Министерство образования Российской Федерации
Ярославский государственный университет имени П.Г. Демидова

В.Д. Корнеев

Параллельное программирование в MPI

Учебное пособие

Ярославль 2002

УДК 518.65
ББК з 973.2 – 018
К 67

Корнеев В.Д.

Параллельное программирование в MPI: Учеб. пособие / Яросл. гос. ун-т. Ярославль, 2002. 104 с.
ISBN 5-8397-0239-0

Пособие посвящено параллельному программированию на базе системы с передачей сообщений MPI. Система MPI является основным средством программирования таких современных высокопроизводительных мультикомпьютеров, как Silicon Graphics Origin 2000, Cray T3D, Cray T3E, IBM SP2 и многих других. Рассмотрены примеры параллельного программирования алгоритмов решения различных стандартных задач: умножения матрицы на вектор и на матрицу, решение систем линейных уравнений (СЛАУ) методом Гаусса и решение СЛАУ методом простой итерации. Дано краткое описание модифицированной версии 1.1 MPI стандарта, приведено описание основных функций MPI.

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

Пособие является руководством при выполнении лабораторных занятий, проводимых со студентами в терминальном классе. Оно может быть полезным для инженеров, аспирантов и сотрудников, осваивающих параллельное программирование на многопроцессорных вычислительных системах.

Печатается при финансовой поддержке федеральной целевой программы "Интеграция" (контракт № А-0068).

Рецензенты: кафедра прикладной математики и вычислительной техники Ярославского государственного технического университета; доктор технических наук В.Э. Малышкин

ISBN 5-8397-0239-0

Введение

Цель пособия – практическое освоение основных приемов параллельного программирования на мультикомпьютерных вычислительных системах с передачей сообщений.

Практический материал для освоения параллельного программирования дан на базе системы параллельного программирования с передачей сообщений MPI [1, 9]. Система MPI является основным средством программирования таких современных высокопроизводительных мультикомпьютеров, как Silicon Graphics Origin 2000, Cray T3D, Cray T3E, IBM SP2 и многих других. MPI работает на самых разных архитектурах мультикомпьютеров как с распределенной памятью, так и с разделяемой памятью. Кроме того, MPI работает на сетях компьютеров (кластерах) однородных и гетерогенных. Количество компьютеров в сети может быть от одного и более. Важнейшей особенностью MPI является то, что пользователь при написании своих параллельных программ не должен учитывать все эти архитектурные особенности конкретных мультикомпьютеров, поскольку MPI предоставляет пользователю виртуальный мультикомпьютер с распределенной памятью и с виртуальной сетью связи между виртуальными компьютерами. Пользователь заказывает количество компьютеров, необходимых для решения его задачи, и определяет топологию связей между этими компьютерами. MPI реализует этот заказ на конкретной физической системе. Ограничением является объем оперативной памяти физического мультикомпьютера. Таким образом пользователь работает в виртуальной среде, что обеспечивает переносимость его параллельных программ. Система MPI представляет собой библиотеку средств параллельного программирования для языков C и Fortran 77.

В чем одно из важных отличий в написании последовательной и параллельной программ?

Здесь имеется в виду параллельная программа для рассматриваемых систем с передачей сообщений. Прежде чем создавать параллельную программу, необходимо знать общую архитектуру параллельной машины (в данном случае виртуальной) и топологию межпроцессорных связей (в данном случае виртуальных), которая существенно используется при программировании. Это связано с тем, что невозможно создание эффективного автоматического распараллеливателя, который позволял бы превращать последовательную программу в параллельную и обеспечивал бы ее высокую производительность. Поэтому в программе приходится в явном виде задавать функции инициации виртуальной топологии и функции обменов данными между процессорами. При написании же последовательной программы знать архитектуру процессора, на котором будет исполняться программа, зачастую нет необходимости, поскольку учет осо-

бенностей архитектуры скалярного процессора может быть сделан компилятором с приемлемыми потерями в производительности программы.

Почему MPI?

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

Эффективность и надежность обеспечиваются:

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

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

Переносимость обеспечивается:

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

4) заданием *виртуальных топологий* (см. п. 2.1). Отображение виртуальных топологий на физическую систему осуществляется системой MPI. Виртуальные топологии обеспечивают оптимальное приближение архитектуры системы к структурам задач при хорошей переносимости задач;

5) компиляторами для Fortran(a) и C.

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

Методы распараллеливания и модели программ, поддерживаемые MPI

Одной из целей, преследуемых при решении задач на вычислительных системах, в том числе и на параллельных, – является эффективность. Эффективность параллельной программы существенно зависит от соотношения времени вычислений ко времени коммуникаций между компьютерами (при обмене данными). И чем меньше в процентном отношении доля времени, затраченного на коммуникации, в общем времени вычислений, тем больше эффективность. Для параллельных систем с передачей сообщений оптимальное соотношение между вычислениями и коммуникациями обеспечивают методы крупнозернистого распараллеливания, когда параллельные алгоритмы строятся из крупных и редко взаимодействующих блоков [2 – 8]. Задачи линейной алгебры, задачи, решаемые сеточными методами, и многие другие достаточно эффективно *распараллеливаются крупнозернистыми методами*.

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

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

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

Указанные модели, помимо поддержки на языковом уровне, поддерживаются архитектурами таких самых современных суперкомпьютеров, как ASCI RED (более 9 000 компьютеров Pentium PRO/200 объединены в единую систему, которая имеет быстродействие около 3,2 Tflops) и ASCI WAIT (8 192 компьютеров, быстродействие 12,2 Tflops), Cray T3D, Cray T3E, IBM SP2.

В первой части дано описание системы параллельного программирования MPI. Даны операторы компиляции и запуска С-программ, программные средства задания системных взаимодействий, виртуальные топологии. В каждом подразделе приводятся примеры программ, закрепляющие понимание и усвоение материала. Эти же примеры могут использоваться как образцы для написания новых программ.

Во второй части даны четыре лабораторных работы, построенные как последовательность шагов по изучению программных средств параллельного программирования и освоению основных навыков написания параллельных программ.

1. Система параллельного программирования MPI

В этом разделе кратко изложены основные функции MPI, необходимые для освоения основных приемов параллельного программирования. К таковым относятся функции парных и коллективных взаимодействий между процессами, функции задания виртуальных топологий и конструирования пользовательских типов данных. Все примеры программ приведены здесь на языке С. Предварительно сделаем некоторые пояснения и опишем операторы компиляции и запуска параллельных программ.

1.1. Компиляция и запуск параллельной программы

Во введении были употреблены понятия "виртуальный компьютер" и "виртуальная топология". Под *виртуальным компьютером* понимается программно реализуемый компьютер. Виртуальный компьютер работает в режиме интерпретации его физическим процессором. В одном физическом компьютере может находиться и работать одновременно столько виртуальных компьютеров,

сколько позволяет память физического компьютера. Работают виртуальные компьютеры в одном физическом режиме квантования времени. Под *виртуальной топологией* здесь понимается программно реализуемая топология связей между виртуальными компьютерами на физической системе.

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

Как уже было сказано во введении, MPI работает на вычислительных системах (ВС) с разнообразной архитектурой. Однако запуск параллельной программы зависит от типа ВС. Различаются запуски параллельных программ для сильносвязных ВС и для слабосвязных. Сильносвязными являются ВС как с разделяемой памятью, например Silicon Graphics Origin 2000, так и с распределенной памятью с быстрыми каналами, компьютеры которых сосредоточены в небольшом пространстве, например Cray T3D, Cray T3E, IBM SP2. Слабосвязными являются ВС с компьютерами объединенными (в кластер) обычной сетью связи.

Запуск параллельной программы продемонстрируем на примере. Допустим, требуется решить некоторую задачу. Алгоритм задачи распараллелен на `N` процессов, независимо выполняющихся и взаимодействующих друг с другом. Здесь рассматривается два варианта: 1) ветви параллельной программы реализуются копиями одной и той же программы; 2) ветви параллельной программы реализуются разными программами.

Вначале рассмотрим *запуск программы на сильносвязной ВС и на одном обычном однопроцессорном компьютере.* На сильносвязной ВС и на одном компьютере запускаются только параллельные программы, ветви которой реализуются копиями одной и той же программы. Пусть программа имеет имя: `program.c`. Программа предварительно компилируется:

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

В квадратных скобках стоят опции нужной оптимизации. Необходимое количество виртуальных компьютеров задаются пользователем в командной строке:

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

N = {1,2,3,…} - указывает количество виртуальных компьютеров, необходимых для решения рассматриваемой программы с именем - program.exe. По этой команде система MPI создает (в оперативной памяти системы из M физических компьютеров) N виртуальных компьютеров, объединенных виртуальными каналами связи со структурой *полный граф*. И этой группе виртуальных компьютеров присваивается стандартное системное имя MPI_COMM_WORLD, после чего пользовательская программа program.exe загружается (копируется) в память каждого из созданных виртуальных компьютеров и стартует (program.exe предварительно должна находиться во всех физических компьютерах в соответствующей директории). Если M < N, то в некоторых (или всех) физических компьютерах будет создано несколько виртуальных. Виртуальные компьютеры, расположенные в одном физическом, будут работать в режиме интерпретации их физическим процессором с разделением времени. Созданные виртуальные компьютеры имеют линейную нумерацию - {0,1,2,3,…} и являются базой для создания различных виртуальных топологий, необходимых для реализации конкретных задач, причем со своей внутренней нумерацией виртуальных компьютеров. (Эта идентификация виртуальных компьютеров в различных структурах и тип топологии связи позволяют "ориентироваться" в системе связей компьютеров копиям пользовательской программы program.exe.)

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

Для однородных и неоднородных ВС запуск программы осуществляется следующей командой:

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

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

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

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

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

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

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

2. Количество и чередование имен компьютеров в списке может быть самым разнообразным. Допустим у нас та же вычислительная система и тот же

заказ виртуальных компьютеров, что и в предыдущем случае. Файл со списком компьютеров может быть и таким:

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

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

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

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

В этом случае в командной строке компьютера host должно находиться имя ветви, стоящей в этом же компьютере. Например, если программа в приведенном примере запускается с компьютера `ssd`, то имя программы в команде `mpirun` должно быть `program2.exe`.

ПРИМЕР 1.1

Программа hello.c

```
#include<mpi.h>
#include<stdio.h>
```

```
int main(int argc, char **argv)
  { printf("Hello, World\n");
    return (0);
  }
```

Компиляция (если ВС неоднородна, то компиляция в каждом компьютере):

```
mpicc  -o hello.exe hello.c
```

Запуск:

```
mpirun  -mashinesfile machines.s -np 4 hello.exe
```

Результат:

```
Hello, World
Hello, World
Hello, World
Hello, World
```

Результат выводится на экран монитора того компьютера, с которого осуществляется запуск программы.

1.2. Коммуникаторы

Коммуникатор (переключатель каналов) - это массив указателей на другие коммуникаторы. Коммуникатор можно представлять себе как сетевую карту компьютера, с помощью которой он связан каналами с другими компьютерами. Группа связанных между собой коммуникаторов (или, что то же самое, компьютеров) определена так, что:

- их каналы формируют полный граф: каждый коммуникатор связан со всеми коммуникаторами в наборе, включая себя;
- каналы имеют совместимые индексы: в каждом коммуникаторе связь `i` указывает на коммуникатор для процесса `i`.

Эта распределенная структура данных коммуникатора иллюстрируется на рис. 1.1 для случая из трех членов коммуникатора.

Каждый коммуникатор имеет имя, которое является уникальным и идентичным во всех коммуникаторах, определяющих одну и ту же связанную группу. Это имя выступает как бы контекстом связи и в то же время параметром многих функций MPI. Коммуникатор используется для задания на связанной группе различных топологий, а также парных и коллективных взаимодействий между процессами. Одна и та же группа компьютеров может быть объединена множеством разных коммуникаторов, что способствует устранению конфликтных ситуаций при обмене данными внутри разных групп процессов.

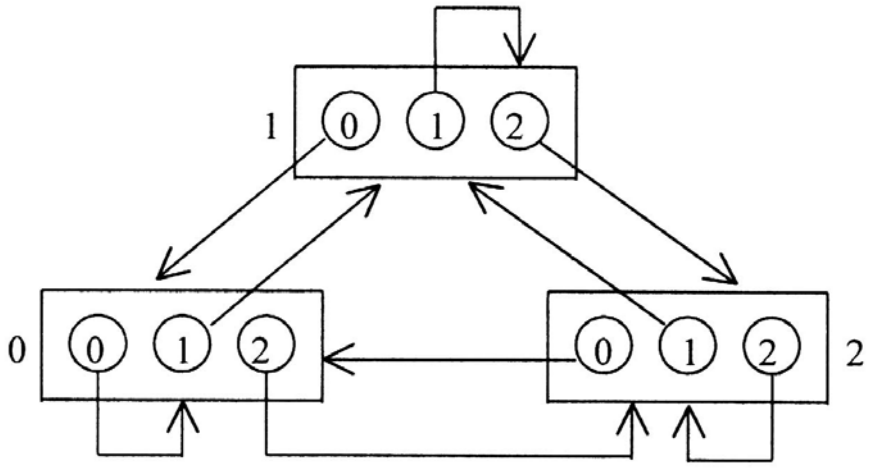


Рис. 1.1. Распределенная структура данных коммуникатора для группы связанных коммуникаторов

После того как пользователь заказал в операторе запуска программы необходимое количество компьютеров, система создает эти (виртуальные) компьютеры и объединяет их универсальным и предопределенным коммуникатором с именем MPI_COMM_WORLD. Созданным виртуальным компьютерам присваиваются порядковые номера от 0 до N-1. Этот коммуникатор служит базовым для всех остальных создаваемых коммуникаторов. Если создан новый коммуникатор, то на основе этого коммуникатора могут быть созданы другие коммуникаторы и т.д.

Далее рассмотрим некоторые операции с коммуникаторами. Операции, которые осуществляют доступ к коммуникаторам, локальные, и их выполнение не требует межпроцессорной связи. Операции, которые создают (уничтожают) коммуникаторы, коллективные, они требуют межпроцессорной связи. Операции представлены в виде функций, а их запись - в трех видах: в общем виде с пояснениями параметров функции, на языке C и на языке Fortran 90.

1.2.1. Информационные функции

Функции, представленные в этом пункте, локальные.

```
MPI_COMM_SIZE(comm, size)
IN  comm          имя коммуникатора
OUT size          количество процессов в подмножестве
                  comm

int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR
```

MPI_COMM_SIZE возвращает размер подмножества, связанного коммуникатором comm.

```
MPI_COMM_RANK(comm, rank)
IN  comm          имя коммуникатора
OUT rank          ранг вызвавшего процесса в comm

int MPI_Comm_rank(MPI_Comm comm, int *rank)

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

MPI_COMM_RANK указывает номер процесса, который вызывает эту функцию; номер в диапазоне от 0,…,size-1, где size – значение, возвращаемое функцией MPI_COMM_SIZE.

1.2.2. Функции создания копии и уничтожения коммуникатора

Следующие функции являются коллективными, они должны синхронно вызываться всеми процессами в множестве процессов, именованных comm.

