1,j) = 1.0
      end do
      IF (IAM.EQ.0) then
      do i = 0,n+1
      A(i,0) = 1.0
      end do
      end if
      IF (IAM.EQ.NPROCS-1) then
      do i = 0,n+1
      A(i,m+1) = 1.0
      end do
      end if
C  Определение номеров процессоров слева и справа. Если таковые
C  отсутствуют, то им присваивается значение MPI_PROC_NULL
C  (для них коммуникационные операции  игнорируются)
      IF (IAM.EQ.0) THEN
      left = MPI_PROC_NULL
      ELSE
      left = IAM - 1
      END IF
      IF (IAM.EQ.NPROCS-1) THEN
      right = MPI_PROC_NULL
      ELSE
      right = IAM+1
      END IF
      tag = 100
      itcnt = 0
      converged = .FALSE.
```

```
C  Цикл по итерациям
      DO k = 1,itmax
      diffnorm = 0.0
      itcnt = itcnt + 1
C  Вычисление новых значений функции по 5-точечной схеме
      DO j = 1, m
      DO i = 1, n
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
        diffnorm = diffnorm + (B(i,j)-A(i,j))**2
      END DO
      END DO
C  Переприсваивание внутренних точек области
      DO j = 1, m
      DO i = 1, n
         A(i,j) = B(i,j)
      END DO
      END DO

C  Пересылка граничных значений в соседние процессоры
      CALL MPI_SENDRECV(B(1,1),n, MPI_DOUBLE_PRECISION, left, tag,
     $ A(1,0), n, MPI_DOUBLE_PRECISION, left, tag, MPI_COMM_WORLD,
     $ status, ierr)

      CALL MPI_SENDRECV(B(1,m), n, MPI_DOUBLE_PRECISION, right,
     $ tag, A(1,m+1), n, MPI_DOUBLE_PRECISION, right, tag,
     $ MPI_COMM_WORLD, status, ierr)
C  Вычисление невязки и проверка условия достижения сходимости
      CALL MPI_Allreduce(diffnorm, gdiffnorm, 1, MPI_DOUBLE_PRECISION,
     $ MPI_SUM, MPI_COMM_WORLD, ierr )
      gdiffnorm = sqrt( gdiffnorm )
      converged = gdiffnorm.LT.eps
      if (converged) goto 777