```
MPI_COMM_DUP(comm, newcomm)
IN  comm              имя коммуникатора
OUT newcomm           копия коммуникатора comm

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
```

MPI_COMM_DUP создает новый коммуникатор newcomm с теми же самыми установленными атрибутами, как и входящие в comm. Новый созданный коммуникатор из процессов коммуникатора comm определяет новую область связи с тем же самым множеством процессов, как и в старом коммуникаторе. Когда коммуникатор дублирован, само множество процессов не копируется, а только прибавляется новая ссылка и увеличивается индекс ссылки. Эта операция может использоваться, чтобы обеспечить параллельный вызов из библиотек модулей, дублирующих связи, которые имеют те же самые реквизиты, как и первоначальный коммуникатор. Типичный запрос мог бы вызывать MPI_COMM_DUP в начале параллельного запроса и MPI_COMM_FREE для того дублированного коммуникатора в конце запроса.

```
MPI_COMM_FREE(comm)
INOUT comm          имя уничтожаемого коммуникатора
```

```
int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR)
INTEGER COMM, IERROR
```

Эта коллективная операция регистрирует объект связи для освобождения. Имя коммуникатора принимает значение `MPI_COMM_NULL`. Любые операции, которые использует в текущий момент этот переключатель каналов, завершаются обычным образом; объект фактически освобожден, только если не имеется никаких других активных ссылок на него. Совмещение имен переключателей каналов (например, comma = commb) возможно, однако не рекомендуется. После запроса `MPI_COMM_FREE` любое совмещенное имя переключателя каналов будет оставлено в неопределенном состоянии.

1.3. Виртуальные топологии

В этом пункте описывается механизм виртуальных топологий MPI. Под виртуальной топологией здесь понимается программно реализуемая топология в виде конкретного графа, например: кольцо, решетка, тор, звезда, дерево и вообще произвольно задаваемый граф на существующей физической топологии. Виртуальная топология обеспечивает очень удобный механизм наименования процессов, связанных коммуникатором, и является мощным средством отображения процессов на оборудование системы. Виртуальная топология в MPI может задаваться только в группе процессов, объединенных коммуникатором.

Как сказано в п. 1.2, группа процессов в MPI - это набор из `n` процессов. Каждому процессу в группе назначен номер между `0` и `n-1`. Во многих параллельных приложениях линейная нумерация процессов адекватно не отражает логическую модель связи процессов (которая обычно определяется основной геометрией задачи и определенным используемым алгоритмом). Часто параллельные алгоритмы представляются в топологических моделях типа двумерных или объемных сеток. В более общем случае логическое расположение процессов описывается некоторым произвольным графом.

Нужно различать виртуальную топологию процессов и топологию основного, физического оборудования. Механизм виртуальных топологий значительно упрощает и облегчает написание параллельных программ, делает программы легко читаемыми и понятными. Пользователю при этом не нужно программировать схему физических связей процессоров, а только схему виртуальных связей между процессами. Отображение виртуальных связей на физические связи осуществляет система, что делает параллельные программы машинно-независимыми и легкопереносимыми. Функции (в этой главе) осуществляют только машинно-независимое отображение.

1.3.1. Функции декартовых топологий

Ниже описываются функции MPI для создания декартовых топологий.

Функция, конструирующая декартову топологию

MPI_CART_CREATE используется для описания декартовой структуры произвольного измерения. Для каждого направления координаты определяется, является ли структура процесса периодической или нет. Для 1D топологии – это линейка, если она не периодическая, и кольцо, если она периодическая. Для 2D топологии – это прямоугольник, цилиндр или тор и т.д.

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder,
                                                  comm_cart)
IN  comm_old       входной (старый) коммуникатор
IN  ndims          количество измерений в декартовой топологии
IN  dims           целочисленный массив размером ndims,
                   определяющий количество процессов в каждом
                   измерении
IN  periods        массив размером ndims логических значений,
                   определяющих периодичность (true) или нет
                   (false) в каждом измерении
IN  reorder        ранги могут быть перенумерованы (true)
                   или нет (false)
OUT comm_cart      коммуникатор новой (созданой) декартовой
                   топологии

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
             int *periods, int reorder, MPI Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER,
                                       COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER
```

MPI_CART_CREATE возвращает управление новому коммуникатору, к которому присоединена информация декартовой топологии. Эта функция коллективная. Как в случае с другими коллективными функциями, вызов этой функции должен быть синхронизован во всех процессах. Если reorder = false, тогда номер каждого процесса в новой группе идентичен ее номеру в старой группе. Иначе функция может переупорядочивать процессы (возможно, чтобы выбрать хорошее отображение виртуальной топологии на физическую топологию). Если полный размер декартовой сетки меньше, чем размер группы comm_old, то некоторые процессы возвращают MPI_COMM_NULL, по аналогии с MPI_COMM_SPLIT. Запрос ошибочен, если он определяет сетку, которая является большей, чем размер группы comm_old.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Рис. 1.2. Связь между рангами и декартовыми координатами для 3x4 2D топологии, верхний номер в каждой клетке - номер процесса, а нижнее значение (строка, столбец) - координаты.

Декартова функция задания решетки

Для декартовой топологии функция MPI_DIMS_CREATE помогает пользователю выбрать сбалансированное распределение процессов по направлению координат в зависимости от числа процессов в группе и необязательных ограничений, которые могут быть определены пользователем. Одно возможное использование этой функции - это разбиение всех процессов (размер группы MPI_COMM_WORLD) в N-мерную топологию.

```
MPI_DIMS_CREATE(nnodes, ndims, dims)
IN    nnodes          количество узлов в решетке
IN    ndims           размерность декартовой топологии
INOUT dims            целочисленный массив, определяющий
                      количество узлов в каждой размерности

int MPI_Dims_create(int nnodes, int ndims, int *dims)

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

Элементы в массиве dims представляют описание декартовой решетки с размерностями ndims и общим количеством узлов nnodes. Размерности устанавливаются так, чтобы быть близко друг к другу насколько возможно, используя соответствующий алгоритм делимости. Пользователь может ограничивать действие этой функции, определяя элементы массива dims. Если в dims[i] уже записано число, то функция не будет изменять количество узлов в измерении i; функция модифицирует только нулевые элементы в массиве, т.е. где dims[i] = 0. Отрицательные значения элементов dims(i) ошибочены. Ошибка будет выдаваться, если nnodes не кратно $\prod_{(i,dims[i]\neq 0)}$ dims[i].

Для dims[i], установленных функцией, dims[i] будут упорядочены в монотонно уменьшающемся порядке. Массив dims подходит для использования как вход к функции MPI_CART_CREATE. Функция MPI_DIMS_CREATE локальная. Отдельные типовые запросы показываются в примере 1.1.

ПРИМЕР 1.2

dims перед вызовом	функции	dims после возврата
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	ошибка

Декартовы информационные функции

Если декартова топология создана, может возникнуть необходимость запросить информацию относительно этой топологии. Эти функции даются ниже, и все они локальные.

```
MPI_CARTDIM_GET(comm, ndims)
IN  comm          коммуникатор с декартовой топологией
OUT ndims         размерность декартовой топологии

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
INTEGER COMM, NDIMS, IERROR
```

MPI_CARTDIM_GET возвращает число измерений декартовой топологии, связанной с коммуникатором comm. Она может использоваться для обеспечения других декартовых функций надлежащим размером массивов. Коммуникатор с топологией на рис. 1.2 возвратил бы ndims = 2.

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)
IN  comm      коммуникатор с декартовой топологией
IN  maxdims   максимальный размер массивов dims, periods
              и coords в вызывающей программе
OUT dims      массив значений, указывающих количество процессов
              в каждом измерении
OUT periods   массив значений, указывающих периодичность
true/false)   в каждом измерении
OUT coords    координаты вызвавшего процесса в декартовой
              топологии

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,
                              int *periods, int *coords)

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

MPI_CART_GET возвращает информацию относительно декартовой топологии, связанной с коммуникатором comm. maxdims, и должен быть, по крайней мере, равен ndims, как возвращает MPI_CARTDIM_GET. Для примера на рис. 1.2 dims = (3,4), а в процессе 6 функция возвратит coords = (1,2).

Декартовы функции транслирования

Функции, приведенные в этом пункте, переводят из ранга в декартовы координаты топологии. Эти запросы локальные.

```
MPI_CART_RANK(comm, coords, rank)
IN  comm          коммуникатор с декартовой топологией
IN  coords        целочисленный массив, определяющий координаты
                  нужного процесса в декартовой топологии
OUT rank          ранг нужного процесса

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

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

Для группы процессов с декартовой структурой функция MPI_CART_RANK переводит логические координаты процессов в номера. Эти номера процессы используют в парных взаимодействиях между процессами. coords - массив размером ndims, как возвращает MPI_CARTDIM_GET. Для примера на рис. 1.2 процесс с coords = (1,2) возвратил бы rank = 6. Для измерения i с periods(i) = true, если координата coords(i) находится вне диапазона, т.е. coords(i) < 0 или coords(i) >= dims(i), она перемещается назад к интервалу 0 <= coords(i) < dims(i) автоматически. Если топология на рис. 1.2 периодическая в обеих размерностях, то процесс с coords = (4,6) также возвратил бы rank = 6. Для непериодических размерностей диапазон вне координат ошибочен.

```
MPI_CART_COORDS(comm, rank, maxdims, coords)
IN  comm          коммуникатор с декартовой топологией
IN  rank          ранг процесса в топологии comm
IN  maxdims       максимальный размер массивов dims, periods
                  и coords в вызывающей программе

OUT coords        целочисленный массив, определяющий координаты
                  нужного процесса в декартовой топологии

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int
*coords)

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

MPI_CART_COORDS переводит номер процесса в координаты процесса в топологии. Это обратное отображение MPI_CART_RANK. maxdims можно взять, например, равным ndims, возвращенным MPI_CARTDIM_GET. Для примера на рис. 1.2 процесс с rank = 6 возвратил бы coords = (1,2).

Декартова функция смещения

Если в декартовой топологии используется функция MPI_SENDRECV (см. далее п. 3.2) для смещения данных вдоль направления какой-либо координаты, то входным аргументом MPI_SENDRECV берется номер процесса source (процесса источника) для приема данных и номер процесса dest (процесса назначения) для передачи данных. Операция смещения в декартовой топологии определяется координатой смещения и размером шага смещения (положительным или отрицательным). Функция MPI_CART_SHIFT возвращает информацию для входных спецификаций, требуемых в вызове MPI_SENDRECV. Функция MPI_CART_SHIFT локальная.

```
MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)
IN  comm          коммуникатор с декартовой топологией
IN  direction     номер измерения (в топологии), где делается
                  смещение
IN  disp          направление смещения (> 0: смещение в сторону
                  увеличения номеров координаты direction, < 0:
                  смещение в сторону уменьшения номеров
                  координаты direction)
OUT rank_source   ранг процесса источника
OUT rank_dest     ранг процесса назначения

int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                           int *rank_source, int *rank_dest)

MPI_CART_SHIFT (COMM, DIRECTION, DISP, RANK SOURCE, RANK DEST,
IERROR)
INTEGER COMM, DIRECTION, DISP, RANK SOURCE, RANK DEST, IERROR
```

Аргумент direction указывает измерение, в котором осуществляется смещение данных. Измерения маркируются от 0 до ndims-1, где ndims - число размерностей. disp указывает направление и величину смещения. Например, в топологии "линейка" или "кольцо" с $N \geq 4$ процессами для процесса с номером 1 при disp = +1 rank_source = 0, а rank_dest = 2; при disp = -1 rank_source = 2, а rank_dest = 0. Для этого же процесса 1 при disp = + 2 rank_source = N-1 для "кольца" и MPI_PROC_NULL для "линейки", а rank_dest = 3 для обоих структур; при disp = -2 rank_source = 3 для обоих структур, а rank_dest = N-1 для "кольца" и MPI_PROC_NULL для "линейки".

В зависимости от периодичности декартовой топологии в указанном направлении координат, MPI_CART_SHIFT обеспечивает идентификаторы rank_source и rank_dest для кольцевого или некольцевого смещения данных. Это имеющие силу входные аргументы к функции MPI_SENDRECV. Ни MPI_CART_SHIFT, ни MPI_SENDRECV не коллективные функции. Не требуется, чтобы все процессы в декартовой решетке одновременно вызвали MPI_CART_SHIFT с теми же самыми direction и disp аргументами, но только тот процесс, который посылает (соответственно, получает) в последующих запросах к MPI_SENDRECV.

ПРИМЕР 1.3

Создание двумерной решетки компьютеров 3 x 4, представленной на рис.1.2. Каждый компьютер находит порядковые номера своих соседей, показанные на том же рисунке.

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char **argv)
  { int size, rank, coords[2], source, dest;
    int dims[2];
    int reorder, periods[2];
    MPI_Comm comm_2d;
 /* Инициализация библиотеки MPI */
    MPI_Init(&argc, &argv);

 /* Каждая ветвь определяет размер системы, если он больше
  * 12, то завершение */
    MPI_Comm_size(MPI_COMM_WORLD, size);
    if(size > 12)
      MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);

 /* Каждая ветвь определяет свой номер */
    MPI_Comm_rank(MPI_COMM_WORLD, rank);

 /* Создание топологии "двумерная решетка" с количеством
  * компьютеров = 12 */

    dims[0] = 3;
    dims[1] = 0;
 /* Делаем разбивку всех компьютеров на двумерную решетку, т.е. с
  * помощью функции MPI_DIMS_CREATE заполняем массив размерностей
  * dims, с учетом того, что dims[0] = 3, указанная фукция запишет
  * dims[1] = 4 */
    MPI_CART_CREATE(size, 2, dims);

    periods[0] = 0;   /* грани вдоль 0-координаты не замкнуты */
    periods[1] = 0;   /* грани вдоль 1-координаты не замкнуты */
    reorder = 0;      /* в новой топологии номера компьютеров
                         останутся прежними */
```

```
   MPI_CART_CREATE(MPI_COMM_WORLD,  2,  dims,  periods,  reorder,
   comm_2d);
/* 12 процессов из MPI_COMM_WORLD объединяются в группу comm_2d,
 * с рангами как в MPI_COMM_WORLD */
/* Находим декартовы координаты */
   MPI_Cart_coords(comm_2d, rank, 2, coords);
/* Каждый компьютер вычисляет соседей  source и dest вдоль
 * нулевой координаты */
   MPI_Cart_shift(comm_2d, 0, 1, source, dest);
/* Каждый компьютер вычисляет соседей  source и dest вдоль
 * первой координаты */
   MPI_Cart_shift(comm_2d, 1, 1, source, dest);


/* Продолжение программы */
   MPI_Finalize();
 }
```

Сделаем необходимые пояснения к этому примеру. Во-первых, `#include<mpi.h>` достаточен для всей библиотеки MPI; во-вторых, `main(int argc, char **argv)` записывается только так; в-третьих, функция `MPI_Init(&argc, &argv)` должна быть обязательно, и она должна стоять в программе перед первой функцией MPI. `MPI_Finalize()` должна быть обязательно, и она должна завершать работу с MPI.

Декартова функция разбиения

```
MPI_CART_SUB(comm, remain_dims, newcomm)
IN  comm          communicator для декартовой топологии
IN  remain_dims   i-й элемент remain_dims определяет
                  соответствующую i-ю размерность, включаемую
                  (true) или не включаемую (false) в подрешетку
OUT newcomm       communicator созданных подрешеток

int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
*newcomm)

MPI_CART_SUB(COMM, REMAIN DIMS, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
LOGICAL REMAIN DIMS(*)
```

Если декартова топология была создана функцией `MPI_CART_CREATE`, то может использоваться функция `MPI_CART_SUB` для разбиения группы, связанной коммуникатором, на подгруппы, которые формируют декартовы подрешетки меньшей размерности, и можно строить для каждой такой подгруппы

коммуникатор, связанный с подрешеткой декартовой топологии. Этот запрос коллективный.

ПРИМЕР 1.4

Предположим, что MPI_CART_CREATE(...,comm) определяет (2 x 3 x 4) решетку. Допустим remain_dims =(true, false, true). Тогда запрос к MPI_CART_SUB(comm, remain_dims, comm_new) создаст три коммуникатора каждый с восьмью процессами 2 x 4 в декартовой топологии. Если remain_dims =(false, false, true), то запрос к MPI_CART_SUB(comm, remain_dims, comm_new) создаст шесть непересекающихся коммуникаторов (каждый с четырьмя процессами) в одномерной декартовой топологии.

1.3.2 Функции создания топологии графа

Этот пункт описывает функции MPI для создания топологии графа.

Функция построения графа

```
MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder,
                                                comm_graph)
IN comm_old      входной коммуникатор
IN nnodes        количество узлов в графе
IN index         целочисленный массив для описания узлов графа
IN edges         целочисленный массив для описания ребер графа
IN reorder       переупорядочить ранги (true) или нет (false)
OUT comm_graph   коммуникатор с топологией построенного графа

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index,
                 int *edges, int reorder, MPI_Comm *comm_graph)

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER,
                                          COMM_GRAPH, IERROR)
INTEGER COMM OLD, NNODES, INDEX(*), EDGES(*), COMM GRAPH, IERROR
LOGICAL REORDER
```

MPI_GRAPH_CREATE возвращает новый коммуникатор, к которому присоединена информация топологии графа. Если reorder=false, то ранг каждого процесса в новой группе идентичен ее рангу в старой группе; иначе функция может переупорядочивать процессы. Если размер, nnodes, графа меньше, чем размер группы comm_old, то некоторые процессы возвращают MPI_COMM_NULL. Запрос ошибочен, если он определяет граф, который является большим, чем размер группы, определяемой comm_old. Эта функция коллективная. Как и с другими коллективными запросами, перед вызовом функций процессы нужно синхронизовать.

Три параметра nnodes, index и edges определяют структуру графа. nnodes - число узлов графа. Узлы маркируются от 0 до nnodes-1. i-й элемент массива index хранит общее число соседей первых i узлов графа. Списки соседних узлов 0, 1, ..., nnodes-1 хранятся в массиве edges. Массив edges - сжатое представление списков ребер. Общее число элементов в index равно nnodes, и общее число элементов в edges равно числу ребер графа. Определения аргументов nnodes, index и edges иллюстрируются в примере 1.5.

ПРИМЕР 1.5

Предположим, что имеются четыре компьютера 0, 1, 2, 3 со следующей матрицей смежности:

Комп.	Соседи
0	1, 3
1	0
2	3
3	0, 2

Тогда, входные аргументы будут иметь следующие значения:

```
nnodes = 4
index = (2, 3, 4, 6)
edges = (1, 3, 0, 3, 0, 2)
```

Таким образом, на C, index[0] – это степень нулевого узла, и index[i] - index[i-1] - это степень узла i, i=1,…,nnodes-1; список соседей нулевого узла хранится в edges[j], для 0 <= j <= index[0]-1 и список соседей узла i, i > 0 хранится в edges[j], для index[i-1] <= j <= index[i]-1.

В Fortran(e), index(1) – это степень нулевого узла, и index(i+1)-index(i) – это степень узла i, i=1,…,nnodes-1; список соседей нулевого узла хранится в edges(j), для 1 <= j <= index(1) и список соседей узла i, i > 0, хранится в edges(j), index(i)+1 <= j <= index(i+1).

Функции запроса графа

Если топология графа установлена, может быть необходимым запросить информацию относительно топологии. Эти функции даются ниже, и все они в данном случае - локальные вызовы.

```
MPI_GRAPHDIMS_GET(comm, nnodes, nedges)
IN  comm                коммуникатор группы, связанной с графом
OUT nnodes              количество узлов в графе
OUT nedges              количество ребер в графе
```

```
int MPI_Graphdims_get(MPI_Comm_comm, int *nnodes, int *nedges)
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)

INTEGER COMM, NNODES, NEDGES, IERROR
```

MPI_GRAPHDIMS_GET возвращает число узлов и число ребер в графе. Число узлов идентично размеру группы, связанному с comm. nnodes, и nedges используются, чтобы снабдить массивы надлежащего размера для index и edges, соответственно, в функции MPI_GRAPH_GET. MPI_GRAPHDIMS_GET возвратил бы nnodes = 4 и nedges = 6 в примере 1.5.

```
MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)
IN  comm              коммуникатор группы со структурой графа
IN  maxindex          длина вектора index в вызвавшей программе
IN  maxedges          длина вектора edges в вызвавшей программе
OUT index             массив чисел, определяющих узлы графа
OUT edges             массив чисел, определяющих узлы графа

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
                                    int *index, int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

MPI_GRAPH_GET возвращает index и edges, какими они были в MPI_GRAPH_CREATE. maxindex и maxedges, по крайней мере, такие же, как nnodes и nedges, соответственно, какие возвращаются функцией MPI_GRAPHDIMS_GET.

ПРИМЕР 1.6

Создание полного графа связей из 12 компьютеров.

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char **argv)
  { int size, rank, reord;
    MPI_Comm comm_gr;
 /* Инициализация библиотеки MPI */
    MPI_Init(&argc, &argv);
 /* Каждая ветвь определяет размер системы, если он больше 12,
  * то завершение */
    MPI_Comm_size(MPI_COMM_WORLD, size);
    if(size > 12)
      MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
 /* Каждая ветвь определяет свой номер */
    MPI_Comm_rank(MPI_COMM_WORLD, rank);
```

```
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф". */
    for(i = 0; i < size; i++)
     { index[i] = (size - 1)*(i + 1);
       v = 0;
       for(j = 0; j < size; j++)
        { if(i != j)
            edges[i * (size - 1) + v++] = j;
        }
     }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,
&comm_gr);

/* Продолжение программы */

    MPI_Finalize();
  }
```

1.4. Парные взаимодействия

Программные средства системных взаимодействий обеспечивают параллелизм вычислений и взаимодействие процессов. Основной механизм связи между процессами в MPI - передача данных между парой взаимодействующих процессов, одной посылающей стороны, другой - получающей.

MPI обеспечивает набор посылающих и получающих функций, которые допускают пересылку типов (`datatype`) данных, связанных (ассоциированных) тегом (`tag`). Тип содержания сообщения необходим для гетерогенной поддержки. Информация о типе необходима для того, чтобы могли быть сделаны правильные преобразования представления данных в разных архитектурах ЭВМ, поскольку данные могут быть посланы от процессов, находящихся в одной архитектуре ЭВМ, к процессам, находящимся в другой архитектуре ЭВМ. Тег допускает избирательность сообщений в приемном конце. На исходном процессе сообщения также обеспечивается избирательность сообщения. Модель обмена данными независима от машины и облегчает переносимое программирование.

Существует много вариантов парных операций. Варианты операций необходимы для оптимизации взаимодействий в каждом конкретном случае при описании параллельного алгоритма задачи. Здесь рассматриваются *блокированные*, *неблокированные* и *синхронные* взаимодействия, реализуемые соответствующими функциями. Имеются все передающие функции указанных выше типов. Принимающих функций только две: *блокированная* и *неблокированная*. Каждая принимающая функция из этих типов совместима со всеми передающими функциями, и наоборот. Блокированные и неблокированные взаимодействия являются асинхронными.

Блокированная передача/прием данных. Блокированная передающая функция не возвращает управление, пока данные сообщения не были безопасно сохранены в промежуточном буфере системы и посылающийся буфер не был снова свободен к доступу (чтению, записи данных). Блокированная принимающая функция возвращает управление только после того, как буфер приема данных получит соответствующее сообщение.

Неблокированная передача/прием данных. Неблокированные операции всегда имеют две части: функции передачи параметров системе, которые инициируют запрошенное действие, и функции "проверки на завершение", которые допускают, чтобы прикладная программа обнаружила, закончилось ли запрошенное действие. Неблокированная передающая функция указывает, что система может начинать копировать данные вне посылающегося буфера. После передачи параметров функции в систему функция возвращает управление. После этого передающий процесс не должен иметь доступа (по записи, а в данном случае - и по чтению) к посылающимуся буферу, т.е. система не гарантирует в этом случае сохранность посылаемых данных. Для проверки завершения рассматриваемой операции используются функции `MPI_WAIT` и `MPI_TEST`. Под завершением операции здесь понимается, что данные сообщения были безопасно сохранены в промежуточном буфере системы и посылающийся буфер снова свободен к доступу. Неблокированная принимающая функция указывает, что система может начинать писать данные в буфер приема. После передачи параметров функции в систему функция возвращает управление. После этого принимающий процесс не должен иметь доступа (по записи, а в данном случае - и по чтению) к буферу приема, т.е. система не гарантирует в этом случае сохранность принимаемых данных. Для проверки завершения рассматриваемой операции используются те же функции `MPI_WAIT` и `MPI_TEST`. Под завершением операции здесь понимается, что принимаемые данные находятся в буфере приема.

Синхронная блокированная/неблокированная передача данных. Синхронный способ взаимодействий имеет семантику реализации "rendezvous". Синхронная передача может быть начата только после того, как соответствующий приемник готов к приему посылаемых данных, т.е. запустилась соответствующая принимающая функция. Таким образом, завершение синхронных передающих операций не только указывает, что посылающийся буфер может теперь использоваться, но также и указывает, что приемник достиг некоторого пункта в его выполнении, а именно: он запустил выполнение соответствующей получающей функции. Синхронный способ обеспечивает семантику синхронной связи: связь не заканчивается с обоих концов перед обоюдным сближением процессов в связи. Для неблокированных операций функции `MPI_WAIT` и `MPI_TEST` проверяют, завершилась операция или нет.

1.4.1. Блокированные посылающая и получающая функции

Блокированная передача данных

```
MPI_SEND(buf, count, datatype, dest, tag, comm)

IN buf          адрес передаваемого буфера
IN count        количество передаваемых элементов
IN datatype     тип передаваемых элементов
IN dest         номер процесса, которому осуществляется
                передача данных
IN tag          тег сообщения
IN comm         имя переключателя каналов (communicator)

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

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

IN - обозначаются входные параметры, OUT - выходные. Длина сообщения определяется в терминах количества *элементов*, а не количества *байтов*. Это связано с тем, что одни и те же типы данных могут иметь разное количество байтов в представлении на ЭВМ с разными архитектурами. Основные типы данных MPI соответствуют основным типам данных базового языка. Возможные значения этого аргумента для Fortran и соответствующие типы Fortran внесены в список ниже.

MPI_datatype	Fortran_datatype
MPI_INTEGER MPI_REAL MPI_DOUBLE_PRECISION MPI_COMPLEX MPI_LOGICAL MPI_CHARACTER MPI_BYTE MPI_PACKED	INTEGER REAL DOUBLE PRECISION COMPLEX LOGICAL CHARACTER(1)

Возможные значения для этого аргумента для C и соответствующие типы C внесены в список ниже.

MPI_datatype	C datatype
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	

Типы MPI_BYTE и MPI_PACKED не соответствуют типам Fortran или C. Значение типа MPI_BYTE состоит из байта (8 двоичных цифр). Байт не интерпретируется и отличается от символа (знака).

Блокированный прием данных

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT buf      адрес буфера для приема данных
IN  count    максимальное количество принимаемых элементов
IN  datatype тип принимаемых элементов
IN  source   ранг (номер) передающего процесса
IN  tag      тег сообщения
IN  comm     имя переключателя каналов (communicator)
OUT status   статус (состояние) принимаемых данных

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

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
                          IERROR)

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

Длина сообщения определяется, как и для функции MPI_SEND, в терминах количества *элементов*, а не количества *байтов* и должна быть меньше или равняться длине буфера для получения данных. Получаемые данные сопровожда-

ются информацией об этих данных, которая записывается в status. На С статус полученных данных представляется структурой типа MPI_Status, которая содержит три поля с именами: MPI_SOURCE, MPI_TAG и MPI_ERROR, содержащими, соответственно, источник (source), тег (tag) и код ошибки полученного сообщения. (Доступ к этим данным: status.MPI_SOURCE, status.MPI_TAG, status. MPI_ERROR.) На Fortran(e) статус - это массив целых чисел длины MPI_STATUS_SIZE. Три константы: MPI_SOURCE, MPI_TAG и MPI_ERROR, которые хранят источник (source), тег (tag) и поле ошибки.

Статус имеет также аргумент, через который возвращается информация относительно длины полученного сообщения. Однако эта информация непосредственно недоступна как поле переменной статуса, а доступна с помощью запроса к функции MPI_GET_COUNT.

```
MPI_GET_COUNT(status, datatype, count)

IN  status              статус принятых данных
IN  datatype            тип принятых элементов
OUT count               количество принятых элементов

int MPI_Get_count(MPI_Status *status, MPI_Datatype
                                      datatype, int *count)

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

MPI_GET_COUNT берет как входной параметр status соответствующего MPI_RECV и вычисляет число полученных элементов. Количество полученных элементов возвращено в переменной count. Аргумент datatype должен быть таким же, как и аргумент в получающей функции MPI_RECV, к статусу которой осуществляется доступ.

Соответствие типов для передаваемых и получаемых данных должно строго выполняться. Типы между посылающей и получающей функциями соответствуют, если они обе определяют идентичные имена типа, т.е. MPI_INT соответствует MPI_INT, MPI_DOUBLE соответствует MPI_ DOUBLE, и так далее.

ПРИМЕР 1.7

Фрагмент программы для источника и приемника.

```
MPI_Comm_rank(comm, rank);
if(rank == 0)
   MPI_Send(a, 10, MPI_FLOAT, 1, tag, comm);
elseif(rank == 1)
   MPI_Recv(b, 15, MPI_FLOAT, 0, tag, comm, status);
```

MPI не делает преобразование типов данных, например, округляя `double` (вещественный) к `int` (целый) числу, но делает преобразование представления данных, изменяя двоичное представление значения данных в рамках одного и того же типа, например, изменяя значение байта или преобразуя 32-разрядное число с плавающей запятой к 64-разрядному числу с плавающей запятой.