      END DO
 777  continue
      time = MPI_Wtime() - time
      IF (IAM.EQ.0) then
      WRITE(*,*) ' At iteration ',itcnt, '  diff is ', gdiffnorm
      WRITE(*,*) ' Time calculation: ', time
      END IF
      CALL MPI_Finalize(ierr)
      stop
      end
```

ЗАКЛЮЧЕНИЕ К ЧАСТИ 2

Приведенные примеры показывают, что при написании параллельных программ с использованием механизма передачи сообщений алгоритмы решения даже простейших задач, таких как, например, перемножения матриц, перестают быть тривиальными. И совсем уж нетривиальной становится задача написания эффективных программ для решения более сложных задач линейной алгебры. Сложность программирования с использованием механизма передачи сообщений долгое время оставалась основным сдерживающим фактором на пути широкого использования многопроцессорных систем с распределенной памятью. В последние годы ситуация значительно изменилась благодаря появлению достаточно эффективных библиотек подпрограмм для решения широкого круга задач. Такие библиотеки избавляют программистов от рутинной работы по написанию подпрограмм для решения стандартных задач численных методов и позволяют сконцентрироваться на предметной области. Однако использование этих библиотек не избавляет от необходимости ясного понимания принципов параллельного программирования и требует выполнения достаточно объемной подготовительной работы. В связи с этим следующая часть данной книги посвящена описанию библиотек подпрограмм для многопроцессорных систем ScaLAPACK и Aztec, позволяющих решать широкий спектр задач линейной алгебры.

Часть 3.
БИБЛИОТЕКИ ПОДПРОГРАММ ДЛЯ МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Использование многопроцессорных систем для решения сложных вычислительных задач значительно расширяет возможности исследователей, занимающихся компьютерным моделированием сложных физических процессов. Однако, как было показано в предыдущих частях, разработка эффективных программ для многопроцессорных систем, особенно с распределенной памятью, представляет собой достаточно трудную задачу. Значительно облегчают решение этой задачи готовые параллельные подпрограммы для решения стандартных задач численных методов. Как правило, библиотеки таких подпрограмм разрабатываются ведущими специалистами в области численных методов и параллельного программирования. В частности, на всех многопроцессорных системах центра высокопроизводительных вычислений РГУ установлены библиотеки параллельных подпрограмм ScaLAPACK и Aztec, которые широко используются для решения реальных прикладных задач. Однако использование этих библиотек не освобождает программиста от необходимости четкого и ясного понимания принципов параллельного программирования и само по себе далеко не тривиально. Предлагаемая вниманию читателя третья часть призвана помочь в освоении этих пакетов.

Глава 14.
БИБЛИОТЕКА ПОДПРОГРАММ ScaLAPACK

14.1. История разработки пакета ScaLAPACK и его общая организация

Важнейшая роль в численных методах принадлежит решению задач линейной алгебры. Свое отражение это находит в том факте, что все

производители высокопроизводительных вычислительных систем поставляют со своими системами высокооптимизированные библиотеки подпрограмм, включающие главным образом подпрограммы решения задач линейной алгебры. Так на компьютерах фирмы SUN в состав библиотеки High Performance Library входят пакеты BLAS, LINPACK и LAPACK. Аналогичный состав имеет и библиотека Compaq Extended Math Library, устанавливаемая на компьютерах фирмы Compaq на базе процессоров Alpha.

С появлением многопроцессорных систем с распределенной памятью начались работы по переносу на эти платформы библиотеки LAPACK как наиболее полно отвечающей архитектуре современных процессоров (подпрограммы оптимизированы для эффективного использования кэш-памяти). В работе по переносу библиотеки LAPACK участвовали ведущие научные и суперкомпьютерные центры США.

Результатом этой работы было создание пакета подпрограмм ScaLAPACK (Scalable LAPACK) [17]. Проект оказался успешным, и пакет фактически стал стандартом в программном обеспечении многопроцессорных систем. В этом пакете почти полностью сохранены состав и структура пакета LAPACK и практически не изменились обращения к подпрограммам верхнего уровня. В основе успеха реализации этого проекта лежали два принципиально важных решения.

- Во-первых, в пакете LAPACK все элементарные векторные и матричные операции выполняются с помощью высокооптимизированных подпрограмм библиотеки BLAS (Basic Linear Algebra Subprograms). По аналогии с этим при реализации ScaLAPACK была разработана параллельная версия этой библиотеки PBLAS, что избавило от необходимости радикальным образом переписывать подпрограммы верхнего уровня.
- Во-вторых, все коммуникационные операции выполняются с использованием подпрограмм из специально разработанной библиотеки

BLACS (Basic Linear Algebra Communication Subprograms), поэтому перенос пакета на различные многопроцессорные платформы требует настройки только этой библиотеки.

14.2. Структура пакета ScaLAPACK

Общая структура пакета ScaLAPACK представлена на рис. 14.1.

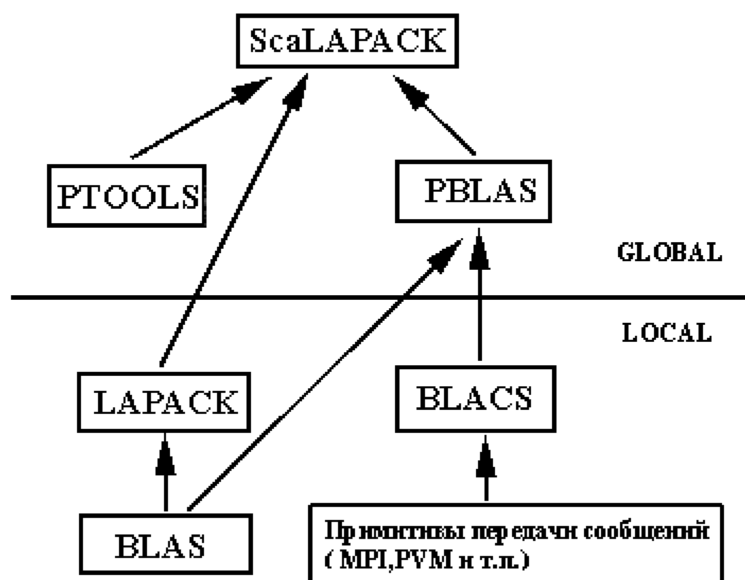

Рис. 14.1. Структура пакета ScaLAPACK.

На этом рисунке компоненты пакета, расположенные выше разделительной линии, содержат подпрограммы, которые выполняются параллельно на некотором наборе процессоров и в качестве аргументов используют векторы и матрицы, распределенные по этим процессорам. Подпрограммы из компонентов пакета ниже разделительной линии вызываются на одном процессоре и работают с локальными данными. Каждый из компонентов пакета – это независимая библиотека подпрограмм, которая не является частью библиотеки ScaLAPACK, но необходима для ее работы. В тех случаях, когда на компьютере имеются оптимизированные фирменные реализации каких-то из этих библиотек (BLAS, LAPACK), то настоятельно рекомендуется для достижения более высокой производительности использовать именно эти реализации.

Собственно сама библиотека ScaLAPACK состоит из 530 подпрограмм, которые для каждого из 4-х типов данных (вещественного,

вещественного с двойной точностью, комплексного, комплексного с двойной точностью) разделяются на три категории:

- Драйверные подпрограммы, каждая из которых решает некоторую законченную задачу, например, решение системы линейных алгебраических уравнений или нахождение собственных значений вещественной симметричной матрицы. Таких подпрограмм 14 для каждого типа данных. Эти подпрограммы обращаются к вычислительным подпрограммам.
- Вычислительные подпрограммы выполняют отдельные подзадачи, например, LU разложение матрицы или приведение вещественной симметричной матрицы к трехдиагональному виду. Набор вычислительных подпрограмм значительно перекрывает функциональные потребности и возможности драйверных подпрограмм.
- Служебные подпрограммы выполняют некоторые внутренние вспомогательные действия.

Имена всех драйверных и вычислительных подпрограмм совпадают с именами соответствующих подпрограмм из пакета LAPACK, с той лишь разницей, что в начале имени добавляется символ P, указывающий на то, что это параллельная версия. Соответственно, принцип формирования имен подпрограмм имеет ту же самую схему, что и в LAPACK. Согласно этой схеме имена подпрограмм пакета имеют вид **PTXXYYY**, где:

T – код типа исходных данных, который может иметь следующие значения:

- **S** – вещественный одинарной точности,
- **D** – вещественный двойной точности,
- **C** – комплексный одинарной точности,
- **Z** – комплексный двойной точности;

XX – указывает вид матрицы:

- **DB** – ленточные общего вида с преобладающими диагональными элементами,
- **DT** – трехдиагональные общего вида с преобладающими диагональными элементами,

GB – ленточные общего вида,
GE – общего вида,
GT – трехдиагональные общего вида,
HE – эрмитовы,
PB – ленточные симметричные или эрмитовы положительно определенные,
PO – симметричные или эрмитовы положительно определенные,
PT – трехдиагональные симметричные или эрмитовы положительно определенные,
ST – симметричные трехдиагональные,
SY – симметричные,
TR – треугольные,
TZ – трапециевидные,
UN – унитарные;

YYY – указывает на выполняемые действия данной подпрограммой:

TRF – факторизация матриц,
TRS – решение СЛАУ после факторизации,
CON – оценка числа обусловленности матрицы (после факторизации),
SV – решение СЛАУ,
SVX – решение СЛАУ с дополнительными исследованиями,
EV и **EVX** – вычисление собственных значений и собственных векторов,
GVX – решение обобщенной задачи на собственные значения,
SVD – вычисление сингулярных значений,
RFS – уточнение решения,
LS – нахождение наименьших квадратов.

Полный список подпрограмм и их назначение можно найти в руководстве по ScaLAPACK [5]. Здесь мы рассмотрим только вопросы, касающиеся использования этой библиотеки.

14.3. Использование библиотеки ScaLAPACK

Библиотека ScaLAPACK требует, чтобы все объекты (векторы и матрицы), являющиеся параметрами подпрограмм, были предварительно распределены по процессорам. Исходные объекты классифицируются как *глобальные* объекты, и параметры, описывающие их, хранятся в специальном описателе объекта – дескрипторе. *Дескриптор* некоторого распределенного по процессорам глобального объекта представляет собой массив целого типа, в котором хранится вся необходимая информация об

исходном объекте. Части этого объекта, находящиеся в памяти какого-либо процессора, и их параметры являются *локальными* данными.

Для того, чтобы воспользоваться драйверной или вычислительной подпрограммой из библиотеки ScaLAPACK, необходимо выполнить 4 шага:

1. инициализировать сетку процессоров;
2. распределить матрицы на сетку процессоров;
3. вызвать вычислительную подпрограмму;
4. освободить сетку процессоров.

Библиотека подпрограмм ScaLAPACK поддерживает работу с матрицами трех типов:

1. заполненными (не разреженными) прямоугольными или квадратными матрицами;
2. ленточными матрицами (узкими);
3. заполненными матрицами, хранящимися на внешнем носителе.

Для каждого из этих типов используется различная топология сетки процессоров и различная схема распределения данных по процессорам. Эти различия обуславливают использование четырех типов дескрипторов. Дескриптор – это массив целого типа, состоящий из 9 или 7 элементов в зависимости от типа дескриптора. Тип дескриптора, который используется для описания объекта, хранится в первой ячейке самого дескриптора. В таблице 14.1 приведены возможные значения типов дескрипторов.

Таблица 14.1. Типы дескрипторов.

Тип дескриптора	Назначение
1	Заполненные матрицы
501	Ленточные и трехдиагональные матрицы
502	Матрица правых частей для уравнений с ленточными и трехдиагональными матрицами
601	Заполненные матрицы на внешних носителях

Инициализация сетки процессоров

Для объектов, которые описываются дескриптором типа 1, распределение по процессорам производится на двумерную сетку

процессоров. На рис. 14.2 представлен пример двумерной сетки размера 2×3 из 6 процессоров.

	0	1	2
0	0	1	2
1	3	4	5

Рис. 14.2. Пример двумерной сетки из 6 процессоров.

При таком распределении процессоры имеют двойную нумерацию: сквозную нумерацию по всему ансамблю процессоров и координатную нумерацию по строкам и столбцам сетки. Связь между сквозной и координатной нумерациями определяется параметром процедуры при инициализации сетки (нумерация вдоль строк – row-major или вдоль столбцов – column-major).

Инициализация сетки процессоров выполняется с помощью подпрограмм из библиотеки BLACS. Ниже приводится формат вызовов этих подпрограмм.

CALL BLACS_PINFO (IAM, NPROCS) –
инициализирует библиотеку BLACS, устанавливает некоторый стандартный контекст для ансамбля процессоров (аналог коммуникатора в MPI), сообщает процессору его номер в ансамбле (**IAM**) и количество доступных задаче процессоров (**NPROCS**).

CALL BLACS_GET (-1, 0, ICTXT) –
выполняет опрос установленного контекста (**ICTXT**) для использования в других подпрограммах.

CALL BLACS_GRIDINIT (ICTXT, 'Row-major', NPROW, NPCOL) –
выполняет инициализацию сетки процессоров размера **NPROW** × **NPCOL** с нумерацией вдоль строк.

CALL BLACS_GRIDINFO (ICTXT, NPROW, NPCOL, MYROW, MYCOL) –
сообщает процессору его позицию (**MYROW, MYCOL**) в сетке процессоров**.**

Для ленточных и трехдиагональных матриц (дескриптор 501) используется одномерная сетка процессоров (**1** × **NPROCS**), т.е. параметр **NPROW = 1**, а **NPCOL = NPROCS.**

Для объектов, описываемых дескриптором 502 (правые части уравнений с ленточными и трехдиагональными матрицами), используется транспонированная одномерная сетка (**NPROCS×1**). Работа с матрицами на внешних носителях (дескриптор 601) в данной книге не рассматривается.

Распределение матрицы на сетку процессоров

Способ распределения матрицы на сетку процессоров также зависит от типа распределяемого объекта. Для заполненных матриц (тип дескриптора 1) принят блочно-циклический способ распределения данных по процессорам. При таком распределении матрица разбивается на блоки размера MB × NB, где MB – размер блока по строкам, а NB – по столбцам, и эти блоки циклически раскладываются по процессорам. Проиллюстрируем это на конкретном примере распределения матрицы общего вида A(9,9), т.е. M=9, N=9, на сетке процессоров NPROW × NPCOL = 2×3 при условии, что размер блока MB × NB = 2×2.

Разбиение исходной матрицы на блоки требуемого размера будет выглядеть следующим образом (рис. 14.3):

a_{11} a_{12} a_{21} a_{22}	a_{13} a_{14} a_{23} a_{24}	a_{15} a_{16} a_{25} a_{26}	a_{17} a_{18} a_{27} a_{28}	a_{19} a_{29}
a_{31} a_{32} a_{41} a_{42}	a_{33} a_{34} a_{43} a_{44}	a_{35} a_{36} a_{45} a_{46}	a_{37} a_{38} a_{47} a_{48}	a_{39} a_{49}
a_{51} a_{52} a_{61} a_{62}	a_{53} a_{54} a_{63} a_{64}	a_{55} a_{56} a_{65} a_{66}	a_{57} a_{58} a_{67} a_{68}	a_{59} a_{69}
a_{71} a_{72} a_{81} a_{82}	a_{73} a_{74} a_{83} a_{84}	a_{75} a_{76} a_{85} a_{86}	a_{77} a_{78} a_{87} a_{88}	a_{79} a_{89}
a_{91} a_{92}	a_{93} a_{94}	a_{95} a_{96}	a_{97} a_{98}	a_{99}

Рис. 14.3. Разбиение исходной матрицы на блоки размера 2×2.

После циклического распределения блоков вдоль строк и столбцов сетки процессоров получим следующее распределение матричных элементов по процессорам (рис. 14.4):

Процессоры

	0	1	2