1.4.2. Объединенная функция передачи/приема данных

MPI имеет специальную функцию (`MPI_SENDRECV`), объединяющую в одном запросе посылающее и получающее действия: посылку одного сообщения по назначению и получение другого сообщения из источника. Источник и назначение, возможно, те же самые. Такая функция полезна для моделей связи, где каждый процесс и посылает, и получает сообщения. Один пример - обмен данными между двумя процессами. Другой пример - смещение данных вдоль цепи процессов. Безопасная программа, которая осуществляет такое смещение, должна использовать одинаковый порядок связи между процессами. `MPI_SENDRECV` может использоваться вместе с функциями, описанными выше. Имеется совместимость между `MPI_SENDRECV` и функциями посылки и получения сообщений, описанными выше. Сообщение, посланное `MPI_SENDRECV`, может быть получено обычной функцией приема сообщений, и, наоборот, `MPI_SENDRECV` может получать сообщение, посланное обычной функцией передачи сообщений.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                  recvcount, recvtype, source, recvtag, comm,
                  status)
IN  sendbuf      адрес посылаемого буфера
IN  sendcount    количество посылаемых элементов
IN  sendtype     тип элементов в посылаемом буфере
IN  dest         ранг (номер) процесса, которому осуществляется
                 передача
IN  sendtag      тег посылаемого сообщения
OUT recvbuf      адрес буфера для приема данных
IN  recvcount    максимальное количество принимаемых элементов
IN  recvtype     тип принимаемых элементов
IN  source       ранг (номер) передающего процесса
IN  recvtag      тег принимаемых данных
IN  comm         имя переключателя каналов (communicator)
OUT status       статус полученного сообщения

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)
```

```
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
INTEGER SOURCE, RECV_TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_SENDRECV выполняет блокированные посылку и получение данных. Обе функции: посылающая и получающая - используют тот же самый переключатель каналов. Посылающийся буфер и буфер для приема не должны пересекаться и могут иметь различные длины. Следующая функция аналогична предыдущей, но у нее посылаемый буфер совпадает с буфером для получения данных.

```
MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag,
                source, recvtag, comm, status)
                для приема данных
IN count        количество элементов в посылаемом буфере
                и буфере для приема
IN datatype     тип элементов в посылаемом буфере и буфере
                для приема
IN dest         ранг (номер) процесса, которому осуществляется
                передача
IN sendtag      тег посылаемого сообщения
IN source       ранг (номер) передающего процесса
IN recvtag      тег принимаемых данных
IN comm         имя переключателя каналов (communicator)
OUT status      статус полученного сообщения

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

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG,
                     SOURCE, RECVTAG, COMM, STATUS, IERROR)

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

MPI_SENDRECV_REPLACE выполняется с блокированием посылки и получения. Используется тот же самый буфер как для посылающей, так и для получающей функции. Посланное сообщение затем заменяется полученным сообщением.

ПРИМЕР 1.8

Имеется N компьютеров, объединенных в топологию "кольцо" с именем comm. В определенный момент все компьютеры пересылают свои данные соседним компьютерам с большими номерами (фрагмент программы).

```
 . . . . .
   MPI_Comm_rank(comm, rank);     /* rank - номер компьютера */
   MPI_Comm_rank(comm, size);     /* size - размер системы */

. . . . . .

int MPI_Sendrecv(sbuf,scount, stype, MPI_INT, (rank+1)% size,
        stag, rbuf, rcount, MPI_INT, (rank+size-1)%size, rtag,
        comm, &status)

. . . . . .
```

Аналогично можно использовать и функцию MPI_Sendrecv_replace( ).

1.4.3. Неблокированные посылающая и получающая функции

Можно улучшить выполнение многих программ, выполняя обмен данными и вычисления с перекрытием, т.е. параллельно. Это особенно хорошо реализуется на системах, где связь может быть выполнена автономно от работы компьютера. Многоподпроцессный режим - один из механизмов для достижения такого перекрытия. В то время как один из подпроцессов блокирован, ожидая завершения связи, другой подпроцесс может выполняться на том же самом процессоре. Этот механизм эффективен, если система поддерживает многоподпроцессный режим.

Неблокированные функции

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN  buf             адрес посылаемого буфера
IN  count           количество элементов в посылаемом буфере
IN  datatype        тип элементов в передаваемом буфере
IN  dest            номер принимающего процессора
IN  tag             тег передаваемых данных
IN  comm            имя коммуникатора связи (communicator)
OUT request         имя (заголовка) запроса

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

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
    IERROR)

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

Неблокированая передача данных инициализирует посылающее действие, но не заканчивает его. Функция возвратит управление прежде, чем сообщение скопировано вне посылающегося буфера. Неблокированная посылающая функция указывает, что система может начинать копировать данные вне посылающегося буфера. Посылающий процесс не должен иметь доступа к посылаемому буферу после того, как неблокированное посылающее действие инициировано, до тех пор, пока функция завершения не возвратит управление.

```
MPI_IRECV(buf, count, datatype, source, tag, comm, request)

OUT buf          адрес буфера приема данных
IN  count        максимальное количество принимаемых элементов
IN  datatype     тип принимаемых элементов
IN  source       номер передающего процесса
IN  tag          тег сообщения
IN  comm         имя коммуникатора связи (communicator)
OUT request      имя (заголовка) запроса

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

MPI_IRECV(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM,REQUEST,IERROR)

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

Неблокированый прием данных инициализирует получающее действие, но не заканчивает его. Функция возвратит управление прежде, чем сообщение записано в буфер приема данных. Неблокированная получающая функция указывает, что система может начинать писать данные в буфер приема данных. Приемник не должен иметь доступа к буферу приема после того, как неблокированное получающее действие инициировано, до тех пор, пока функция завершения не возвратит управление.

Эти обе функции размещают данные в системном буфере и возвращают заголовок этого запроса в `request`. `request` используется, чтобы опросить состояние связи.

Функции завершения неблокированных операций

Чтобы закончить неблокированные посылку и получение данных, используются *завершающие* функции `MPI_WAIT` и `MPI_TEST`. Завершение посылающего процесса указывает, что он теперь свободен к доступу посылающегося буфера. Завершение получающего процесса указывает, что буфер приема данных содержит сообщение, приемник свободен к его доступу.

```
MPI_WAIT(request, status)
INOUT request        имя запроса
OUT   status         статус оъекта

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS SIZE), IERROR
```

Запрос к MPI_WAIT возвращает управление после того, как операция, идентифицированная request, выполнилась. Это блокированная функция. Если объект системы, указанный request, был первоначально создан неблокированными посылающей или получающей функциями, то этот объект освобождается функцией MPI_WAIT, и request устанавливается в MPI_REQUEST_NULL. *Статус* объекта содержит информацию относительно выполненной операции.

```
MPI_TEST(request, flag, status)
INOUT request      имя запроса
OUT   flag         true, если операция выполнилась, иначе false
OUT   status       статус объекта

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

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS SIZE), IERROR
```

Запрос к MPI_TEST возвращает flag = true, если операция, идентифицированная request, выполнилась. В этом случае *статус* состояния содержит информацию относительно законченной операции. Если объект системы, указанный request, был первоначально создан неблокированными посылающей или получающей функциями, то этот объект освобождается функцией MPI_TEST, и request устанавливается в MPI_REQUEST_NULL. Запрос возвращает flag = false, если операция не выполнилась. В этом случае значение *статуса* состояния не определено. Это неблокированная функция.

ПРИМЕР 1.9

Следующий пример показывает работу многих производителей с единственным потребителем. Последний в группе процесс потребляет сообщения, посланные другими процессами.

```
 ...
   typedef struct
     {
       char data[MAXSIZE];
       int datasize;
       MPI_Request req;
     } Buffer;
   Buffer buffer[];
   MPI_Status status;
...
   /* Каждая ветвь определяет количество компьютеров в системе и свой
ранг */
   MPI_Comm_rank(comm, &rank);
   MPI_Comm_size(comm, &size);
   /* Производители */
   if(rank != size-1)
     {
    /* инициализация одного буфера */
      buffer = (Buffer *)malloc(sizeof(Buffer));
    /* главный цикл */
      while(1)
        {
       /* производство файлов данных и возврат
        * количества байтов, сохраненных в буфере */
          produce(buffer->data, &buffer->datasize);
       /* передача данных */
     MPI_Send(buffer->data, buffer->datasize, MPI_CHAR, size-1, tag,
comm);
        }
     }
   /* rank == size-1; потребитель */
   else
     {
    /* инициализация одного буфера */
      buffer = (Buffer *)malloc(sizeof(Buffer)*(size-1));
      for(i=0; i ! size-1; i++)
    /* инициализация приема от каждого производителя */
      MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
                                          &(buffer[i].req));
    /* Главный цикл */
```

```
        for(i = 0;  ; i = (i+1)%(size-1))
          { MPI_Wait(&(buffer[i].req), &status);
         /* Определение количества реально принятых байтов */
            MPI_Get_count(&status, MPI_CHAR,
                                      &(buffer[i].datasize));
         /* Буфер приема элементов данных */
            consume(buffer[i].data, buffer[i].datasize);
         /* инициализация нового приема */
            MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i,tag,
                                         comm,&(buffer[i].req));
          }
      }
```

Каждый производитель "крутится" в бесконечном цикле, где повторяет производство одного сообщения и передает его. Потребитель обслуживает каждого производителя в цикле, и принимает сообщения.

Пример накладывает строгую циклическую дисциплину, так как потребитель получает одно сообщение от каждого производителя, по очереди. В некоторых случаях предпочтительно использовать дисциплину "first-come-first-served". Это достигается за счет использования `MPI_TEST` вместо `MPI_WAIT`, как показано ниже. Заметьте, что MPI может только предлагать дисциплину "первый пришел первый обслужился", так как сообщения не обязательно находятся в том порядке, в каком они были посланы.

ПРИМЕР 1.10

Модифицированный многократный производитель с единственным потребителем, использующий проверочные вызовы.

```
...
   typedef struct
     {
       char data[MAXSIZE];
       int datasize;
       MPI_Request req;
     } Buffer;
   Buffer buffer[];
   MPI_Status status;
 ...
/* Каждая ветвь определяет количество компьютеров в системе и свой ранг */
   MPI_Comm_rank(comm, &rank);
   MPI_Comm_size(comm, &size);
/* Производитель */
   if(rank != size-1)
     { buffer = (Buffer *)malloc(sizeof(Buffer));
    /* Главный цикл */
```

```
        while(1)
          { produce(buffer->data, &buffer->datasize);
            MPI_Send(buffer->data, buffer->datasize, MPI_CHAR, size-1,
            tag, comm);
          }
      }
/* rank == size-1; потребитель */
   else
     { buffer = (Buffer *)malloc(sizeof(Buffer)*(size-1));
       for(i = 0; i < size-1; i++)
        MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
                                                &buffer[i].req);
       i = 0;
       while(1)      /* главный цикл */
         { for(flag = 0; !flag; i = (i+1)%(size-1))
        /* проверка для завершения приема */
            MPI_Test(&(buffer[i].req), &flag, &status);
            MPI_Get_count(&status, MPI_CHAR, &buffer[i].datasize);
            consume(buffer[i].data, buffer[i].datasize);
            MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
                                                  &buffer[i].req);
         }
      }
```

Если здесь нет сообщения, ожидаемого от производителя, тогда потребляющий процесс перейдет к предыдущему производителю.

1.4.4. Синхронные посылающие функции

```
MPI_SSEND(buf, count, datatype, dest, tag, comm)

IN buf                  адрес передаваемого буфера
IN count                количество передаваемых элементов
IN datatype             тип передаваемых элементов
IN dest                 ранг приемника
IN tag                  тег сообщения
IN comm                 коммуникатор (communicator)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
                                              int tag, MPI_Comm comm)

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IER
```

`MPI_SSEND` - блокированная, синхронная функция передачи данных.

```
MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

IN  buf                   адрес передаваемого буфера
IN  count                 количество передаваемых элементов
IN  datatype              тип передаваемых элементов
IN  dest                  ранг приемника
IN  tag                   тег сообщения
```

```
IN  comm                коммуникатор (communicator)
OUT request             заголовок запроса

int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
                        int tag, MPI_Comm comm, MPI_Request *request)

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type. BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

MPI_ISSEND - неблокированная, синхронная функция передачи данных. Если соответствующей принимающей функцией является неблокированная принимающая функция MPI_IRECV, то передающая функция MPI_ISSEND синхронизуется с переданными в систему параметрами соответствующей неблокированной принимающей функции. А функции Wait и Test со стороны передающей функции только проверяют наличие этих выставленных параметров со стороны неблокированной принимающей функции.

При синхронных взаимодействиях пересылаемый буфер передается в принимаемый буфер "напрямую" (память-память), минуя сохранение в промежуточных буферах.

1.5. Коллективные взаимодействия

Коллективная связь обеспечивает обмен данными среди всех процессов в группе, указанной аргументом коммуникатора. MPI обеспечивает следующие коллективные функции связи:

• синхронизация (barrier) - синхронизует все процессы группы;

• глобальные функции связи, которые иллюстрируются на рис. 1.3. Они включают:

– передачу данных (broadcast) от одного процесса группы ко всем процессам группы;

– сбор данных (gather) от всех процессов группы к одному процессу этой группы;

– разброс данных (scatter) от одного процесса группы ко всем процессам группы;

– сбор данных в цикле по всем процессам группы; все элементы группы получают результат от всех;

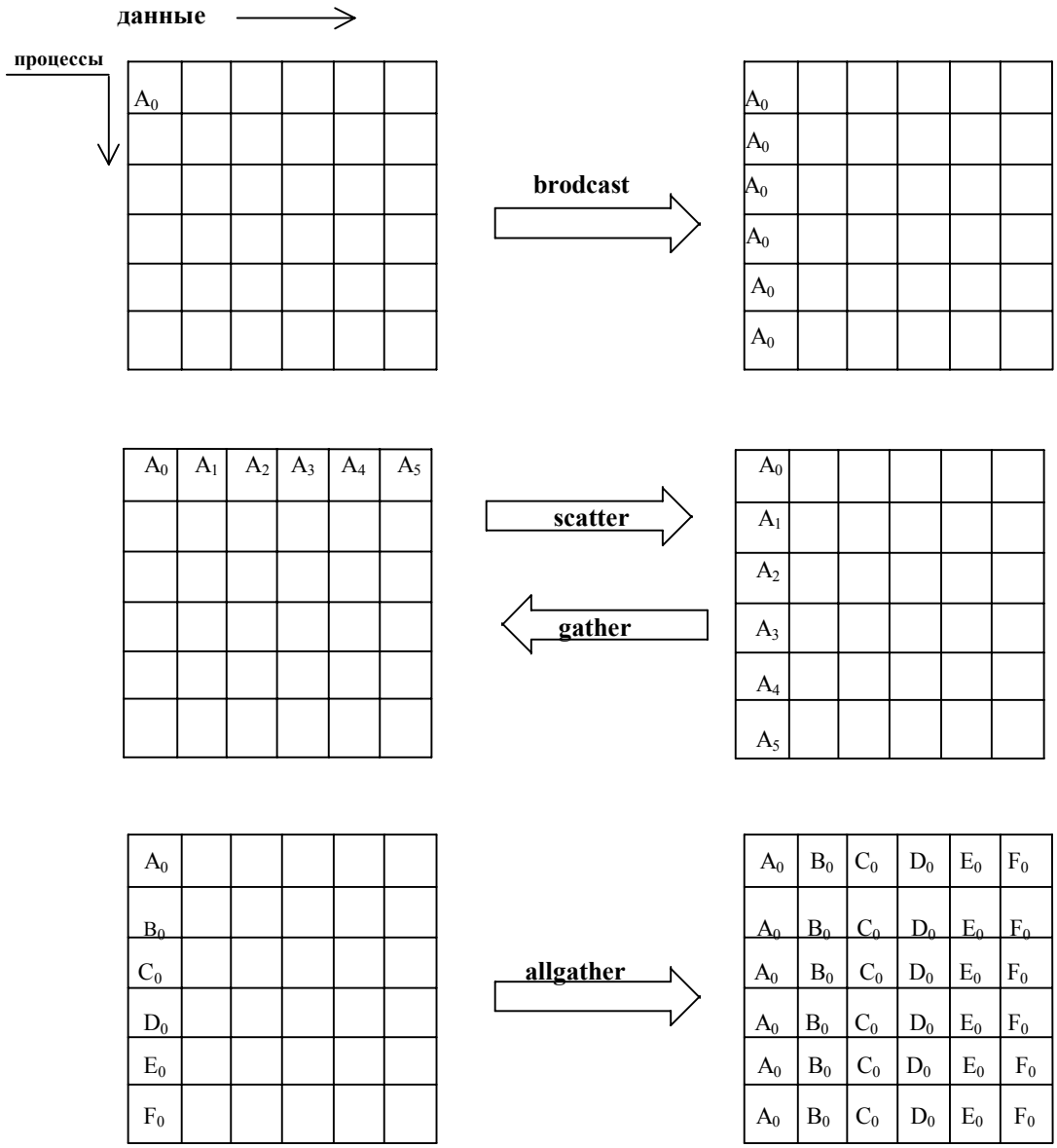
– разброс/сбор (scatter/gather) данных от всех элементов ко всем элементам группы (функция названа полным обменом, или "все ко всем");

• глобальные операции редуцирования типа sum, max, min или определяемые пользователем функции. Они включают:

– редуцирование, где результат возвращается всем процессам группы, и версию, где результат возвращается только одному процессу группы (п. 6.10);

– объединенное редуцирование и операцию `scatter` (разброс). Развертка по всем элементам группы (также назван префикс).

Рисунок 1.3 дает иллюстрированное представление функций глобальной связи. Все эти функции (исключая `broadcast`) имеют два варианта: простой вариант, где все передаваемые данные имеют один и тот же размер, и векторный ("vector") вариант, где каждый набор данных может иметь различный размер. Кроме того, в простом варианте данные, многократно передаваемые одним и тем же процессом или получаемые одним и тем же процессом, являются прилегающими (соприкасающимися) в памяти; векторный вариант допускает выбор различных наборов данных из нескольких несмежных участков памяти.



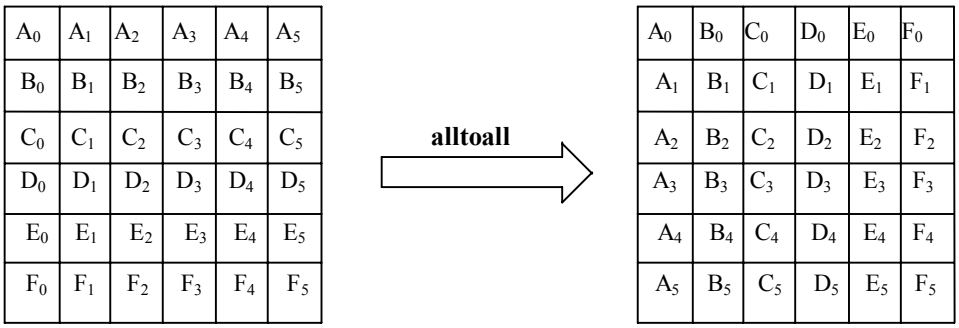


Рис. 1.3. Иллюстрация коллективных передающих функций для группы из шести процессов. В каждой клетке представлены локальные данные в одном процессе. Например, в `broadcast` передает данные A_0 только первый процесс, а другие процессы принимают эти данные.

Некоторые из этих функций, типа `broadcast` или `gather`, имеют единственный передающий процесс или единственный процесс получения. Такой процесс назван корнем (`root`). Функции глобальной связи в основном имеют три модели:

1. Корень посылает данные ко всем процессам (включая корень): `broadcast` и `scatter`.

2. Корень получает данные из всех процессов (включая корень): `gather`.

3. Каждый процесс связывается с каждым процессом (включая текущий): `allgather` и `all-to-all`.

Коллективные функции сделаны более ограниченными, чем point-to-point операции. В отличие от point-to-point операций, количество посланных данных в этих функциях должно быть точно согласовано с количеством данных, указанных приемником. Коллективные функции имеют только блокированные версии и не используют аргумент тега. Аргумент типа данных должен быть одним и тем же во всех процессах, участвующих во взаимодействии. Внутри каждой области связи коллективные запросы строго согласованы согласно порядку выполнения. В коллективном запросе к `MPI_BCAST` должны участвовать все процессы, объединенные коммуникатором. Коллективные функции не согласуются с функциями парных взаимодействий.

1.5.1. Синхронизация

```
MPI_BARRIER(comm)
IN comm            коммуникатор

int MPI_Barrier(MPI_Comm comm)

MPI_BARRIER(COMM, IERROR)
```

```
INTEGER COMM, IERROR
```

MPI_BARRIER блокирует вызывающий оператор, пока все элементы группы не вызовут его. В любом процессе запрос возвращается только после того, как все элементы группы вошли в запрос.

1.5.2. Трансляционный обмен данными

```
MPI_BCAST(buffer, count, datatype, root, comm)

INOUT buffer                адрес буфера
IN    count                 количество элементов в буфере
IN    datatype              тип данных
IN    root                  ранг корневого процесса
IN    comm                  коммуникатор

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
                                       int root, MPI_Comm comm)
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

MPI_BCAST передает сообщение из процесса с рангом root ко всем процессам группы. Аргументы корня и на всех других процессах должены иметь идентичные значения, и comm должна представлять ту же самую область связи. После возвращения содержимое буфера buffer из корня копируется ко всем процессам в буфер buffer.

ПРИМЕР 1.11

Передается 100 элементов из процесса 0 ко всем процессам группы.

```
MPI_Comm comm;
int array[100];
int root=0;
 ...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

1.5.3. Сбор данных

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                 recvtype, root, comm)

IN  sendbuf        адрес передаваемого буфера
IN  sendcount      количество передаваемых элементов
IN  sendtype       тип передаваемых элементов
OUT recvbuf        адрес буфера приема
IN  recvcount      количество принимаемых элементов в каждом процессе
IN  recvtype       тип принимаемых данных
IN  root           ранг принимающего процесса
IN  comm           коммуникатор

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
```

```
                int root, MPI_Comm comm)

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

Каждый процесс (включая процесс корня) посылает содержимое его посылаемого буфера к процессу корня. Процесс корня получает сообщения и хранит их в порядке рангов посылающих процессов. Результат выглядит так, как будто каждый из `n` процессов в группе (включая процесс корня) выполнил запрос к `MPI_SEND(sendbuf,sendcount,sendtype,root, ...)` и корень выполнил `n` запросов к `MPI_RECV(recvbuf+i*recvcount,recvcount, recvtype,i,...)`. Приемный буфер игнорируется для всех процессов, не равных корню. Аргумент `recvcount` в корне указывает число элементов, которые получает корень от каждого процесса, а не общее число элементов, которые он получает всего.

ПРИМЕР 1.12

Прием корнем 100 элементов от каждого процесса группы (рис. 1.4).

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
 ...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

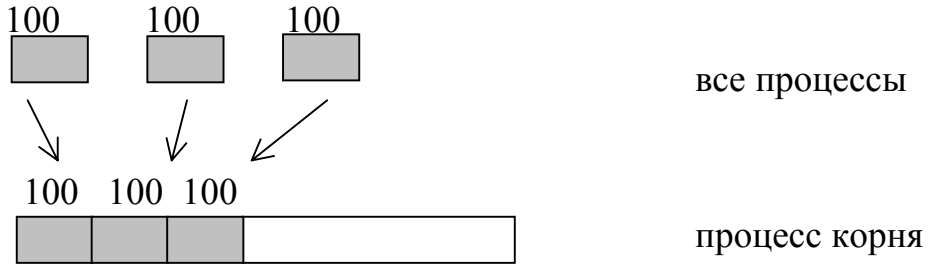


Рис. 1.4. Процесс корня принимает 100 элементов от каждого процесса в группе

1.5.4. Сбор данных (векторный вариант)

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,displs,
                                                  recvtype, root, comm)

IN  sendbuf             адрес передаваемого буфера
IN  sendcount           количество передаваемых элементов
IN  sendtype            тип передаваемых элементов
OUT recvbuf             адрес буфера приема
```

```
IN  recvcounts          целочисленный массив, указывающий количество
                        принимаемых элементов в каждом процессе
IN  displs              целочисленный массив смещений принятых
                        пакетов данных относительно друг друга
IN  recvtype            тип принимаемых данных
IN  root                ранг принимающего процесса
IN  comm                коммуникатор (communicator)

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

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

MPI_GATHERV расширяет функциональные возможности MPI_GATHER, допуская изменяющийся counts данных из каждого процесса, так как recvcounts теперь массив. Она также допускает большее количество гибкости относительно того, где данные размещаются на корне, обеспечивая новый аргумент, displs. Данные, посланные из процесса j, размещаются в j-м блоке в буфере приема recvbuf на процессе корня. Блок j-й в буфере recvbuf начинается в смещении от начала предыдущего пакета в displs[j] элементов (в терминах recvtype). Буфер приема игнорируется для всех процессов, не принадлежащих корню. Все аргументы в функции на корне процесса значимы, в то время как на других процессах значимы только аргументы sendbuf, sendcount, sendtype, root и comm. Аргументы должны иметь идентичные значения на всех процессах, и comm должна представлять ту же самую область связи.

ПРИМЕР 1.13

Каждый процесс посылают 100 элементов. В принимающем процессе пакеты (в 100 элементов) нужно разместить на растоянии с некоторым шагом. Используется MPI_GATHERV и аргумент displs, чтобы достичь этого эффекта. Допустим шаг stride ≥ 100 (рис. 1.5).

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
 ...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i)
```

```
   { displs[i] = i*stride;
     rcounts[i] = 100;
   }
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs,
            MPI_INT, root, comm);
```

Программа ошибочна, если `-100 < stride < 100`.

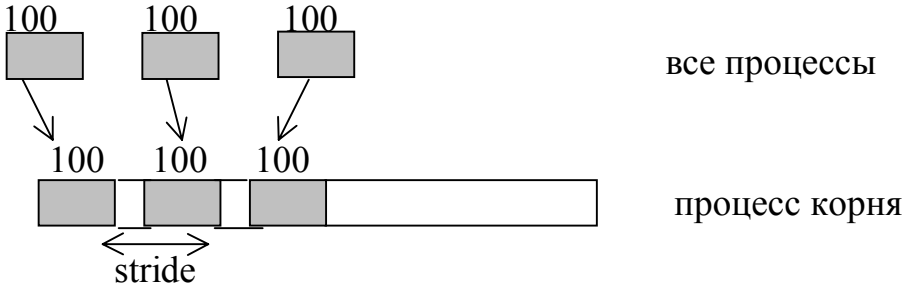


Рис. 1.5. Процесс корня принимает 100 элементов от каждого процесса в группе (каждый набор размещается с шагом `stride` элементов друг от друга)

ПРИМЕР 1.14

Процесс `i` посылает `(100-i)` элементов из `i`-й колонки из массива 100*150. Пакеты поступают в буфер с большим шагом (рис. 1.6).

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs, i, *rcounts;
 ...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i)
   { displs[i] = i*stride;
     rcounts[i] = 100-i;
     scounts[i] = 100-myrank;
   }
/* sptr - адрес начала колонки "myrank" */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, scounts, MPI_INT, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

Заметим, что различное количество данных получено от каждого процесса.

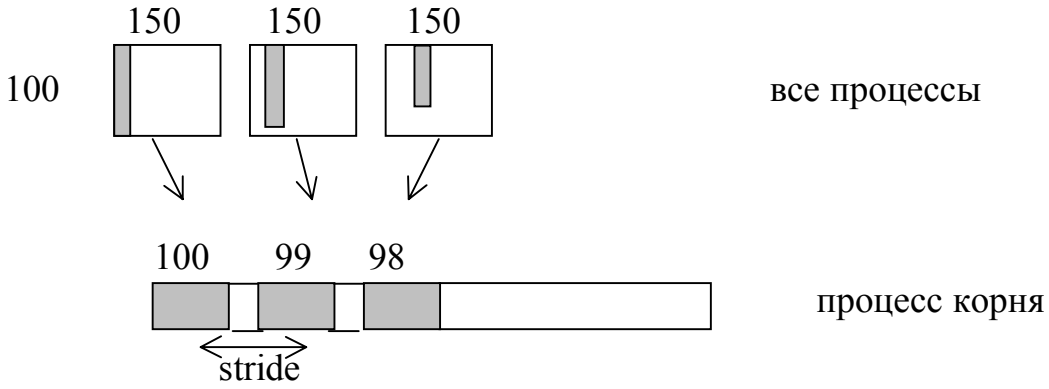


Рис. 1.6. Процесс корня принимает `100-i` элементов из колонки `i` массива 100*150 (каждый набор размещается с шагом `stride` элементов друг от друга)

ПРИМЕР 1.15

Тот же самый, как пример 1.13 в посылающей стороне, но в получающей стороне делается разный шаг между полученными блоками (рис. 1.7).

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs, i, *rcounts, offset;
 ...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
 ...
/* Устанослен stride[i] от i = 0 до gsize-1 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for(i = 0; i < gsize; ++i)
   { displs[i] = offset;
     offset += stride[i];
     rcounts[i] = 100-i;
     scounts[i] = 100-myrank;
   }
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, scounts, MPI_INT, rbuf, rcounts, displs, MPI_INT,
root, comm);
```

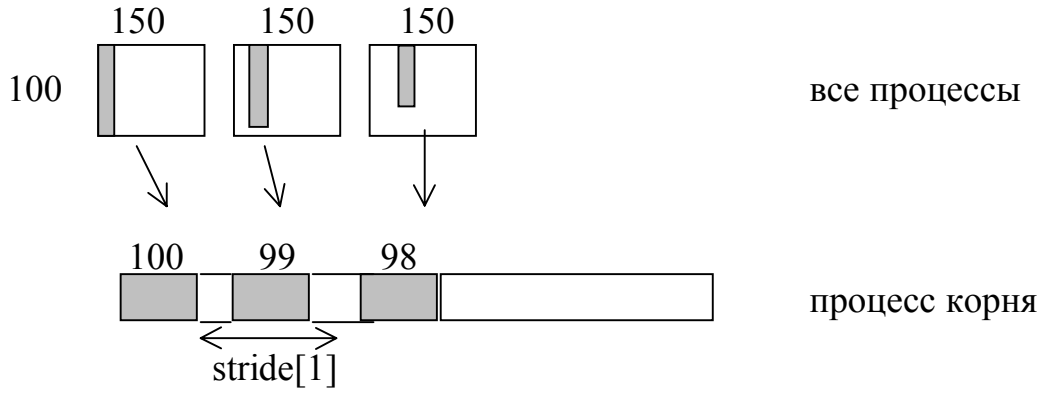


Рис. 1.7. Процесс корня принимает `100-i` элементов из колонки `i` массива 100*150 (каждый набор размещается с изменяющимся шагом `stride[1]` элементов)

1.5.5. Разброс данных

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                          recvtype, root, comm)

IN  sendbuf         адрес передаваемого буфера
IN  sendcount       количество передаваемых элементов каждому процессу
IN  sendtype        тип передаваемых данных
OUT recvbuf         адрес буфера приема
IN  recvcount       количество принимаемых элементов
IN  recvtype        тип принимаемых данных
IN  root            ранг передающего процесса
IN  comm            коммуникатор (communicator)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
            RECVTYPE, ROOT, COMM, IERROR)

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

`MPI_SCATTER` - обратная операция к `MPI_GATHER`. Результат выглядит так, как будто корень выполнил `n` посылающих операций `MPI_Send (sendbuf+i *sendcount *extent(sendtype), sendcount,sendtype,i,...)`, `i=0` до `n-1`. И каждый принимающий процесс выполнил функцию `MPI_Recv(recvbuf, recvcount, recvtype, root,...)`. Аргументы `sendcount` и `sendtype` в корне должны быть равны аргументам `recvcount` и `recvtype` во всех процессах. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждым процессом и корнем. Все аргументы в функции значимы на корневом процессе, в то время как на других процессах значимы только аргументы `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`.

ПРИМЕР 1.16

`MPI_Scatter` передает 100 элементов из корня каждому процессу в группе (рис. 1.8).

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

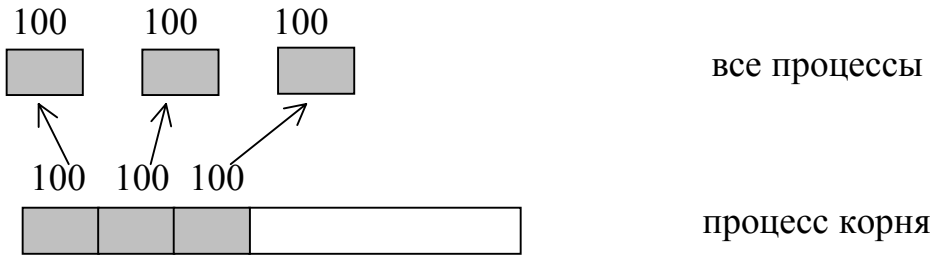


Рис. 1.8. Процесс корня передает наборы 100 элементов каждому процессу в группе

1.5.6. Разброс данных (векторный вариант)