0	a_{11} a_{12} a_{17} a_{18} a_{21} a_{22} a_{27} a_{28} a_{51} a_{52} a_{57} a_{58} a_{61} a_{62} a_{67} a_{68} a_{91} a_{92} a_{97} a_{98}	a_{13} a_{14} a_{19} a_{23} a_{24} a_{29} a_{53} a_{54} a_{59} a_{63} a_{64} a_{69} a_{93} a_{94} a_{99}	a_{15} a_{16} a_{25} a_{26} a_{55} a_{56} a_{65} a_{66} a_{95} a_{96}
1	a_{31} a_{32} a_{37} a_{38} a_{41} a_{42} a_{47} a_{48} a_{71} a_{72} a_{77} a_{78} a_{81} a_{82} a_{87} a_{88}	a_{33} a_{34} a_{39} a_{43} a_{44} a_{49} a_{73} a_{74} a_{79} a_{83} a_{84} a_{89}	a_{35} a_{36} a_{45} a_{46} a_{75} a_{76} a_{85} a_{86}

Рис. 14.4. Распределение элементов матрицы на сетке процессоров 2×3.

Следует иметь в виду, что описанная выше процедура – это не более чем наглядная иллюстрация сути вопроса. На самом деле полная матрица A и ее разбиение на блоки (рис. 14.4) существует только в нашем воображении. В реальности задача состоит в том, чтобы в каждом процессоре сформировать массивы заведомо меньшей размерности, чем исходная матрица, и заполнить эти массивы матричными элементами в соответствии с принятыми параметрами распределения. Точное значение – сколько строк и столбцов должно находиться в каждом процессоре – позволяет вычислить подпрограмма-функция **NUMROC** из библиотеки PTOOLS, формат вызова которой приводится ниже.

NP = NUMROC (M, MB, MYROW, RSRC, NPROW)

NQ = NUMROC (N, NB, MYCOL, CSRC, NPCOL)

Здесь:

NP – число строк в локальных подматрицах в строке процессоров MYROW;

NQ – число столбцов в локальных подматрицах в столбце процессоров MYCOL.

Входные параметры:

M,	**N**	–	число строк и столбцов исходной матрицы;
MB,	**NB**	–	размеры блоков по строкам и по столбцам;
MYROW,	**MYCOL**	–	координаты процессора в сетке процессоров;
RSRC,	**CSRC**	–	координаты процессора, начиная с которого выполнено распределение матрицы

(подразумевается возможность распределения не по всем процессорам);

NPROW, NPCOL – число строк и столбцов в сетке процессоров.

Важно также уметь оценить верхние значения величин **NP** и **NQ** для правильного описания размерностей локальных массивов. Можно использовать, например, такие формулы:

NROW = (M-1)/NPROW+MB

NCOL = (N-1)/NPCOL+NB

Дескриптор для данного типа матриц – это массив целого типа из 9 элементов. Он может быть заполнен либо непосредственно с помощью операторов присваивания, либо с помощью специальной подпрограммы из библиотеки PTOOLS. Описание назначения полей дескриптора и присваиваемых им значений приводится в следующей таблице.

Таблица 14.2. Дескриптор для заполненных матриц.

DESC(1)	**= 1**	– тип дескриптора;
DESC(2)	**= ICTXT**	– контекст ансамбля процессоров;
DESC(3)	**= M**	– число строк исходной матрицы;
DESC(4)	**= N**	– число столбцов исходной матрицы;
DESC(5)	**= MB**	– размер блока по строкам;
DESC(6)	**= NB**	– размер блока по столбцам;
DESC(7)	**= RSRC**	– номер строки в сетке процессоров, начиная с которой распределяется матрица (обычно 0);
DESC(8)	**= CSRC**	– номер столбца в сетке процессоров, начиная с которого распределяется матрица (обычно 0);
DESC(9)	**= NROW**	– число строк в локальной матрице (leading dimension).

Вызов подпрограммы из PTOOLS, выполняющей аналогичное действие по формированию дескрипторов, имеет вид:

CALL DESCINIT(DESC, M, N, MB, NB, RSRC, CSRC, ICTXT, NROW,INFO)

Параметр **INFO** – статус возврата. Если **INFO = 0**, то подпрограмма выполнилась успешно; **INFO = -I** означает, что **I**-й параметр некорректен.

В расчетных формулах при решении задач линейной алгебры, как правило, фигурируют выражения, содержащие матричные элементы, идентифицируемые их глобальными индексами **(I,J)**. После распределения матрицы по процессорам, а точнее, при заполнении локальных матриц на

каждом процессоре, мы имеем дело с локальными индексами **(i,j)**, поэтому очень важно знать связь между локальными и глобальными индексами. Так, если в сетке процессоров **NPROW** строк, а размер блока вдоль строк равен **MB**, то **i**-я строка локальных подматриц в **MYROW** строке сетки процессоров должна быть заполнена **I**-ми элементами строки исходной матрицы. Формула для вычисления индекса **I** имеет вид:

I = MYROW*MB + ((i -1)/MB)*NPROW*MB + mod(i -1,MB) + 1

Аналогично можно записать формулу для индексов столбцов:

J = MYCOL*NB + ((j -1)/NB)*NPCOL*NB + mod(j - 1,NB) + 1

Здесь деление подразумевает деление нацело, а функция **mod** вычисляет остаток от деления.

Симметричные и эрмитовы матрицы на сетке процессоров хранятся в виде полных матриц, однако используется при этом либо верхний, либо нижний треугольники. Поэтому в процедурах, работающих с такими матрицами, указывается параметр **UPLO = ’U’** для верхнего треугольника или **UPLO = ‘L’** для нижнего треугольника.

Совершенно другой принцип распределения по процессорам принят для ленточных и трехдиагональных матриц. В этом случае используется блочный принцип распределения. В каждом процессоре хранится только один блок, размер которого выбирается из соображений равномерности распределения, т.е. **NB ≈ N/NPROCS**. При этом объекты левой части уравнения распределяются по столбцам, а правой – по строкам.

На рис. 14.5 представлен пример ленточной матрицы A(7,7).

$$
A = \begin{pmatrix}
a11 & a12 & a13 & 0 & 0 & 0 & 0 \\
a21 & a22 & a23 & a24 & 0 & 0 & 0 \\
a31 & a32 & a33 & a34 & a35 & 0 & 0 \\
0 & a42 & a43 & a44 & a45 & a46 & 0 \\
0 & 0 & a53 & a54 & a55 & a56 & a57 \\
0 & 0 & 0 & a64 & a65 & a66 & a67 \\
0 & 0 & 0 & 0 & a75 & a76 & a77
\end{pmatrix}
$$

Рис. 14.5. Пример ленточной матрицы размера 7×7.

Для характеристики ленточных матриц вводится параметр ширины ленты **BW** для симметричных матриц и два параметра **BWL** (ширина поддиагонали) и **BWU** (ширина наддиагонали) для несимметричных матриц. Так для рассмотренного на рис. 14.5 примера:

- если матрица A симметричная положительно определенная, то BW = 2;
- если матрица A несимметричная, то BWL = 2 и BWU = 2.

Кроме того, если матрица несимметрична, то возможны два варианта факторизации этой матрицы: без выбора главного элемента и с выбором главного элемента столбца. В обоих случаях матрицы хранятся в виде прямоугольных матриц и распределяются по столбцам, однако в первом случае количество строк матрицы равно **BWL+BWU+1**, во втором **2*(BWL+BWU) +1**.

Схема хранения матричных элементов несимметричной матрицы при использовании процедур “без выбора главных элементов” в трех процессорах при условии **NB = 3** представлена на рис. 14.6. Звездочками отмечены неиспользуемые позиции.

Процессоры

0			1			2
*	*	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Рис. 14.6. Распределение по процессорам несимметричной ленточной матрицы “без выбора главного элемента”.

При использовании процедур “с выбором главного элемента” необходимо предусмотреть выделение дополнительной памяти для рабочих ячеек. Схема распределения матричных элементов в этом случае представлена на рис. 14.7, где через F обозначена дополнительная память.

Процессоры

0			1			2
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
F	F	F	F	F	F	F
*	*	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Рис. 14.7. Распределение по процессорам несимметричной ленточной матрицы "с выбором главного элемента”.

В случае, когда матрица A является симметричной положительно определенной, достаточно хранить либо нижние поддиагонали (рис. 14.8), либо верхние (рис. 14.9).

Процессоры

0			1			2
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	*
a_{31}	a_{42}	a_{53}	a_{64}	a_{75}	*	*

Рис. 14.8. Распределение по процессорам симметричной положительно определенной ленточной матрицы (UPLO = ‘L’).

Процессоры

0			1			2
*	*	a_{13}	a_{24}	a_{35}	a_{46}	a_{57}
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}

Рис. 14.9. Распределение по процессорам симметричной положительно определенной ленточной матрицы (UPLO = ‘U’).

Трехдиагональные матрицы хранятся в виде трех векторов (DU, D, DL) в случае несимметричных матриц (рис. 14.10) и двух векторов (D, E) для симметричных матриц (рис. 14.11, 14.12).

Процессоры

	0			1			2
DL	*	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
DU	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}	*

Рис. 14.10. Пример распределения по процессорам несимметричной трехдиагональной матрицы.

Для симметричных матриц в вектор E могут заноситься либо поддиагональные элементы (рис. 14.11), либо наддиагональные (рис. 14.12).

Процессоры

	0			1			2
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
E	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	a_{76}	

Рис. 14.11. Пример распределения по процессорам симметричной трехдиагональной матрицы (UPLO = ‘L’).

Процессоры

	0			1			2
D	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	a_{77}
E	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{67}	

Рис. 14.12. Пример распределения по процессорам симметричной трехдиагональной матрицы (UPLO = ‘U’).

Для всех рассмотренных выше матриц в качестве дескриптора используется массив целого типа из 7 элементов. Его заполнение выполняется напрямую с помощью операторов присваивания в соответствии со следующей таблицей:

Таблица 14.3. Дескриптор для ленточных и трехдиагональных матриц.

DESC(1)	**= 501**	– тип дескриптора;
DESC(2)	**= ICTXT**	– контекст ансамбля процессоров;
DESC(3)	**= N**	– число столбцов исходной матрицы;
DESC(4)	**= NB**	– размер блока по столбцам;
DESC(5)	**= ICSRC**	– номер столбца в сетке процессоров, начиная с которого распределяется матрица (обычно 0);

DESC(6)	**= NROW**	– число строк в локальной матрице (leading dimension): **NROW=** **BWL + BWU + 1** для несимметричных матриц в процедурах “без выбора главного элемента”; **NROW=** **2*(BWL + BWU) + 1** для несимметричных матриц в процедурах “с выбором главного элемента”; **NROW=** **BW + 1** для симметричных положительно определенных матриц; **NROW**– не используется для трехдиагональных матриц;
DESC(7)		– не используется.

Дескриптор для правых частей (таблица 14.4) имеет такую же структуру, как и для левых. Отличие состоит в том, что разложение по процессорам производится не по столбцам, а по строкам. Кроме того, во избежание лишних пересылок данных его параметры должны соответствовать параметрам дескриптора для левых частей.

Таблица 14.4. Дескриптор для правых частей уравнений с ленточными и трехдиагональными матрицами.

DESC(1)	**= 502**	– тип дескриптора;
DESC(2)	**= ICTXT**	– контекст ансамбля процессоров;
DESC(3)	**= M**	– число строк исходной матрицы (**M = N**);
DESC(4)	**= MB**	– размер блока по строкам (**MB = NB**);
DESC(5)	**= IRSRC**	– номер столбца в сетке процессоров, начиная с которого распределяется матрица (обычно 0);
DESC(6)	**=NROW**	– число строк в локальной матрице (leading dimension) (**NROW = NB**);
DESC(7)		– не используется.

Обращения к подпрограммам библиотеки ScaLAPACK

Обращение к подпрограммам ScaLAPACK рассмотрим на примере вызова подпрограммы решения систем линейных алгебраических уравнений с матрицами общего вида. Имя подпрограммы **PDGESV** указывает, что:

1. тип матриц – double precision (второй символ **D**);
2. матрица общего вида, т.е. не имеет ни симметрии, ни других специальных свойств (3-й и 4-й символы **GE**);

3. подпрограмма выполняет решение системы линейных алгебраических уравнений A * X = B (последние символы **SV**).

Обращение к подпрограмме имеет вид:

CALL PDGESV (N, NRHS, A, IA, JA, DESCA, IPIV, B, IB, JB, DESCB, INFO)

Здесь

N – размерность исходной матрицы A (полной);
NRHS – количество правых частей в системе (сколько столбцов в матрице B);
A – на входе локальная часть распределенной матрицы A, на выходе локальная часть LU разложения;
IA, JA – индексы левого верхнего элемента подматрицы матрицы A, для которой находится решение (т.е. подразумевается возможность решать систему не для полной матрицы, а для ее части);
DESCA – дескриптор матрицы A (подробно рассмотрен выше);
IPIV – рабочий массив целого типа, который на выходе содержит информацию о перестановках в процессе факторизации. Длина массива должна быть не меньше количества строк в локальной подматрице;
B – на входе локальная часть распределенной матрицы B, на выходе локальная часть полученного решения (если решение найдено);
IB, JB – то же самое, что IA, JA для матрицы A;
DESCB – дескриптор матрицы B;
INFO – целая переменная, которая на выходе содержит информацию о том, успешно или нет завершилась подпрограмма, и причину аварийного завершения.

Примерно такой же набор параметров используется для всех драйверных и вычислительных подпрограмм. Назначение их достаточно понятно, следует только внимательно следить за тем, какие параметры относятся к исходным (глобальным) объектам, а какие имеют локальный характер (в первую очередь это касается размерностей массивов и векторов). Подробную информацию обо всех подпрограммах ScaLAPACK можно найти на WWW серверах [5].

Освобождение сетки процессоров

Программы, использующие пакет ScaLAPACK, должны заканчиваться закрытием всех BLACS-процессов. Это осуществляется с помощью вызова подпрограмм:

CALL BLACS_GRIDEXIT (ICTXT) – закрытие контекста;

CALL BLACS_EXIT (0) – закрытие всех BLACS-процессов.

Примечание: Кроме перечисленных подпрограмм в библиотеку BLACS входят подпрограммы пересылки данных, синхронизации процессов, выполнения глобальных операций по всему ансамблю процессоров или его части (строке или столбцу). Это позволяет обойтись без использования каких-либо других коммуникационных библиотек, тем не менее, допускается и совместное использование BLACS с другими коммуникационными библиотеками. Полное описание подпрограмм библиотеки BLACS включено в документацию по пакету ScaLAPACK [5].

14.4. Примеры использования пакета ScaLAPACK

В качестве первого примера использования пакета ScaLAPACK рассмотрим уже знакомую нам задачу перемножения матриц. Это позволит сопоставить получаемые результаты и производительность двух программ, созданных с использованием непосредственно среды параллельного программирования MPI (см. раздел 13.2) и с помощью пакета ScaLAPACK.

Пример 1. Перемножение матриц

Используется подпрограмма **PDGEMM** из PBLAS, которая выполняет матричную операцию $\mathbf{C = \alpha A*B + \beta C}$, где **A**, **B** и **C** – матрицы, α и β – константы. В нашем случае мы полагаем $\alpha = \mathbf{1}$, $\beta = \mathbf{0}$.

```
      program abcsl
      include 'mpif.h'
c  Параметр nm определяет максимальную размерность блока матрицы
c  на одном процессоре, массивы описаны как одномерные
```

```
      parameter (nm = 1000, nxn = nm*nm)
      double precision a(nxn), b(nxn), c(nxn), mem(nm)
      double precision time(6), ops, total, t1
c  Параметр NOUT – номер выходного устройства (терминал)
      PARAMETER  ( NOUT = 6 )
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER  ( ONE = 1.0D+0, ZERO = 0.0D+0 )
      INTEGER     DESCA(9), DESCB(9), DESCC(9)

c  Инициализация BLACS
      CALL BLACS_PINFO( IAM, NPROCS )
c  Вычисление формата сетки процессоров,
c  наиболее приближенного к квадратному
      NPROW = INT(SQRT(REAL(NPROCS)))
      NPCOL = NPROCS/NPROW
c  Считывание параметров решаемой задачи ( N – размер матриц и
c  NB – размер блоков ) 0-м процессором и печать этой информации
      IF ( IAM.EQ.0 ) THEN
      WRITE(*,* ) ' Input N and NB: '
      READ( *, * ) N, NB
      WRITE( NOUT, FMT = * )
      WRITE( NOUT, FMT = 9999 )
     $          'The following parameter values will be used:'
      WRITE( NOUT, FMT = 9998 ) 'N    ', N
      WRITE( NOUT, FMT = 9998 ) 'NB   ', NB
      WRITE( NOUT, FMT = 9998 ) 'P    ', NPROW
      WRITE( NOUT, FMT = 9998 ) 'Q    ', NPCOL
      WRITE( NOUT, FMT = * )
      END IF

c  Рассылка считанной информации всем процессорам
      call MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      call MPI_BCAST(NB, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
c  Теоретическое количество операций при перемножении
c  двух квадратных матриц
      ops = (2.0d0*dfloat(n)-1)*dfloat(n)*dfloat(n)

c  Инициализация сетки процессоров
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL)
      CALL BLACS_GRIDINFO(ICTXT, NPROW, NPCOL, MYROW,MYCOL)

c  Если процессор не вошел в сетку, то он ничего не делает;
c  такое может случиться, если заказано, например, 5 процессоров
      IF ( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )
     $    GO TO 500
c
c  Вычисление реальных размеров матриц на процессоре
      NP    = NUMROC( N, NB, MYROW, 0, NPROW )
      NQ    = NUMROC( N, NB, MYCOL, 0, NPCOL )
```

```
c  Инициализация дескрипторов для 3-х матриц
      CALL DESCINIT( DESCA,N,N,NB,NB,0,0, ICTXT, MAX(1,NP), INFO )
      CALL DESCINIT( DESCB,N,N,NB,NB,0,0, ICTXT, MAX(1,NP), INFO )
      CALL DESCINIT( DESCC,N,N,NB,NB,0,0, ICTXT, MAX(1,NP), INFO )
*
      lda = DESCA(9)
c  Вызов процедуры генерации матриц A и B
      call pmatgen(a, DESCA, np, nq, b, DESCB, nprow, npcol, myrow, mycol)

      t1 = MPI_Wtime()
*
*  Вызов процедуры перемножения матриц
      CALL PDGEMM('N', 'N', N, N, N, ONE, A, 1, 1, DESCA, B,1,1, DESCB,
     $                        ZERO, C, 1, 1, DESCC)
*
      time(2) = MPI_Wtime( ) - t1
c  Печать угловых элементов матрицы C
c  с помощью служебной подпрограммы
      if (IAM.EQ.0) write(*,*) 'Matrix C...'
      CALL PDLAPRNT( 1, 1, C, 1, 1, DESCC, 0, 0, 'C', 6, MEM )
      CALL PDLAPRNT( 1, 1, C, 1, N, DESCC, 0, 0, 'C', 6, MEM )
      CALL PDLAPRNT( 1, 1, C, N, 1, DESCC, 0, 0, 'C', 6, MEM )
      CALL PDLAPRNT( 1, 1, C, N, N, DESCC, 0, 0, 'C', 6, MEM )
c  Вычисление времени, затраченного на перемножение,
c  и оценка производительности в Mflops
      total = time(2)
      time(4) = ops/(1.0d6*total)
      if (IAM.EQ.0) then
      write(6,80) lda
 80 format(' times for array with leading dimension of', i4)
      write(6,110) time(2), time(4)
110 format(2x,'Time calculation: ', f12.4, ' sec.',
     $        ' Mflops = ', f12.4)
      end if
c  Закрытие BLACS-процессов
      CALL BLACS_GRIDEXIT( ICTXT )
      CALL BLACS_EXIT(0)
9998 FORMAT( 2X, A5, '   :        ', I6 )
9999 FORMAT(2X, 60A )
500 continue
      stop
      end

      subroutine pmatgen(a, DESCA, np, nq, b, DESCB, nprow, npcol,
     $                          myrow, mycol)
      integer n, i, j, DESCA(*), DESCB(*), nprow, npcol, myrow, mycol
      double precision a(*), b(*)
      nb = DESCA(5)
c  Заполнение локальных частей матриц A и B,
c  матрица A формируется по алгоритму A(I,J) = I, a
c  матрица B(I,J) = 1./J
c  Здесь имеются в виду глобальные индексы.
```

```
      k = 0
      do 250 j = 1,nq
      jc = (j-1)/nb
      jb = mod(j-1,nb)
      jg = mycol*nb + jc*npcol*nb + jb + 1
      do 240 i = 1,np
      ic = (i-1)/nb
      ib = mod(i-1,nb)
      ig = myrow*nb + ic*nprow*nb + ib + 1
      k = k + 1
      a(k) = dfloat(ig)
      b(k) = 1.D+0/dfloat(jg)
240 continue
250 continue
      return
      end
```

Пример 2. Решение системы линейных уравнений с матрицей общего вида

В данном примере решается система линейных уравнений с матрицей общего вида, которая формируется с помощью генератора случайных чисел. Правая часть системы формируется таким образом, чтобы получить единичное решение. Для решения системы используются две вычислительные подпрограммы: **PDGETRF** (для факторизации матрицы) и **PDGETRS** (для нахождения решения). Общий шаблон вызовов функций мало отличается от предыдущего примера. Отличие, главным образом, состоит в том, что в этом примере все коммуникационные операции выполняются с помощью подпрограмм из библиотеки BLACS (чисто из иллюстративных соображений), хотя многие операции можно компактнее записать на MPI.

```
      program pdlusl
      include 'mpif.h'
      parameter (nsize = 3000, nxn = nsize*nsize)
      double precision a(nxn), b(nsize), x(nsize)
      double precision time(6), cray, ops, total, norma, normx, t1
      double precision resid, residn, eps, epslon, rab, RANN
      integer ipvt(nsize), iwork(5), init(4)
      PARAMETER  ( NOUT = 6 )
      DOUBLE PRECISION   ONE
      PARAMETER   ( ONE = 1.0D+0 )
      INTEGER     DESCA( 9 ), DESCB( 9 ), DESCX( 9 )
```

```
c  Параметр NRHS – количество правых частей
      NRHS = 1
      CALL BLACS_PINFO( IAM, NPROCS )
      NPROW = INT(SQRT(REAL(NPROCS)))
      NPCOL  = NPROCS/NPROW
c  Теоретическое число операций, которое необходимо выполнить для
c  решения системы
      ops = (2.0d0*dfloat(n)**3)/3.0d0 + 2.0d0*dfloat(n)**2

c  Формирование одномерной сетки процессоров
c  (только для рассылки параметров)
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', 1, NPROCS )
c  На 0-м процессоре считываем параметры, упаковываем в массив
c  и рассылаем всем остальным с помощью процедуры IGEBS2D
      IF ( IAM.EQ.0 ) THEN
      WRITE(*,* ) ' Input N and NB: '
      READ( *, * ) N, NB
      IWORK( 1 ) = N
      IWORK( 2 ) = NB