```
MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf,
                            recvcount, recvtype, root, comm)

IN  sendbuf             адрес передаваемого буфера
IN  sendcounts          массив, содержащий количество передаваемых
                        элементов каждому процессу
IN  displs              целочисленный массив смещений пакетов
                        относительно друг друга
IN  sendtype            тип передаваемых данных
OUT recvbuf             адрес буфера приема
IN  recvcount           количество принимаемых элементов
IN  recvtype            тип принимаемых данных
IN  root                ранг передающего процесса
IN  comm                коммуникатор (communicator)

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype, void* recvbuf, int
                 recvcount, MPI_Datatype recvtype, int root,
                 MPI_Comm comm)

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
                    RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*),DISPLS(*),SENDTYPE,RECVCOUNT,RECVTYPE,ROOT
INTEGER COMM,IERROR
```

MPI_SCATTERV - обратная операция к MPI_GATHERV. MPI_SCATTERV расширяет функциональные возможности MPI_SCATTER, позволяя посылать каждому процессу изменяющееся количество данных, так как sendcounts теперь массив. Она также допускает больше возможностей относительно того, где данные взяты из корня, обеспечивая новый аргумент displs.

Результат выглядит так, как будто корень выполнил n посылающих операций MPI_Send(sendbuf+displs[i] *extent(sendtype), sendcounts[i], sendtype, i,...), i=0 до n-1, и каждый процесс выполнил функцию приема MPI_Recv(recvbuf,recvcount, recvtype,root,...). Все аргументы в функции значимы на корневом процессе, в то время как на других процессах значимы только аргументы recvbuf, recvcount, recvtype, root, comm.

ПРИМЕР 1.17

Обратный примеру 1.12. Корневой процесс разбрасывает по 100 элементов к другим процессам, но наборы по 100 элементов расположены в буфере посылок с некоторым шагом stride друг от друга, где шаг stride > 100. Это требует использования MPI_SCATTERV (рис. 1.9).

```
    MPI_Comm comm;
    int gsize,*sendbuf;
    int root, rbuf[100], i, *displs, *scounts;
     ...
    MPI_Comm_size(comm, &gsize);
    sendbuf = (int *)malloc(gsize*stride*sizeof(int));
     ...
    displs = (int *)malloc(gsize*sizeof(int));
    scounts = (int *)malloc(gsize*sizeof(int));
    for(i = 0; i!gsize; ++i)
       { displs[i] = i*stride;
         scounts[i] = 100;
       }
    MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
root, comm);
```

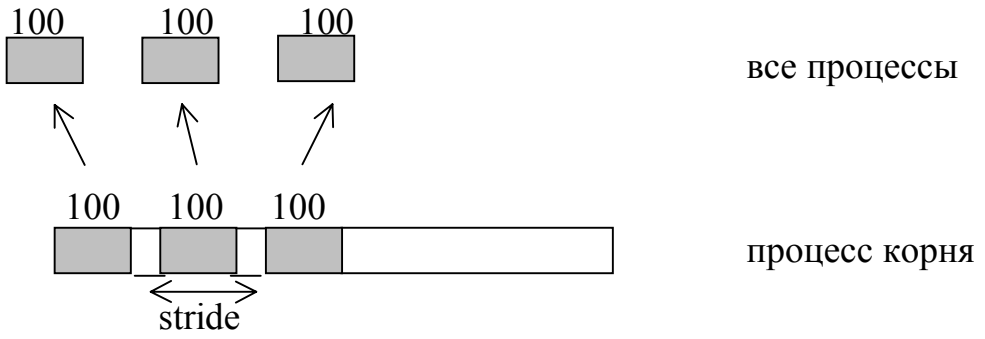


Рис. 1.9. Корневой процесс разбрасывает по 100 элементов, смещенных друг от друга с некоторым шагом stride > 100 (в терминах элементов)

ПРИМЕР 1.18

Обратный примеру 1.14. Передаваемые блоки в буфере корня расположены с изменяющимся шагом между блоками, получающая сторона принимает `100-i` элементов в `i`-ю колонку массива 100*150 в процессе `i` (рис. 1.10).

```
    MPI_Comm comm;
    int gsize,recvarray[100][150],*rptr;
    int root, *sendbuf, myrank, bufsize, *stride;
    MPI_Datatype rtype;
    int i, *displs, *scounts, offset;
     ...
    MPI_Comm_size(comm, &gsize);
    MPI_Comm_rank(comm, &myrank);
    stride = (int *)malloc(gsize*sizeof(int));
     ...
    /* stride[i] для i=0 до gsize-1 установлены */
     ...
    displs = (int *)malloc(gsize*sizeof(int));
    scounts = (int *)malloc(gsize*sizeof(int));
    offset = 0;
    for(i = 0; i < gsize; ++i)
       { displs[i] = offset;
         offset += stride[i];
         scounts[i] = 100-i;
         rcounts[i] = 100-i;
       }
    MPI_Type_commit(&rtype);
    rptr = &recvarray[0][myrank];
    MPI_Scatterv(sendbuf,  scounts,  displs,  MPI_INT,  rptr,  rcounts,
MPI_INT, root, comm);
```

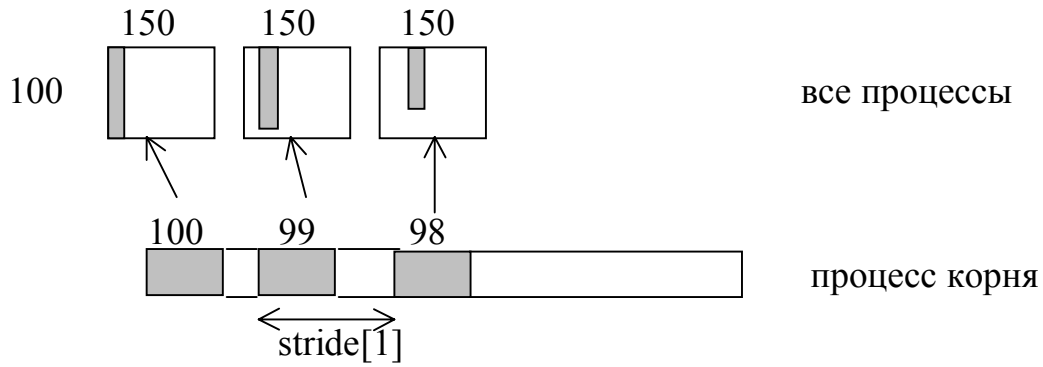


Рис. 1.10. Корень разбрасывает блоки по `100-i` элементов в колонки `i` массива 100*150 (в корне блоки расположены в буфере с шагом `stride[i]` элементов)

1.5.7. Сбор данных у всех процессов

```
MPI_ALLGATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,
                                                              comm)

IN  sendbuf             адрес передаваемого буфера
IN  sendcount           количество передаваемых элементов
IN  sendtype            тип передаваемых данных
OUT recvbuf             адрес буфера приема
IN  recvcount           количество принимаемых элементов от процессов
IN  recvtype            тип принимаемых данных
IN  comm                коммуникатор (communicator)

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

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                        RECVTYPE, COMM, IERROR)

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

`MPI_ALLGATHER` аналогична операции `MPI_GATHER`, за исключением того, что все процессы получают результат от всех процессов вместо только одного корня. `j`-й блок данных, посланных из каждого процесса, получен каждым процессом и размещается в `j`-м блоке буфера `recvbuf`.

Аргументы `sendcount` и `sendtype` в процессе должны быть равны во всех процессах.

Результат запроса к `MPI_ALLGATHER(...)` выглядит так, как будто все процессы выполнили `n` запросов к `MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`, для `root=0,...,n-1`.

ПРИМЕР 1.19

Версия примера 1.11. Используется `MPI_ALLGATHER`, принимается по 100 элементов из каждого процесса в группе в каждом процессе.

```
MPI_Comm comm;
int gsize, sendarray[100];
int *rbuf;
 ...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

После запроса каждый процесс имеет конкатенацию наборов данных в количестве размера группы.

1.5.8. Сбор данных у всех процессов (векторный вариант)

```
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                              displs, recvtype, comm)

IN  sendbuf       адрес передаваемого буфера
IN  sendcount     количество передаваемых элементов
IN  sendtype      тип передаваемых данных
OUT recvbuf       адрес буфера приема
IN  recvcounts    массив, указывающий количество принимаемых
                  элементов от процессов
IN  displs        целочисленный массив смещений пакетов данных друг
                  относительно друга
IN  recvtype      тип принимаемых данных
IN  comm          коммуникатор (communicator)

int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype
        sendtype, void* recvbuf, int *recvcounts, int *displs,
                          MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
                                     DISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE,
COMM, IERROR
```

MPI_ALLGATHERV аналогична MPI_GATHERV, за исключением того, что все процессы получают результат вместо только одного кореня. j-й блок данных посылается из каждого процесса, получается каждым процессом и размещается в j-м блоке буфера recvbuf. Не все эти блоки могут иметь тот же самый размер. Аргументы sendcount и sendtype в процессе j должны быть равны аргументам recvcounts[j] и recvtype в любом другом процессе. Результат выглядит так, как будто все процессы выполнили запросы к MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm), для root=0, ..., n-1.

1.5.9. Разброс/сбор - все ко всем

```
MPI_ALLTOALL(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                                            recvtype,comm)

IN  sendbuf       адрес передаваемого буфера
IN  sendcount     количество передаваемых элементов к каждому процессу
IN  sendtype      тип передаваемых данных
OUT recvbuf       адрес буфера приема
IN  recvcount     количество принимаемых элементов от каждого процесса
IN  recvtype      тип принимаемых данных
IN  comm          коммуникатор (communicator)

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

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                        RECVTYPE, COMM, IERROR)
```

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

MPI_ALLTOALL - расширение MPI_ALLGATHER для случая, когда каждый процесс посылает различные данные на каждый из приемников. j-й блок, посланный из процесса i, получен процессом j и размещен в i-м блоке буфера recvbuf. Аргументы sendcount и sendtype в процессе должны быть равны аргументам recvcount и recvtype в любом другом процессе. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждой парой процессов. Результат выглядит так, как будто каждый процесс выполнил посылающий к каждому процессу (включая самого себя) запрос MPI_Send (sendbuf + i*sendcount *extent (sendtype) sendcount, sendtype, i, ...) и получает данные из каждого другого процесса запросом к MPI_Recv(recvbuf + i*recvcount * extent(recvtype), recvcount, i, ...), где i=0,…,n-1.

1.5.10. Все ко всем (векторный вариант)

```
MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                            recvcounts, rdispls, recvtype,comm)

IN  sendbuf        адрес передаваемого буфера
IN  sendcounts     массив, содержащий количество передаваемых
                   элементов каждому процессу
IN  sdispls        массив смещений передаваемых пакетов данных
                   относительно друг друга
IN  sendtype       тип передаваемых данных
OUT recvbuf        адрес буфера приема
IN  recvcounts     массив, указывающий количество принимаемых
                   элементов от процессов
IN  rdispls        массив смещений принимаемых пакетов данных
                   относительно друг друга
IN  recvtype       тип принимаемых данных
IN  comm           коммуникатор (communicator)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
         MPI_Datatype sendtype, void* recvbuf, int *recvcounts,
            int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
                  RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*),
                        RDISPLS(*), RECVTYPE, COMM, IERROR
```

MPI_ALLTOALLV прибавляет гибкость к MPI_ALLTOALL в том, что адреса данных для посылающего процесса определены в массиве sdispls и адреса размещения данных в получающей стороне определены в массиве rdispls. j-й блок, посланный из процесса i, получен процессом j и размещен в i-м блоке массива recvbuf. Не все эти блоки имеют тот же самый размер. Аргументы sendcount[j] и sendtype в процессе i должны быть равны аргу-

ментам recvcount[i] и recvtype в процессе j. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных, попарно между каждой парой процессов. Результат выглядит так, как будто каждый процесс послал сообщение процессу i функцией MPI_SEND (sendbuf + displs[i] * extent (sendtype), sendcounts[i], sendtype, i, ...) и получил сообщение из процесса i запросом к MPI_RECV(recvbuf + displs[i] * extent (recvtype), recvcounts[i], recvtype, i, ...), где i=0,…,n-1.

1.6. Определяемые пользователем типы данных

Часто желательно посылать данные, которые неоднородны (типа структуры) или несмежные в памяти элементы (типа секции массива). MPI обеспечивает два механизма, чтобы достичь этого. Пользователь может определять производные типы (datatypes), которые выявляют более общее расположение данных. Определяемый пользователем тип может использоваться в MPI-функциях связи аналогично, как и базовый, предопределенный тип.

Производный тип данных строится из основных типов данных с использованием строителей типов. Строители могут применяться рекурсивно.

Производный тип данных - непрозрачный объект, который определяет два предмета:

- последовательность примитивных типов и

- последовательность целого числа смещений (в количестве байтов) значений этих типов от начального адреса.

Не требуется, чтобы смещения были положительными, различными или в увеличивающемся порядке. Следовательно, порядок значений типов в списке производного типа не обязательно должен совпадать с их порядком в памяти и значения типов могут появляться в списке больше, чем один раз. Такая пара последовательностей (или последовательность пар) называется *отображением типа*. Последовательность примитивных типов данных (смещения игнорируются) называется *сигнатурой типа* данных.

Предположим

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

такое отображение типа, где $type_i$ - примитивные типы и $disp_i$ - смещения значений этих типов относительно базового адреса.

Предположим

$$Typesig = \{type_0, ..., type_{n-1}\} -$$

соответствующая сигнатура типа. Это отображение типа вместе с базовым адресом buf определяет коммуникационный буфер, который состоит из n элемен-

тов, где i-й элемент стоит по адресу $buf+disp_i$ и имеет тип $type_i$. Сообщение, собранное из отдельных типов, будет состоять из **n** значений типов, определенных Typesig.

Имя производного типа может появляться как аргумент в посылающей или получающей функции вместо аргумента примитивного типа. Функция MPI_SEND(buf, 1, datatype,...) использует адрес посылаемого буфера как базовый адрес buf производного типа данных и тип посылаемых данных как производный тип, соответствующий datatype. Она генерирует сообщение с сигнатурой типа, определенной аргументом datatype. Функция MPI_RECV(buf, 1, datatype,...) использует адрес буфера приема как базовый адрес buf производного типа и тип принимаемых данных как производный тип, соответствующий datatype.

Производные типы данных могут использоваться во всех посылающих и принимающих функциях, включая коллективные функции.

Диапазон (extent) типа данных определен как поле памяти от первого байта до последнего байта, заполненного элементами этого типа данных, округляемыми в большую сторону, чтобы удовлетворить требования к точности совмещения.

ПРИМЕР 1.20

Предположим, что Type={(double,0),(char,8)} (double со смещением ноль, char со смещением восемь). Предположим, кроме того, что вещественные величины (double) должны строго выравниваться по адресам, кратным восьми байтам. Тогда $lb(Type)=min_j disp_j=0$, $ub(Type) = max_j(disp_j + sizeof(type_j)) + e = (8 + 1 + 7)=16$, и диапазон этого типа равен 16 (8 байтов (double) + 1 байт (char) = 9, которая округляется к следующему кратному 8, это есть 16). Отображение этого типа проиллюстрировано на рис. 1.11.

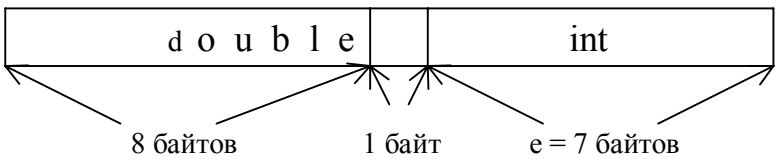


Рис. 1.11. Отображение типа Type={(double,0),(char,8)}

Следующие функции возвращают информацию относительно типов данных.

```
MPI_TYPE_EXTENT(datatype, extent)

IN  datatype        тип данных
OUT extent          диапазон типа datatype

int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

```
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
```

MPI_TYPE_EXTENT возвращает диапазон типа datatype. В дополнение к его использованию с производным datatypes функция может использоваться, чтобы запросить относительно диапазона примитивного datatypes. Например, MPI_TYPE_EXTENT(MPI_INT, extent) возвратит в extent размер в байтах, int - то же самое значение, которое было бы возвращено в С функцией sizeof(int).

Имена типов данных в MPI - непрозрачные, поэтому нужно использовать функцию MPI_TYPE_EXTENT, чтобы определить размер ("size") типа. Нельзя использовать по аналогии, как в С, функцию sizeof(datatype), например, sizeof(MPI_DOUBLE). Она возвратит размер непрозрачного заголовка, который является размером указателя, и, конечно же, отличается от значения sizeof(double).

```
MPI_TYPE_SIZE(datatype, size)
IN  datatype          тип данных
OUT size              размер типа данных

int MPI_Type_size(MPI_Datatype datatype, int *size)

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, SIZE, IERROR
```

MPI_TYPE_SIZE возвращает полный размер, в байтах, входов в сигнатуре типа, связанной с datatype, т.е. полный размер данных в сообщении, которое было бы создано с этим datatype. Элементы, которые встречаются многократно в datatype, учитываются с их кратностью. Для примитивного datatypes эта функция возвращает ту же самую информацию, как MPI_TYPE_EXTENT.

ПРИМЕР 1.21

Допустим datatype имеет тип отображения Type, определенный в примере 1.21. Тогда запрос к MPI_TYPE_EXTENT(datatype,i) возвратит i=16; запрос к MPI_TYPE_SIZE(datatype,i) возвратит i=9 (8 байтов (double)+1 байт(char)=9).

1.6.1. Строитель смежных типов данных CONTIGUOUS

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
IN  count                        количество копий
IN  oldtype                      старый тип данных
OUT newtype                   новый (сконструированный) тип данных
```

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

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

MPI_TYPE_CONTIGUOUS – самый простой строитель типа данных. Он конструирует новый тип данных путем размножения count копий исходного типа в смежные поля. Аргумент newtype – новый полученный тип, представляющий собой count смежных копий исходного типа oldtype. При сочленении копий используется extent(oldtype) (диапазон исходного типа) как размер составных копий. Действие CONTIGUOUS-строителя схематично представлено на рис. 1.12.

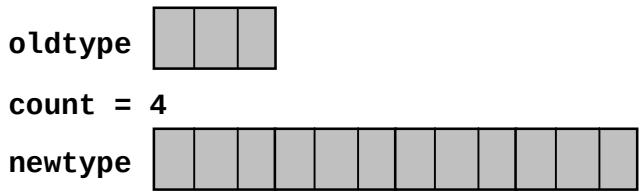


Рис. 1.12. Построение типа данных функцией строителя типов MPI_TYPE_CONTIGUOUS

ПРИМЕР 1.22

Допустим oldtype имеет отображение типа {(double,0), (char,8)}, с диапазоном 16, и допустим count=3. Отображение типа datatype, возвращенного newtype, есть:

```
{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40)},
```

т.е. чередуются double и char элементы со смещениями 0, 8, 16, 24, 32, 40.

1.6.2. Векторный строитель типов данных VECTOR

```
MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)

IN  count                 количество конструируемых блоков
IN  blocklength           количество элементов в каждом блоке
IN  stride                расстояние между началами последовательных
                          блоков, выраженное в количестве
                          исходных элементов
IN  oldtype               исходный тип данных
OUT newtype               новый тип данных

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

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

MPI_TYPE_VECTOR–строитель размножает копии исходного типа oldtype в поля, которые являются равноотстоящими друг от друга блоками. Каждый блок получен из заданного количества (blocklength) смежных копий исходного типа oldtype. Расстояние между блоками здесь задается как расстояние между началами двух соседних блоков. Это расстояние измеряется в единицах oldtype диапазона и одинаково для всех рядом стоящих блоков. В отображении типа смещения блоков даются относительно базового адреса. Например, смещение для нулевого в последовательности блока равно 0*stride, для пятого блока равно 5*stride, и т.д. Действие VECTOR-строителя представлено схематично на рис. 1.13.

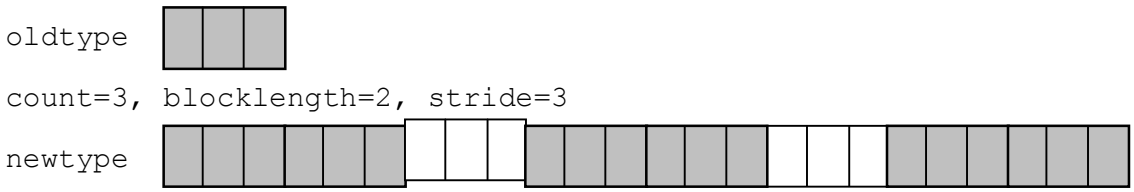


Рис. 1.13. Построение типа данных
функцией строителя типов MPI_TYPE_VECTOR

ПРИМЕР 1.23

Допустим, что oldtype имеет отображение типа {(double,0), (char,8)}, с диапазоном 16. Запрос к MPI_TYPE_VECTOR (2,3,4, oldtype, newtype) создаст новый тип данных newtype с отображением:

```
{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40),
(double,64),(char,72),(double,80),(char,88),(double,96),(char,104)}
```

т.е. два блока с тремя копиями каждый из исходного типа, с расстояниями между началами последовательных блоков в четыре исходных (oldtype) элемента (4 * 16 байтов = 64).

1.6.3. Модифицированный векторный строитель типов данных HVECTOR

Векторный строитель типа, описанный в предыдущем пункте, задает расстояние между началами соседних блоков в единицах oldtype диапазона. Иногда полезно ослабить это предположение и допускать расстояние, измеряемое в байтах. Строитель типа Hvector задает расстояния между началами соседних блоков в байтах. Использование Vector и Hvector-строителей иллюстрируется в примере 1.24.

```
MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)
IN  count                количество конструируемых блоков
```

```
IN  blocklength          количество элементов в каждом блоке
IN  stride               расстояние между началами соседних блоков,
                         выраженное в байтах
IN  oldtype              исходный тип данных
OUT newtype              новый тип данных

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

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

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

MPI_TYPE_HVECTOR идентичен MPI_TYPE_VECTOR, за исключением того, что расстояние (stride) дается в байтах. (Н олицетворяет "heterogeneous" системы). Действие Hvector-строителя схематично представлено на рис. 1.14.

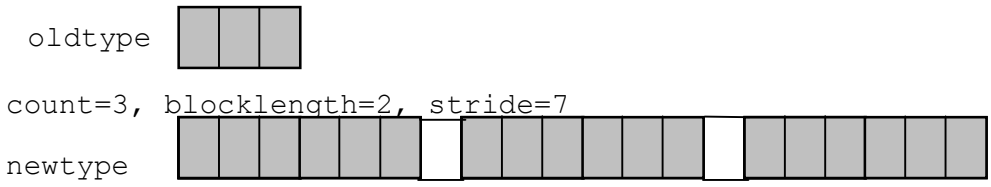


Рис. 1.14. Построение типа данных функцией строителя типов MPI_TYPE_HVECTOR

Пример 1.24 показывает, как определяемый пользователем тип данных используется при посылке верхней триангуляции матрицы. На рис. 1.15 показана диаграмма расположения в памяти определяемого пользователем типа данных.

ПРИМЕР 1.24

Фрагмент программы, которая передает верхнюю триангуляцию матрицы.

```
    double a[100][100], disp[100], blocklen[100], i;
    MPI_Datatype upper;
    /*Вычисление начала и размера каждой строки матрицы (начиная от диа-
гонали) */
    for(i=0; i<100; ++i)
       { disp[i] = 100*i+i;
         blocklen[i] = 100-i;
       }
    /* Создание типа для верхней триангуляции матрицы */
    MPI_Type_indexed(100, blocklen, disp, MPI_DOUBLE, &upper);
    MPI_Type_commit(&upper);
    /* .. и его передача */
    MPI_Send(a, 1, upper, dest, tag, MPI_COMM_WORLD);
```

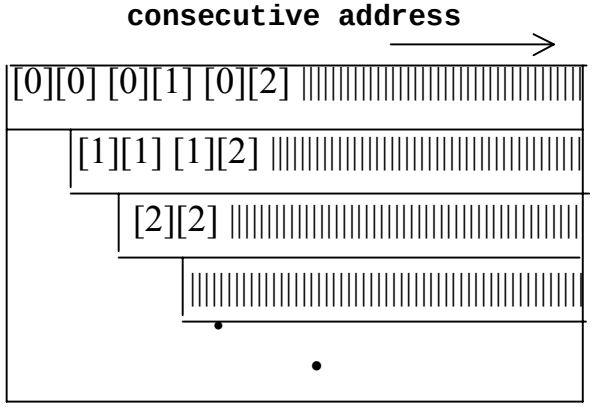


Рис. 1.15. Диаграмма ячеек памяти, представляющих определяемый пользователем тип данных `upper` (Заштрихованные ячейки - это элементы матрицы, которые будут посланы)

1.6.4. Индексированный строитель типов данных INDEXED

Индексированный строитель определяет расположение данных, состоящих из нескольких несмежных блоков. Передающий процесс компонует все блоки в один пакет и посылает их в одном сообщении. Получающий процесс полученные в сообщении блоки раскомпоновывает в соответствии с определением типа.

```
MPI_TYPE_INDEXED(count, array_of_blocklengths,
                 array_of_displacements, oldtype, newtype)
IN  count                          количество блоков
IN  array_of_blocklengths          количество элементов в каждом блоке
IN  array_of_displacements         смещения каждого блока, измеряемых
                                   в единицах исходных элементов
IN  oldtype                        исходный тип данных
OUT newtype                        новый тип данных

int MPI_TYPE_indexed(int count, int *array_of_blocklengths,
                  int *array_of_displacements, MPI_Datatype oldtype,
                  MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                  ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)

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

`MPI_TYPE_INDEXED` делает многократное копирование исходного типа в последовательность блоков, где каждый блок может содержать различное количество копий `oldtype` и иметь различное смещение от начального базового адреса. Размеры каждого блока и расстояния каждого блока от базового адреса задаются параметрами, записанными в массивах. Все смещения блоков от базового адреса измеряются в единицах `oldtype` диапазона. Действие индексированного строителя представлено схематично на рис. 1.16.

Рис. 1.16. Построение типа данных функцией строителя типов MPI_TYPE_INDEXED

ПРИМЕР 1.25

Допустим oldtype имеет отображение типа {(double,0),(char,8)} с диапазоном 16, B=(3,1) и D=(4,0). Функция MPI_TYPE_INDEXED (2,B,D, oldtype, newtype) возвратит следующее отображение типа:

```
{(double,64),(char,72),(double,80),(char,88),(double,96),(char,104),
                                        (double,0), (char, 8)},
```

т.е. три копии исходного типа со смещением от базового адреса 4 * 16 = 64 и одной копии со смещением 0.

1.6.5. Модифицированный индексированный строитель типов данных HINDEXED

Иногда удобно измерять смещения в однотипных элементах диапазона oldtype, но иногда необходимо учесть произвольные смещения. Hindexed-строитель удовлетворяет последнему требованию.

```
MPI_TYPE_HINDEXED(count,array_of_blocklengths,array_of_displacements,
                                                  oldtype, newtype)
IN  count                          количество блоков
IN  array_of_blocklengths          количество элементов в каждом блоке
IN  array_of_displacements         смещения каждого блока, измеряемые
                                   в байтах
IN  oldtype                        исходный тип данных
OUT newtype                        новый тип данных

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

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)

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

MPI_TYPE_HINDEXED идентичен MPI_TYPE_INDEXED, за исключением того, что смещения блоков в массиве смещений определены в байтах, а не в однотипных элементах oldtype диапазона. Действие Hindexed-строителя схематично представлено на рис. 1.17.

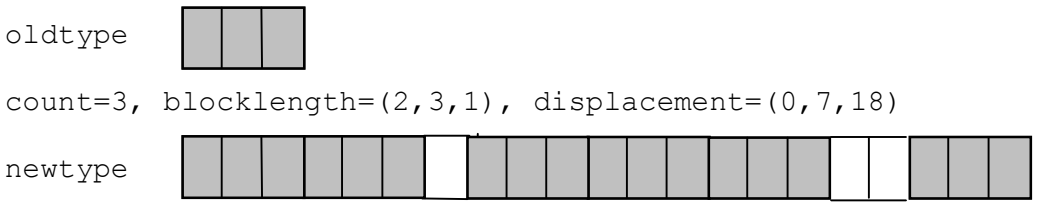


Рис. 1.17. Построение типа данных функцией строителя типов MPI_TYPE_HINDEXED

ПРИМЕР 1.26

Здесь используются те же самые аргументы функции MPI_TYPE_INDEXED, как в примере 1.25. Таким образом, oldtype имеет отображение типа {(double,0),(char,8)} с диапазоном 16, B=(3,1) и D=(4,0). Запрос к MPI_TYPE_HINDEXED(2,B,D,oldtype,newtype) возвращает следующее отображение типа:

```
{(double,4),(char,12),(double,20),(char,28),(double,36),(char;44),
                                              (double,0),(char,8)}.
```

Частичное перекрытие между элементами типа double подразумевает, что тип ошибочен, если этот тип данных используется в посылающем действии. Чтобы получать тот же самый тип данных, как в примере 1.25, нужно, чтобы D=(64,0).

1.6.6. Структурный строитель типов данных STRUCT

```
MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements,
                                          array_of_types, newtype)
IN  count                    количество блоков
IN  array_of_blocklengths    количество элементов в каждом блоке
IN  array_of_displacements   смещения каждого блока, измеряемые
                             в байтах
IN  array_of_types           типы элементов в каждом блоке
OUT newtype                  новый тип данных

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

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
                                    ARRAY_OF_TYPES, NEWTYPE, IERROR)

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

MPI_TYPE_STRUCT - наиболее общий строитель типа. Он далее обобщает MPI_TYPE_HINDEXED и допускает, чтобы каждый блок состоял из дублирований различного типа данных. Для этого имеется массив для описания типов элементов каждого блока. Действие Struct-строителя схематично представлено на рис. 1.18.