      CALL IGEBS2D( ICTXT, 'All', ' ', 2, 1, IWORK, 2 )
      WRITE( NOUT, FMT = 9999 )
     $          'The following parameter values will be used:'
      WRITE( NOUT, FMT = 9998 ) 'N     ', N
      WRITE( NOUT, FMT = 9998 ) 'NB    ', NB
      WRITE( NOUT, FMT = 9998 ) 'P     ', NPROW
      WRITE( NOUT, FMT = 9998 ) 'Q     ', NPCOL
      WRITE( NOUT, FMT = * )
*
      ELSE
c  На не 0-м процессоре получаем массив с процессора (0,0) и
c  распаковываем его
      CALL IGEBR2D( ICTXT, 'All', ' ', 2, 1, IWORK, 2, 0, 0 )
      N     = IWORK( 1 )
      NB    = IWORK( 2 )
      END IF

c  Уничтожаем временную сетку процессоров
      CALL BLACS_GRIDEXIT( ICTXT )
c  Формируем рабочую сетку процессоров
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL,MYROW,MYCOL)
c
c  Проверка процессоров, не вошедших в сетку
      IF ( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )
     $    GO TO 500
c  Определяем точное число строк и столбцов распределенной матрицы
c  в процессоре
      NP   = NUMROC( N, NB, MYROW, 0, NPROW )
      NQ   = NUMROC( N, NB, MYCOL, 0, NPCOL )
```

```
c  Формируем дескрипторы
      CALL DESCINIT( DESCA,N,N,NB,NB,0,0,ICTXT, MAX(1,NP), INFO )
      CALL DESCINIT( DESCB,N,NRHS,NB,NB,0,0, ICTXT, MAX(1,NP),
     $                     INFO )
      CALL DESCINIT( DESCX,N,NRHS,NB,NB,0,0, ICTXT, MAX(1,NP),
     $                     INFO )
      lda = DESCA(9)
c  Вызов процедуры генерации матрицы А и вектора В
      call pmatgenl(a, DESCA, NP, NQ, b, DESCB, nprow,npcol,myrow,mycol)

      t1 = MPI_Wtime()
*  Обращение к процедуре факторизации матрицы А
      CALL PDGETRF( N, N, A, 1, 1, DESCA, ipvt, INFO )
*
      time(1) = MPI_Wtime() - t1
      t1 = MPI_Wtime()
c  Обращение к процедуре решения системы уравнений
c  с факторизованной матрицей
      CALL PDGETRS('No',N, NRHS,A,1,1,DESCA, ipvt,B,1,1,DESCB, INFO)
*
      time(2) = MPI_Wtime() - t1
      total = time(1) + time(2)
c  На этом собственно решение задачи заканчивается, далее идет
c  подготовка печати и сама печать
      if (iam.eq.0) then
      write(6,40)
40    format('         x(1)          x(np)')
      write(6,50) x(1), x(np)
50    format(1p5e16.8)
      write(6,60) n
60    format(//'    times are reported for matrices of order ',i5)
      write(6,70)
70    format(6x,'factor',5x,'solve',6x,'total',5x,'mflops',7x,'unit', 6x,'ratio')
c
      time(3) = total
      time(4) = ops/(1.0d6*total)
      time(5) = 2.0d0/time(4)
      time(6) = total/cray
      write(6,80) lda
80    format(' times for array with leading dimension of', i4)
      write(6,110) (time(i), i=1,6)
110 format(6(1pe11.3))
      write(6,*)' end of test'
      end if
c
      CALL BLACS_GRIDEXIT( ICTXT )
      CALL BLACS_EXIT(0)
9998 FORMAT( 2X, A5, '   :         ', I6 )
9999 FORMAT( 2X, 60A )
500  continue
      stop
      end
```

```
c  Процедура генерации матрицы A с помощью генератора случайных
c  чисел RANN.
c  Последовательности случайных чисел на процессорах должны быть
c  независимые, чтобы матрица не оказалась вырожденной.
      subroutine pmatgenl(a, DESCA, NP, NQ, b, DESCB, nprow, npcol,
     $                        myrow, mycol)
      integer n,init(4),i,j,DESCA(*),DESCB(*),nprow,npcol,myrow,mycol
      double precision a(*),b(*),rann
c
      nb = DESCA(5)
      ICTXT = DESCA(2)
c  Инициализация генератора случайных чисел
      init(1) = 1
      init(2) = myrow + 2
      init(3) = mycol + 3
      init(4) = 1325
c  Заполнение матрицы A
      k = 0
      do 250 j = 1,nq
      do 240 i = 1,np
      k = k + 1
      a(k) = rann(init) - 0.5
240 continue
250 continue
      na = k
c  Вычисление вектора B такого, чтобы получить единичное решение;
c  сначала вычисляются локальные суммы по строке на каждом
c  процессоре, а затем выполняется суммирование по всем процессорам.
      do 350 i = 1,np
      k = i
      b(i) = 0
      do 351 j = 1,nq
      b(i) = b(i) + a(k)
      k = k + np
351 continue
      CALL BLACS_BARRIER( ICTXT, 'All' )
      CALL DGSUM2D( ICTXT, 'Row', ' ', 1, 1, b(i), 1, -1, 0)
350 continue
      return
      end
```

Пример 3. Решение системы линейных алгебраических уравнений с ленточной матрицей

В данном примере решается система линейных алгебраических уравнений с симметричной положительно определенной ленточной матрицей. Используются подпрограммы **PDPBTRF** и **PDPBTRS** библиотеки ScaLAPACK. Матрица A формируется следующим образом:

```
 6  -4   1   0   0   0 ...
-4   6  -4   1   0   0 ...
 1  -4   6  -4   1   0 ...
 0   1  -4   6  -4   1 ...
 0   0   1  -4   6  -4 ...
 0   0   0   1  -4   6 ...
```

с хранением верхнего треугольника (параметр UPLO =‘U’), где звездочкой помечены неиспользуемые позиции

*	*	a_{13}	a_{24}	a_{35}	a_{46}	...	$a_{n-2,n}$
*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	...	$a_{n-1,n}$
a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}	...	$a_{n,n}$

```
      program bandu
c  nsize максимальное число столбцов матрицы в одном процессоре
      parameter (nsize = 30000)
      double precision a(3*nsize), b(nsize), x(nsize), bg(nsize)
      double precision AF(3*nsize), WORK(10)
      integer ipvt(nsize), BW
      PARAMETER  ( NOUT=6 )
      INTEGER   DESCA(7), DESCB(7), DESCX(7)

      CALL BLACS_PINFO( IAM, NPROCS )
c  Задаем размеры матрицы и сетки процессоров
      N = 9000
      NRHS = 1
      NPROW = 1
      NPCOL = 4
c  BW – ширина ленты над диагональю
      BW=2

c  Вычисляем NB – длину блока матрицы A
      NDD = mod(N, NPCOL)
      IF (NDD.EQ.0) THEN
      NB = N/NPCOL
      ELSE
      NB = N/NPCOL + 1
      END IF
      NB = MAX(NB,2*BW)
      NB = MIN(N,NB)

      IF (IAM.EQ.0) THEN
      WRITE( 6, FMT = 9998 ) 'N    ', N
      WRITE( 6, FMT = 9998 ) 'NRHS ', NRHS
      WRITE( 6, FMT = 9998 ) 'BW   ', BW
      WRITE( 6, FMT = 9998 ) 'P    ', NPROW
      WRITE( 6, FMT = 9998 ) 'Q    ', NPCOL
      WRITE( 6, FMT = 9998 ) 'NB   ', NB
      END IF
```

```
c  Вычисляем размеры рабочих массивов
      LWORK = 2*BW*BW
      LAF = (NB+2*BW)*BW
c  Инициализируем сетку процессоров
      CALL BLACS_GET( -1, 0, ICTXT )
      CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL,MYROW,MYCOL)
      IF ( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )
     $    GO TO 500
*
      NP    = NUMROC( (BW+1), (BW+1), MYROW, 0, NPROW )
      NQ    = NUMROC( N, NB, MYCOL, 0, NPCOL )
c  Формируем дескрипторы для левых и правых частей уравнения
      DESCA(1) = 501
      DESCA(2) = ICTXT
      DESCA(3) = N
      DESCA(4) = NB
      DESCA(5) = 0
      DESCA(6) = BW+1
      DESCA(7) = 0
      DESCB(1) = 502
      DESCB(2) = ICTXT
      DESCB(3) = N
      DESCB(4) = NB
      DESCB(5) = 0
      DESCB(6) = NB
      DESCB(7) = 0

      lda = NB
c  Вызов подпрограммы генерации ленточной матрицы и правой части
      call pmatgenb(a,DESCA,bw,b,DESCB,nprow,npcol,myrow,mycol,n,bg)

c  Факторизация матрицы
      CALL PDPBTRF('U',N,BW,A,1,DESCA,AF,LAF,WORK,LWORK,INFO3)

c  Решение системы
      CALL PDPBTRS('U',N,BW,NRHS,A,1,DESCA,B,1,DESCB,AF,LAF,
     $                      WORK, LWORK, INFO)
      if (iam.eq.0) then
      write(6,40)
40   format('x(1),...,x(4)')
      write(6,50)   (b(i),i=1,4)
50   format(4d16.8)
      end if
      CALL BLACS_GRIDEXIT( ICTXT )
      CALL BLACS_GRIDEXIT( ICTXTB )
      CALL BLACS_EXIT (0)
500 continue
      stop
      end

c  Подпрограмма-функция генерации (I,J) матричного элемента
      double precision function matij(i,j)
      double precision rab
```

```
      rab = 0.0d0
      if (i.eq.j)                   rab =  6.0d0
      if (i.eq.j+1.or.j.eq.i+1) rab = -4.0d0
      if (i.eq.j+2.or.j.eq.i+2) rab =  1.0d0
      matij = rab
      return
      end

c  Подпрограмма генерации матрицы A и вектора B
      subroutine pmatgenb(a,DESCA,bw, b,DESCB, nprow, npcol, myrow,
     $                         mycol,n,bg)
      integer i, j, DESCA(*), DESCB(*), nprow, npcol, bw, bw1, myrow, mycol
      double precision a(bw+1,*), b(*), bg(*), matij

      nb = DESCA(4)
      ICTXT = DESCA(2)
      n = DESCA(3)
      BW1 = BW + 1
c  Генерация всех компонент вектора B таким образом,
c  чтобы решение X(I) = I
      do 231 i = 1,n
      bg(i) = 0.0
      n1 = max(1,i-bw)
      n2 = min(n,i+bw)
      do 231 j = n1,n2
      bg(i) = bg(i) + matij(i,j)*j
231 continue

c   Вычисление локальной части матрицы A
      jcs = MYCOL*NB
      NC = MIN(NB,N-jcs)
      do 250 j = 1,NC
      jc = jcs + j
      do 240 i = 1,BW1
      ic = jc - BW1 + i
      if (ic.ge.1)  a(i,j) = matij(ic,jc)
240 continue
250 continue
c  Заполнение локальной части вектора B
      do 350 i = 1,NC
      b(i) = bg(jcs+i)
351 continue
350 continue
      return
      end
```

Глава 15.
Использование библиотеки параллельных подпрограмм Aztec

15.1. Общая организация библиотеки Aztec

Множество задач, решаемых на многопроцессорных системах, требуют решения больших систем линейных уравнений с разреженными матрицами

A*X = B, (15.1)

где A – задаваемая пользователем разреженная матрица $n \times n$,
B – задаваемый пользователем вектор длины n,
X – вектор длины n, который должен быть вычислен.

В исследовательской лаборатории параллельных вычислений Сандии (США) разработан достаточно эффективный и удобный в использовании пакет подпрограмм Aztec [18] для решения итерационными методами системы уравнений (15.1). Пакет изначально разрабатывался для того, чтобы облегчить перенос приложений с однопроцессорных вычислительных систем на многопроцессорные. Предоставляемые средства трансформации данных позволяют легко создавать разреженные неструктурированные матрицы для решения как на однопроцессорных, так и на многопроцессорных системах. В суперкомпьютерном центре ЮГИНФО РГУ эта библиотека установлена на всех высокопроизводительных вычислительных системах (nCUBE2, SUN Ultra 60, Alpha DS20E, Linux-кластер), и программы, разработанные с использованием этой библиотеки, без какой-либо модификации выполняются на любой из этих систем.

Aztec включает в себя процедуры, реализующие ряд итерационных методов Крылова:

- метод сопряженных градиентов (**CG**),
- обобщенный метод минимальных невязок (**GMRES**),
- квадратичный метод сопряженных градиентов (**CGS**),
- метод квазиминимальных невязок (**TFQMR**),
- метод бисопряженного градиента (**BiCGSTAB**) со стабилизацией.

Все методы используются совместно с различными переобуславливателями (полиномиальный метод и метод декомпозиции областей, использующий как прямой метод LU, так и неполное LU разложение в подобластях). Хотя матрица A может быть общего вида, пакет ориентирован на матрицы, возникающие при конечно-разностной аппроксимации дифференциальных уравнений в частных производных (Partial Differential Equations – PDE). Наконец, Aztec может использовать одно из двух представлений разреженных матриц: поэлементный формат модифицированной разреженной строки (**MSR**) или блочный формат переменной блочной строки (**VBR**). Полный комплект документации можно найти на сервере [19]. В данном пособии приводится краткая инструкция по использованию пакета.

15.2. Конфигурационные параметры библиотеки Aztec

Для использования библиотеки Aztec в текст программы должен быть включен include-файл, в котором определен набор необходимых переменных:

```
C:
   #include <az_aztec.h>
FORTRAN:
   include ‘az_aztecf.h’
```

Пакет написан на языке C, поэтому все используемые при вызове подпрограмм массивы должны быть описаны индексируемыми от 0 (в том числе и на Фортране). Вызов различных подпрограмм решения систем линейных алгебраических уравнений выполняется через драйверную подпрограмму **AZ_solve.** Управление режимами работы решателя осуществляется с помощью двух массивов:

- массива целого типа *options*(**0:AZ_OPTIONS_SIZE**);
- массива вещественного двойной точности *params*(**0:AZ_PARAMS_SIZE**).

Возвращаемая информация помещается в массив двойной точности *status*(**0:AZ_STATUS_SIZE**).

Приведем список наиболее важных опций. Значения констант, используемых в качестве индексов элементов массивов *options, params, status* и в качестве значений, присваиваемых этим элементам, определены в include-файлах.

*options[***AZ_solver***]* Специфицирует алгоритм решения. Возможные значения задаются либо в виде именованных констант, либо числовых значений, которые приведены в скобках:

AZ_cg (0) Метод сопряженных градиентов (применяется только к симметричным, положительно определенным матрицам).
AZ_gmres (1) Обобщенный метод минимальных невязок.
AZ_cgs (2) Квадратичный метод сопряженных градиентов.
AZ_tfqmr (3) Метод квазиминимальных невязок.
AZ_bicgstab (4) Метод бисопряженного градиента со стабилизацией.
AZ_lu (5) Прямой LU метод решения (только на одном процессоре).

По умолчанию: **AZ_gmres**

*options[***AZ_scaling***]* Определяет алгоритм масштабирования. Масштабируется вся матрица (запись идет на место старой). Кроме того, при необходимости масштабируются правая часть, начальные данные и полученное решение. Возможные значения:

AZ_none Без масштабирования.
AZ_Jacobi Точечное масштабирование Якоби.
AZ_BJacobi Блочное масштабирование Якоби, где размер блоков соответствует **VBR** блокам. Точечное масштабирование Якоби применяют, когда используется **MSR** формат.
AZ_row_sum Масштабирование каждой строки такое, что сумма ее элементов равна 1.
AZ_sym_diag Симметричное масштабирование такое, что диагональные элементы равны 1.
AZ_sym_row_sum Симметричное масштабирование с использованием суммы элементов строки.

По умолчанию: **AZ_none.**

*options[***AZ_precond***]* Специфицирует переобуславливатель.

Возможные значения:

AZ_none Переобуславливания нет.

AZ_Jacobi *k* шаговый метод Якоби. Число *k* шагов Якоби задается через *options[***AZ_poly_ord***]*.

AZ_Neumann Ряд полиномов Неймана, где степень полинома задается через *options[***AZ_poly_ord***]*.

AZ_ls Полином наименьших квадратов, где степень полинома задается через *options[***AZ_poly_ord***]*.

AZ_lu Метод декомпозиции областей (аддитивная процедура Шварца), использующий неполную LU факторизацию с величиной допустимого отклонения *params[***AZ_drop***]* на подматрице каждого процессора. Обработка внешних переменных в подматрице определяется с помощью *options[***AZ_overlap***]*.

AZ_ilu Подобно **AZ_lu**, но используется **ilu(0)** вместо **LU**.

AZ_bilu Подобно **AZ_lu**, но используется **bilu(0)** вместо **LU**, где каждый блок соответствует **VBR** блоку.

AZ_sym_GS *k*-шаговый симметричный метод Гаусса-Зейделя для неперекрывающейся декомпозиции области (аддитивная процедура Шварца). Симметричная процедура Гаусса-Зейделя декомпозиции области используется тогда, когда каждый процессор независимо выполняет один шаг симметричного метода Гаусса-Зейделя на его локальной подматрице, после чего с помощью межпроцессорных коммуникаций обновляются граничные значения, требуемые на следующем локальном шаге метода Гаусса-Зейделя. Число шагов *k* задается через *options[***AZ_poly_ord***]*.

По умолчанию: **AZ_none.**

*options[***AZ_conv***]* Определяет выражение невязки, используемое для подтверждения сходимости. Итерационное решение завершается, если соответствующее выражение невязки меньше, чем *params[***AZ_tol***]*.

Возможные значения:

AZ_r0 $\|r\|_2 / \|r^{(0)}\|_2$

AZ_rhs $\|r\|_2 / \|b\|_2$

AZ_Anorm $\|r\|_2 / \|A\|_\infty$

AZ_sol $\|r\|_\infty / (\|A\|_\infty * \|x\|_1 + \|b\|_\infty)$

AZ_weighted $\|r\|_{WRMS}$, где $\|\cdot\|_{WRMS} = \sqrt{(1/n)\sum_{i=1}^{n}(r_i/w_i)^2}$, n – общее число неизвестных, w – вес вектора, задаваемый пользователем через *params[***AZ_weights***]*, и $r^{(0)}$ – начальная невязка.

По умолчанию: **AZ_r0**.

*options[***AZ_output***]* Специфицирует информацию, которая должна быть выведена на печать в процессе решения (выражения невязки см. *options[***AZ_conv***]*).

Возможные значения:

AZ_all Выводит на печать матрицу и индексирующие векторы для каждого процессора. Печатает все промежуточные значения невязок.

AZ_none Промежуточные результаты не печатаются.

AZ_warnings Печатаются только предупреждения Aztec'a.

AZ_last Печатается только окончательное значение невязки.

n (n >0) Печатается значение невязки на каждой n-ой итерации.

По умолчанию: 1.

*options[***AZ_pre_calc***]* Показывает, использовать ли информацию о факторизации из предыдущего вызова **AZ_solve.**

Возможные значения:

AZ_calc Не использовать информацию из предыдущих вызовов **AZ_solve.**

AZ_recalc Использовать информацию предварительной обработки из предыдущего вызова, но повторно вычислять переобуславливающие множители.

AZ_reuse Использовать переобуславливатель из предыдущего вызова **AZ_solve**. Кроме того, можно использовать масштабирующие множители из предыдущего обращения для масштабирования правой части, начальных данных и окончательного решения.

По умолчанию: **AZ_calc.**

*options[***AZ_max_iter***]* Максимальное число итераций.

По умолчанию: 500.

*options[***AZ_poly_ord***]* Степень полинома при использовании полиномиального переобуславливателя. А также число шагов при переобуславливании Якоби или симметричном переобуславливании Гаусса-Зейделя.

По умолчанию: 3.

*options[***AZ_overlap***]* Определяет подматрицы, факторизуемые алгоритмами декомпозиции области **AZ_lu, AZ_ilu, AZ_bilu**.

Возможные значения:

AZ_none Факторизовать определенную на этом процессоре локальную подматрицу, отбрасывая столбцы, которые соответствуют внешним элементам.

AZ_diag Факторизовать локальную подматрицу на этом процессоре, расширенную диагональной (блочно-диагональной для **VBR** формата) матрицей. Эта диагональная матрица состоит из диагональных элементов строк матрицы, связанных с внешними элементами.

AZ_full Факторизовать определенную на этом процессоре локальную подматрицу, расширенную строками, связанными с внешними переменными. Результирующая процедура – аддитивная процедура Шварца с совмещением.

По умолчанию: **AZ_none**.

*options[***AZ_kspace***]* Размер подпространства Крылова для **GMRES**.

По умолчанию: 30.

*options[***AZ_orthog***]* Схема ортогонализации **GMRES**.

Возможные значения:

AZ_classic Классическая ортогонализация Грамм-Шмидта.

AZ_modified Модифицированная ортогонализация Грамм-Шмидта.

По умолчанию: **AZ_classic**.

Массив *params* двойной точности служит для передачи в процедуры значений параметров вещественного типа. Элементы этого массива имеют следующие значения:

*params[***AZ_tol***]* Определяет критерий сходимости.
По умолчанию: 10^{-6}

*params[***AZ_drop***]* Определяет величину допустимого отклонения для LU переобуславливателей. По умолчанию: 0.0.

*params[***AZ_weights***]* Когда *options[***AZ_conv***]* = **AZ_weighted**, **i**-ая локальная компонента вектора веса хранится в элементе *params[***AZ_weights+i***]*.

Вся целочисленная информация, возвращаемая из **AZ_solve**, переводится в вещественные значения двойной точности и помещается в массив *status*. Содержание элементов *status* описано ниже.

*status[***AZ_its***]* Число итераций, выполненных итерационным методом.

*status[***AZ_why***]* Причина завершения программы **AZ_solve**.

Возможные значения

AZ_normal	Критерий сходимости, который запросил пользователь, удовлетворен.
AZ_param	Некорректно заданная опция.
AZ_breakdown	Произошло численное прерывание.
AZ_loss	Произошла потеря точности.
AZ_ill_cond	Метод Хейссенберга в **GMRES** некорректен. Это может быть вызвано сингулярностью матрицы. В этом случае **GMRES** пробует найти решение наименьшими квадратами.
AZ_maxits	Сходимость не достигнута после выполнения максимального число итераций, заданного соответствующим параметром

*status[***AZ_r***]* Истинная норма невязки, соответствующая выбору *options[***AZ_conv***]* (для вычисления нормы используется полученное решение).

*status[***AZ_scaled_r***]* Истинное выражение отношения невязки как определено в *options[***AZ_conv***]*.

*status[***AZ_rec_r***]* Норма соответствующая *options[***AZ_conv***]* конечной невязки или оценочной конечной невязки (рекурсивно вычисленной итерационным методом).

15.3. Основные подпрограммы библиотеки Aztec

В состав библиотеки Aztec входит около 50 подпрограмм (функций в языке С), однако большая их часть имеет вспомогательный характер и не требуется прямое обращение к ним. Стандартное использование

библиотеки требует обращения к 5-6 подпрограммам. Рассмотрим их в соответствии с очередностью вызовов этих подпрограмм в программах, основанных на библиотеке Aztec.

Функция идентификации процессоров AZ_processor_info

Эта функция является аналогом функции whoami в PSE и функций MPI_Comm_rank, MPI_Comm_size в MPI.

call AZ_processor_info(proc_config)

Информация возвращается в массив

integer proc_config(0:AZ_PROC_SIZE)

При этом номер текущего процессора **IAM** и количество процессоров **NPROCS** заносятся в соответствующие элементы этого массива:

IAM	**=**	**proc_config(AZ_node)**	–	номер вызывающего процессора;
NPROCS	**=**	**proc_config (AZ_N_procs)**	–	общее число процессоров, доступных программе.

Функция рассылки информации по процессорам AZ_broadcast

Это аналог функции MPI_Bcast, ее реализация выполнена в стиле PVM: рассылаемая информация сначала упаковывается в системный буфер, а затем посылается одной командой. Рассылать может только 0-й процессор. Вызов функции имеет следующий вид:

call AZ_broadcast(n, 4, proc_config, AZ_PACK)	–	упаковываем в буфер целую переменную **n**
call AZ_broadcast(d, 8, proc_config, AZ_PACK)	–	упаковываем в буфер переменную **d** типа real*8
call AZ_broadcast(NULL, 0, proc_config, AZ_SEND)	–	посылаем буфер

Функция определения параметров декомпозиции матрицы AZ_read_update

Эта функция определяет, сколько строк глобальной матрицы должно находиться в данном процессоре и номера этих строк. Вызов функции имеет следующий вид:

call AZ_read_update(N_update, update, proc_config, nz, 1, 0)

N_update	–	целая переменная, возвращает количество строк на процессоре;
update	–	массив целого типа размерности не меньше, чем **N_update+1**, после вызова содержит номера строк в глобальной нумерации;
nz	–	размерность глобальной матрицы (входной параметр).

Два последних параметра вызова процедуры – константы.

После вызова всех рассмотренных выше функций в каждом процессоре должна быть сформирована размещенная в нем часть разреженной матрицы, представленная в виде двух массивов:

- массива двойной точности val, в котором хранятся ненулевые матричные элементы,
- целого массива bindx, в котором хранятся номера столбцов ненулевых матричных элементов (в случае хранения в формате **MSR**).

Этот этап целиком и полностью возлагается на программиста и требует от него некоторой изобретательности. Для **VBR** формата требуется сохранять больше информации (в данном пособии мы его рассматривать не будем). Кроме того, должны быть сформированы векторы правых частей b и начальное приближение решения x (только расположенные в данном процессоре части).

Функция преобразования глобальной индексации в локальную AZ_transform

```
call AZ_transform(proc_config, external, bindx, val, update,
$                 update_index,  extern_index, data_org, N_update,
$                 NULL, NULL, NULL, NULL, AZ_MSR_MATRIX)
```

Входные параметры этой процедуры proc_config, bindx, val, update, N_update рассмотрены выше. Выходные параметры – целые массивы external, update_index, extern_index, data_org размерности не меньше, чем N_update + 1 –будут передаваться процедуре решения

системы уравнений **AZ_solve**. Параметры, имеющие значения NULL, в случае **MSR** формата не используются.

Перед обращением к процедуре решения системы вектор правых частей и начальное приближение должны быть переопределены в соответствии с новой индексацией.

```
do  i = 0, N_update-1
xn(update_index(i)) = x(i)
bn(update_index(i)) = b(i)
end do
```

Функция инициализации предопределенных параметров решателя AZ_defaults

call AZ_defaults(options, params)

В данной процедуре оба параметра выходные и были подробно рассмотрены выше. Установки по умолчанию могут быть переопределены:

options(AZ_solver) = AZ_cgs	– выбор метода решения CGS
options(AZ_precond) = AZ_dom_decomp	– выбор переобуславливателя
options(AZ_subdomain_solve) = AZ_ilu	– неполное LU разложение
options(AZ_output) = AZ_none	– диагностику не выдавать
options(AZ_max_iter) = 1000	– максимальное число итераций 1000
params(AZ_tol) = 1.d-8	– критерий сходимости 10^{-8}

Функция решения системы линейных алгебраических уравнений AZ_solve

Эта функция является основной "рабочей лошадкой" пакета. Ее вызов имеет следующий вид:

```
call AZ_solve(xn, bn, options, params, NULL, bindx, NULL,
$              NULL, NULL, val, data_org,  status,  proc_config)
```

Параметры NULL не используются в случае **MSR** формата хранения матрицы. Выходными параметрами являются xn – распределенное по процессорам решение системы, и рассмотренный выше массив *status*.

15.4. Хранение разреженных матриц в MSR формате

При использовании пакета Aztec самой сложной для программиста задачей является формирование надлежащим образом распределенной по процессорам матрицы.

На рис. 12 представлен пример разреженной матрицы размера 6x6. Предположим, что мы хотим распределить ее в 3 процессора (нумерация процессоров с 0). Если бы мы воспользовались процедурой AZ_read_update, то, скорее всего, для каждого процессора N_update было бы установлено равным 2, и в массив update были бы соответственно занесены значения (0,1),(2,3),(4,5). Однако, для общности, рассмотрим другой вариант распределения. Пусть 0-й процессор хранит нулевую, первую и третью строки, 1-й процессор – одну четвертую строку, 2-й процессор – вторую и пятую строки. Этим самым мы однозначно задаем значения переменной N_update и массива update. Наша задача состоит в том, чтобы заполнить массивы bindx и val так, чтобы они адекватно описывали принятое распределение.

Массив val формируется следующим образом. Сначала в этот массив заносятся диагональные элементы хранимых строк, одна позиция пропускается (это сделано для соответствия с bindx) и затем, строка за строкой, в массив заносятся недиагональные ненулевые элементы (в том порядке, в котором они перечислены в массиве update).

В начале массива bindx хранятся номера позиций, с которых в массиве val начинаются недиагональные элементы соответствующей строки (включая и первую несуществующую – для того, чтобы можно было определить, сколько элементов в последней строке). Далее в массиве следуют номера столбцов, соответствующих ненулевым матричным элементам. Значения, которые должны быть занесены в соответствующие массивы в каждом процессоре, приведены ниже. Напоминаем, что индексация массивов начинается с 0.

$$A = \begin{pmatrix} a_{00} & a_{01} & & a_{03} & a_{04} & \\ a_{10} & a_{11} & & a_{13} & & \\ & & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & & a_{42} & a_{43} & a_{44} & \\ & & a_{52} & a_{53} & & a_{55} \end{pmatrix}$$

Рис. 12. Разреженная исходная матрица

proc 0:

N_update:	3													
update:	0	1	3											
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
bindx**:**	4	7	9	14	1	3	4	0	3	0	1	2	4	5
val:	a_{00}	a_{11}	a_{33}	–	a_{01}	a_{03}	a_{04}	a_{10}	a_{13}	a_{30}	a_{31}	a_{32}	a_{34}	a_{35}

proc 1:

N_update:	1				
update:	4				
index	0	1	2	3	4
bindx:	2	5	0	2	3
val:	a_{44}	–	a_{40}	a_{42}	a_{43}

proc 2:

N_update:	2							
update:	2	5						
Index	0	1	2	3	4	5	6	7
Bindx:	3	6	8	3	4	5	2	3
val:	a_{22}	a_{55}	–	a_{23}	a_{24}	a_{25}	a_{52}	a_{53}

На практике, как было сказано выше, библиотека Aztec применяется для решения систем линейных алгебраических уравнений, возникающих при решении дифференциальных уравнений в частных производных. Для таких матриц, как правило, заранее известно количество ненулевых матричных элементов в строке и их местоположение в матрице, поэтому алгоритмы заполнения массивов val и bindx бывают достаточно простые.

15.5. Пример использования библиотеки Aztec

В заключение настоящей главы рассмотрим пример программы, в которой Aztec используется для решения системы уравнений, возникающей при конечно-разностной аппроксимации 3-х мерного

оператора Лапласа. Для простоты мы не будем использовать систему уравнений, связанную с решением какой-либо реальной физической задачи, а вместо этого будем формировать правую часть таким образом, чтобы получить заранее известное решение, например, x(i) = i + 1.

```
      program Aztec
c  Параметры:
c  ndim – максимальное число неизвестных,
c  lda   – максимальный размер матрицы
      parameter (ndim = 125000, lda = 7*ndim)
      include 'mpif.h'
      include 'az_aztecf.h'
      integer   n, nz
c
      integer i, ierror, recb(0:63), disp(0:63)
      double precision b(0:ndim), x(0:ndim), tmp(0:ndim)
      double precision s, time, total
c
      integer IAM, NPROCS
      integer N_update
      integer proc_config(0:AZ_PROC_SIZE), options(0:AZ_OPTIONS_SIZE)
      integer update(0:ndim), external(0:ndim), data_org(0:ndim)
      integer update_index(0:ndim), extern_index(0:ndim)
      integer bindx(0:lda)
      double precision params(0:AZ_PARAMS_SIZE)
      double precision status(0:AZ_STATUS_SIZE)
      double  precision val(0:lda)
      common    /global/ n
c  Инициализация MPI (для Aztec не обязательна, если не использовать
c  MPI функции)
      CALL MPI_Init(ierror)
c  Получаем номер процессора и общее число процессоров,
c  выделенных задаче
      call AZ_processor_info(proc_config)
      IAM = proc_config(AZ_node)
      NPROCS = proc_config(AZ_N_procs)
c  На 0-м процессоре печатаем пояснительную информацию и считываем
c  параметр, определяющий размер решаемой задачи
c  (максимально 50x50x50)
      if (IAM.eq.0) then
      write(6,2)
  2   format(' '/
     $   '  Aztec example driver'//
     $   '  This program creates an MSR matrix corresponding to'/
     $   '  a 7pt discrete approximation to the 3D Poisson operator'/
     $   '  on an n x n x n square and solves it by various methods.'/
     $   '  n must be <= 50.'//)
      write (6,*) 'input number of grid points on each direction n = '
      read(*,*) n
      endif
```

```
c  Рассылаем считанный параметр всем процессорам
      CALL MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
c  Вычисляем количество неизвестных в системе
      nz = n*n*n
c
c  Цикл по всем методам решения
      DO 777 K = 0,4
      time = MPI_Wtime()
c  Определяем параметры разбиения матрицы по процессорам
      call AZ_read_update(N_update, update, proc_config, nz, 1, 0)
c  Формируем матрицу в виде массивов val и bindx,
c  в bindx(0) заносим номер позиции, с которой начнут располагаться
c  недиагональные элементы, и выполняем цикл по всем строкам
      bindx(0) = N_update+1
      do 250 i = 0, N_update-1
      call add_row_7pt(update(i), i, val, bindx)
 250 continue
c  Формируем правую часть системы таким образом, чтобы получить
c  известное решение  b(i) =∑A(i,j)*j
c
      do 341 i = 0, N_update-1
      in = bindx(i)
      ik = bindx(i+1) - 1
      s = val(i)*(update(i)+1)
      do 342 j = in,ik
      s = s + val(j)*(bindx(j)+1)
342  continue
      tmp(i) = s
 341 continue
c  Преобразуем глобальную индексацию в локальную
      call AZ_transform(proc_config, external, bindx, val,update,
     $                 update_index, extern_index, data_org,
     $                 N_update, NULL,NULL,NULL,NULL, AZ_MSR_MATRIX)
c  Устанавливаем параметры для решателя
      call AZ_defaults(options, params)
      options(AZ_solver) = K
      options(AZ_precond) = AZ_dom_decomp
      options(AZ_subdomain_solve) = AZ_ilu
      options(4) = 1
      options(AZ_output) = AZ_none
      options(AZ_max_iter) = 2000
      params(AZ_tol) = 1.d-10
c  0-й процессор печатает параметры решения задачи
      if (IAM.eq.0) then
      write(6,780)
      write (6,*) '            Method of solution is =',K
      write (6,*) '            Dimension matrices =',nz,
     $ '   NPROCS = ', NPROCS
      endif
c  Преобразуем правую часть в соответствии с новой индексацией
      do 350 i = 0, N_update-1
```

```
      x(update_index(i)) = 0.0
      b(update_index(i)) = tmp(i)
 350 continue
c  Решаем систему уравнений
      call AZ_solve(x, b, options, params, NULL, bindx, NULL, NULL,
     $                    NULL, val, data_org, status, proc_config)
c  Полученное решение возвращаем к исходной индексации
      DO 415 I = 0,N_update - 1
415   tmp(i) = x(update_index(i))
c  Выполняем сборку полученных частей решения в один вектор
      CALL MPI_ALLgather(N_update,1,MPI_INTEGER,recb,1,
     *MPI_INTEGER,MPI_COMM_WORLD,ierror)
      DISP(0) = 0
      DO 404 I = 1,NPROCS-1
 404 DISP(I) = DISP(I-1) + RECB(I-1)
      CALL MPI_ALLgatherv(tmp, N_update, MPI_DOUBLE_PRECISION, x,
     *recb, disp, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierror)
c  Печатаем первую и последнюю компоненты решения
      nz1 = nz-1
      if (IAM.eq.0) then
      do 348 i = 0,nz1,nz1
      s = x(i)
      write (6,956) i,s
 348 continue
 956 format (2x,'i=',I12,2x,'x(i)=',F18.8)
      endif
      time = MPI_Wtime() - time
      if(iam.eq.0) write(*,435) time
435   FORMAT(2x,'TIME CALCULATION (SEC.) = ',F12.4)
777   continue
      CALL MPI_FINALIZE(IERROR)
      stop
      end
c  Подпрограмма добавления очередной строки матрицы в массивы
c  val и bindx
      subroutine add_row_7pt(row, location, val, bindx)
      integer row, location, bindx(0:*)
      double precision val(0:*)
      integer   n3, n, k
      common    /global/ n
c  Входные параметры:  row        – глобальный номер добавляемой строки
c                      location   – локальный номер строки в процессоре
c  n3 – максимально возможный номер столбца
      n3 = n*n*n - 1
c  Далее идет проверка: существует ли в трехмерной решетке ближайший сосед
c  по каждому из направлений; если да, то соответствующий матричный элемент
c  заносится  и счетчик увеличивается на 1
      k = bindx(location)
      bindx(k)  = row + 1
      if (bindx(k).le.n3) then
      val(k) = -1.
```

```
      k = k + 1
      endif
      bindx(k)  = row - 1
      if (bindx(k).ge.0) then
      val(k) = -1.
      k = k + 1
      endif
      bindx(k)  = row + n
      if (bindx(k).le.n3) then
      val(k) = -1.0
      k = k + 1
      endif
      bindx(k)  = row - n
      if (bindx(k).ge.0) then
      val(k) = -1.0
      k = k + 1
      endif
      bindx(k)  = row + n*n
      if (bindx(k).le.n3) then
      val(k) = -1.0
      k = k + 1
      endif
      bindx(k)  = row - n*n
      if (bindx(k).ge.0) then
      val(k) = -1.0
      k = k + 1
      endif
c  Занесение диагонального матричного элемента
      val(location)   = 6.0
c  Подготовка следующего шага:  заносим номер позиции, с которой
c  будут располагаться недиагональные элементы следующей строки
      bindx(location+1) = k
      return
      end
```

ЗАКЛЮЧЕНИЕ К ЧАСТИ 3

Рассмотренные в этой книге библиотеки параллельных подпрограмм, конечно же, не исчерпывают весь список имеющихся в мире библиотек. Достаточно полный список таких пакетов можно найти на сервере [20]. Выбор именно этих библиотек обусловлен их универсальностью и тем, что они реально используются широким кругом пользователей.

ЛИТЕРАТУРА И ИНТЕРНЕТ-РЕСУРСЫ

1. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982. – 416 с.
2. Воеводин Вл. В. Легко ли получить обещанный гигафлоп? // Программирование. – 1995. – № 4. – С. 13-23.
3. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 600 с.
4. The Cost Effective Computing Array (COCOA). – http://cocoa.aero.psu.com
5. ScaLAPACK Users' Guide. 1997. – http://www.netlib.org/scalapack/scalapack_home.html http://rsusu1.rnd.runnet.ru/ncube/scalapack/scalapack_home.html
6. The OpenMP Application Program Interface (API). – http://www.openmp.org
7. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. – Version 1.1. 1995. – http://www-unix.mcs.anl.gov/mpi
8. High Performance Fortran Language Specification. High Performance Fortran Forum. – Version 2.0. 1997. – http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20
9. ADAPTOR. High Performance Fortran (HPF) Compilation System. – http://www.gmd.de/SCAI/lab/adaptor
10. Коновалов Н. А., Крюков В. А., Погребцов А. А., Сазанов Ю. Л. C-DVM – язык разработки мобильных параллельных программ. // Программирование. – 1999. – № 1. – С. 20-28.
11. Ian Foster. Designing and Building Parallel Programs. – http://www.hensa.ac.uk/parallel/books/addison-wesley/dbpp http://rsusu1.rnd.runnet.ru/ncube/design/dbpp/book-info.html
12. G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. // Proc. 1967 AFIPS Conf., AFIPS Press. – 1967. – V. 30. – P. 483.
13. nCUBE 2 Programmers Guide / r2.0, nCUBE Corporation, Dec., 1990.
14. Portable Batch System. – http://www.openpbs.org
15. MPI: The Complete Reference. – http://rsusu1.rnd.runnet.ru/ncube/mpi/mpibook/mpi-book.html
16. MPI: The Message Passing Interface. – http://parallel.ru/tech/tech_dev/mpi.html
17. The ScaLAPACK Project. – http://www.netlib.org/scalapack
18. Aztec. A Massively Parallel Iterative Solver Library for Solving Sparse Linear Systems. – http://www.cs.sandia.gov/CRF/aztec1.html
19. Aztec User's Guide. Version 1.1 – http://rsusu1.rnd.runnet.ru/ncube/aztec/index.html
20. Специализированные параллельные библиотеки. – http://parallel.ru/tech/tech_dev/par_libs.html

А. А. Букатов, В. Н. Дацюк, А. И. Жегуло.

Программирование многопроцессорных вычислительных систем

Редактор Букатов А. А.

Издательство ООО «ЦВВР». Лицензия ЛР № 65-36 от 05.08.99 г.
Подписано в печать 08.08.2003.г. Заказ № 405
Бумага офсетная, Гарнитура «Таймс», печать офсетная.
Тираж 200 экз. Печ. лист 13,00. Формат 60*84 1/16. Усл.печ.л. 12,09.
Компьютерный набор и верстка.
Издательско-полиграфический комплекс « Биос» РГУ
344090, г. Ростов-на-Дону, ул. Зорге, 28/2, корп. 5 «В», 4 этаж.
Лицензия на полиграфическую деятельность № 65-125 от 09.02.98 г.