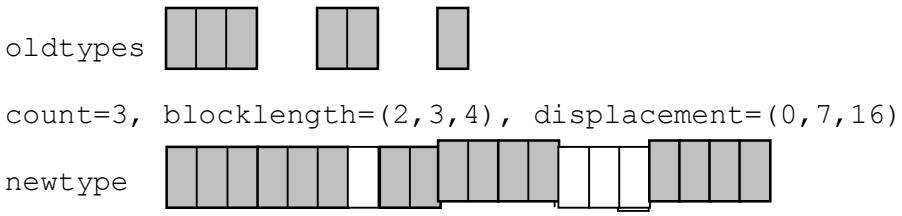


Рис. 1.18. Построение типа данных функцией строителя типов MPI_TYPE_STRUCT

ПРИМЕР 1.27

Допустим type1 имеет отображение типа {(double,0),(char,8)} с диапазоном 16. Допустим B=(2,1,3), D=(0,16,26) и T=(MPI_FLOAT, type1, MPI_CHAR). Тогда вызов MPI_TYPE_STRUCT (3,B,D,T, newtype) построит новый тип с отображением:

{(float,0),(float,4),(double,16),(char,24),(char,26),(char,27),(char,28)}.

Здесь две копии MPI_FLOAT имеют смещение 0, далее одна копия type1 имеет смещение 16, три копии MPI_CHAR имеют смещение 26. (Здесь предполагается, что float занимает четыре байта.)

ПРИМЕР 1.28

Посылка массива структур

```
Struct Partstruct
     {
      char   class;       /* класс частицы */
      double d[6];        /* координаты частицы */
      char   b[7];        /* некоторая другая информация */
     };
struct Partstruct particle[1000];
int  i, dest, rank;
MPI_Comm comm;

/* Построение типа данных, описывающих структуру */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_CHAR};
int      blocklen[3] = {1,6,7};
MPI_Aint disp[3] = {0, sizeof(double), 7*sizeof(double)};
MPI_Type.struct(3, blocklen, disp, type, &Particletype);
```

```
MPI_Type_commit(&Particletype);
/* Послать массив */
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
```

Массив disp был инициализирован, предполагая, что double является выровненным двойным словом. Если double's являются единственным выровненным словом, то disp был инициализирован к (0,sizeof(int), sizeof (int) + 6*sizeof(double)).

1.6.7. Передача и освобождение типа

В данном случае производный тип данных (datatype) передается операционной системе, а не каким-то пользовательским процессам. Производный тип данных должен быть передан прежде, чем он будет использоваться при обмене данными. Переданный datatype может продолжать использоваться как аргумент входа в строителях типов (так, чтобы другой datatypes мог быть получен из переданного datatype). Примитивный тип данных передавать не нужно.

```
MPI_TYPE_COMMIT(datatype)
INOUT datatype        тип данных, который передается в ОС

int MPI_TYPE_commit(MPI_Datatype *datatype)

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

MPI_TYPE_COMMIT передает производный тип данных datatype в ОС. Передача не подразумевает что datatype привязан к текущему содержанию буфера связи. После того, как datatype был передан, он может неоднократно повторно использоваться, чтобы идентифицировать данные.

Объект datatype освобождается запросом к MPI_TYPE_FREE.

```
MPI_TYPE_FREE(datatype)
INOUT datatype        тип данных, который освобождается

int MPI_TYPE_free(MPI_Datatype *datatype)

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

MPI_TYPE_FREE регистрирует объект типа данных, связанный с datatype для освобождения, и устанавливает datatype к MPI_DATATYPE_NULL. Любая связь, которую в это время (постоянно) использует этот datatype, завер-

шится обычно. Производные типы данных, которые были определены из освобожденного типа данных (datatype), не повреждаются.

ПРИМЕР 1.29

Следующий фрагмент программы дает пример использования MPI_TYPE_COMMIT и MPI_TYPE_FREE.

```
int type1, type2;
 /* создание объекта нового типа */
MPI_Type_contiguous(5, MPI_FLOAT, type1);
MPI_Type_commit(type1)        /* новый type1 может быть использован */
                              /* для обменов данными */
type2 = type1                 /* type2 может быть использован для
                                 обменов данными */
 /* создается объект нового типа */
MPI_Type_vecyor(3, 5, 4, MPI_FLOAT, type1)
MPI_Type_commit(type1)     /* новый type1 может использоваться для
                              обменов */
MPI_Type_free(type2)       /* освобождение типа */
type2 = type1              /* type2 может использоваться для
                              обменов */
MPI_Type_free(type2)       /* type1 и type2 недействительны;
                              type2 имеет величину */
                           /* MPI_DATATYPE_NULL и type1 неопределен */
```

1.6.8. Соответствие типов

Предположим, что посылающая функция MPI_Send(buf, count, datatype, dest, tag, comm) выполнена, где datatype имеет отображение типа

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

с диапазоном extent. Функция посылает n*count элементов, где элемент (i,j) записан по адресу $addr_{i,j}$ = buf+extent*i+$disp_j$ и имеет тип $type_j$, для i=0,…,count-1 и j=0,…,n-1.

Точно так же предположим, что выполнена получающая функция MPI_Recv (buf, count, datatype, source, tag, comm, status). Получающее действие получает до n*count элементов, где элемент (i,j) записан по адресу buf+extent*i+$disp_j$ и имеет тип $type_j$. Соответствующий тип определен согласно наименованию типа передачи datatypes, т.е. как последовательность примитивных компонентов типа. Соответствующий тип не зависит от других аспектов определения datatype, типа смещений (расположение в памяти) или промежуточных типов, используемых, чтобы определить datatypes.

Для посылки datatype может определяться с накладывающимися элементами. Это неверно для получения. Если datatype, используемый в получающем действии, определяет накладывающиеся элементы, то запрос ошибочен.

ПРИМЕР 1.30

Этот пример показывает, что соответствующий тип определен только в терминах примитивных типов, которые составляют производный тип.

```
 ...
MPI_Type_contiguous(2, MPI_FLOAT, type2, ...);
MPI_Type_contiguous(4, MPI_FLOAT, type4, ...);
MPI_Type_contiguous(2, type2, type22, ...);
 ...
MPI_Send(a, 4, MPI_FLOAT, ...);
MPI_Send(a, 2, type2, ...);
MPI_Send(a, 1, type22, ...);
MPI_Semd(a, 1, type4, ...); ...
MPI_Recv(a, 4, MPI_FLOAT, ...);
MPI_Recv(a, 2, type2, ...);
MPI_Recv(a, 1, type22, ...);
MPI_Recv(a, 1, type4, ...);
```

Любая посылающая функция соответствует любой получающей функции.

1.7. Таймеры

MPI определяет таймер.

```
MPI_WTIME()

double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME возвращает число с плавающей запятой, представляющее секунды, истекшие за период времени начиная с некоторого момента в прошлом.

Гарантируется, что "время в прошлом" не изменится в течение жизни процесса. Пользователь ответствен за преобразование больших чисел, указывающих секунды, к другим единицам, если они необходимы.

Эта функция переносима (она возвращает секунды, не "ticks"), она допускает высокое разрешение. Можно использовать ее аналогично этому:

```
{
double starttime, endtime;
starttime = MPI_Wtime();

/*.... Программа ... */

endtime = MPI_Wtime();
printf("That took %f seconds\n", endtime-starttime);
}
```

Возвращенные времена, местные к узлам, которые вызывают функцию:

```
MPI_WTICK()

double MPI_Wtick(void)

DOUBLE PRECISION MPI_WTICK()
```

MPI_WTICK возвращает решение в секундах, т.е. число возвращается с двойным значением точности количества секунд между последовательными засечениями времени. Например, если датчик времени выполнен счетчиком оборудования с разрешающей способностью до миллисекунды, возвращаемое значение MPI_WTICK должно быть 10^{-3}.

1.8. Инициализация и выход

MPI требует, чтобы библиотека была установлена прежде, чем другие функции MPI будут использованы. Чтобы обеспечить это, MPI включает функцию инициализации MPI_INIT.

```
MPI_INIT()

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

MPI_INIT(IERROR)
INTEGER IERROR
```

Эта функция должна быть названа перед любой другой функцией MPI. Она должна быть названа не более одного раза; последующие запросы к этой функции будут ошибочны.

В версии для ANSI C argc и argv обеспечиваются аргументами в main:

```
int main(int argc, char **argv)
  {
    MPI_Init(&argc, &argv);
 /* Разбор аргументов */
 /* Главная программа */
    MPI_Finalize();        /* Смотри ниже */
  }
```

Версия Fortran(a) берет только IERROR.

MPI приложения требуют, чтобы аргументы в C были аргументами к main. Аргументы командной строки обеспечиваются MPI_Init для того, чтобы MPI

приложения использовали их в инициализации окружающей среды MPI. Они передаются по ссылке.

```
MPI_FINALIZE()

int MPI_Finalize(void)

MPI_FINALIZE(IERROR)
INTEGER IERROR
```

Эта функция освобождает (очищает) все MPI установки. Если только эта функция названа, никакая MPI функция (даже `MPI_INIT`) не может быть уже названа. В конце программы обязательно должна вызываться функция `MPI_FINALIZE`.

```
MPI_INITIALIZED(flag)
OUT flag Флаг равен true, если MPI_INIT была вызвана, иначе
false

int MPI_Initialized(int *flag)

MPI_INITIALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR
```

Эта функция может использоваться, чтобы определить, была ли `MPI_INIT` вызвана. Это единственная функция, которая может быть вызвана прежде, чем функция `MPI_INIT`.

```
MPI_ABORT(comm,errorcode)
IN comm       Коммуникатор, предназначенный для освобождения
IN errorcode  Код ошибки для возвращения вызываемой окружающей средой

int MPI_Abort(MPI_Comm comm, int errorcode)

MPI_ABORT(COMM, ERRORCODE, IERROR)
INTEGER COMM, ERRORCODE, IERROR
```

Эта функция прерывает все задачи в группе `comm`. Окружающая среда Unix или POSIX должна обратиться с этим как `return errorcode` из главной программы или `abort(errorcode)`.

Для MPI приложений требуется определить поведение `MPI_ABORT` по крайней мере для `comm`, равного `MPI_COMM_WORLD`. MPI приложения могут игнорировать аргумент `comm` и действовать, как будто `comm` была `MPI_COMM_WORLD`.

Поведение функции MPI_ABORT(comm,errorcode) для comm другого, чем MPI_COMM_WORLD, является зависимым от выполнения. Запрос к MPI_ABORT (MPI_COMM_WORLD, errorcode) должен всегда заставлять все процессы в группе MPI_COMM_WORLD прерываться.

1.9. Коды ошибок

Большинство MPI-функций возвращает код ошибки, указывающий успешное выполнение (MPI_SUCCESS) или обеспечение информации относительно типа MPI ошибки, которая произошла. MPI обеспечивает стандартный набор значений ошибок.

Имеются следующие классы ошибок

MPI_SUCCESS	Нет ошибок
MPI_ERR_BUFFER	Недействительный буферный указатель
MPI_ERR_COUNT	Недействительный аргумент счета
MPI_ERR_TYPE	Недействительный аргумент типа данных(datatype)
MPI_ERR_TAG	Недействительный аргумент тега
MPI_ERR_COMM	Недействительный переключатель каналов
MPI_ERR_RANK	Недействительный ранг
MPI_ERR_REQUEST	Недействительный запрос
MPI_ERR_ROOT	Недействительный корень
MPI_ERR_GROUP	Недействительная группа
MPI_ERR_OP	Недействительная операция
MPI_ERR_TOPOLOGY	Недействительная топология
MPI_ERR_DIMS	Недействительный аргумент измерения декартовой структуры
MPI_ERR_ARG	Недействительный аргумент некоторого другого вида
MPI_ERR_UNKNOWN	Неизвестная ошибка
MPI_ERR_TRUNCATE	Сообщение, усеченное при получении
MPI_ERR_OTHER	Известная ошибка не в этом списке
MPI_ERR_INTERN	Внутренняя MPI ошибка
MPI_ERR_IN_STATUS	Ошибочный код в состоянии (status)
MPI_ERR_PENDING	Ждущий запрос
MPI_ERR_LASTCODE	Последний код ошибки

2. Лабораторные работы

В этом разделе приведены лабораторные работы, проводимые с учащимися в терминальном классе. Лабораторные работы рассчитаны приблизительно на 18 - 20 занятий. Предполагается, что учащиеся одновременно работают за несколькими терминалами.

Цель лабораторных работ – освоение и закрепление навыков по параллельному программированию на системе с передачей сообщений MPI. В лабораторных работах приводятся примеры решения конкретных задач: умножение матрицы на вектор, умножение матрицы на матрицу, решение систем линейных уравнений методами Гаусса и простой итерации. Программирование осуществ-

ляется на языке С. Каждая лабораторная работа построена по принципу последовательного наращивания сложности С-программ, предлагаемых учащимся.

Лабораторная работа № 1
ПРОГРАММИРОВАНИЕ НА БАЗОВОЙ ТОПОЛОГИИ MPI_COMM_WORLD

Цель - практическое освоение парных взаимодействий параллельных процессов на базовой топологии `MPI_COMM_WORLD`, являющейся полным графом. Освоение операторов компиляции С-программ и Фортран-программ, операторов запуска программ на системе, MPI–функций системных парных взаимодействий параллельных процессов.

ПРИМЕР 2.1

Ветви параллельной программы (п-программы) выводят на экран заданный текст. Этот пример в основном ориентирован на усвоение операторов компиляции и запуска программ на системе. С помощью текстового редактора mc (или любого другого редактора) наберите программу на языке С.

Команда: `mcedit hello.c`

Программа:

```
#include<stdio.h>
 int main()
  { printf("hello, world\n");
    return(0);
  }
```

Выйдите из редактора с запоминанием набранной программы.

2.1.1. Наберите в командной строке и исполните оператор компиляции:

```
mpicc -o hello hello.c
```

Если есть ошибки, исправьте (в том же редакторе).

2.1.2. Наберите в командной строке и исполните оператор запуска исполняемой программы:

```
mpirun -np 4 hello
```

На экран монитора выводится результат:

```
hello, world
hello, world
hello, world
hello, world
```

Число 4 в операторе запуска означает, что исполняемая программа запущена как четыре независимых параллельных процесса, т.е. как четыре ветви параллельной программы (см. п. 1.1 этого пособия). Везде далее понятия "па-

раллельные процессы" и "ветви параллельной программы" являются синонимами. Следует обратить внимание учащихся на принцип работы оператора `mpirun`: программа `hello` загружается в подсистему путем копирования в каждый из четырех созданных виртуальных компьютеров и после этого каждая копия запускается.

ПРИМЕР 2.2

Каждая ветвь п-программы выводит на экран свой идентификационный номер и размер заказанной параллельной системы, т.е. количество виртуальных компьютеров, в каждый из которых загружается ветвь п-программы. Предыдущую программу (с целью экономии времени) переписать в следующую программу:

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("SIZE = %d RANK = %d\n",size,rank);
    return(0);
  }
```

Повторите пункты 2.1.1 и 2.1.2.

ПРИМЕР 2.3

Ветвь с номером 0 пересылает информацию (в данном случае свой идентификационный номер) ветви с номером 7. Ветвь 7 выводит свой номер и принятый номер от нулевого. Предыдущую программу продолжаем редактировать, добавляя нужные операторы.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank, r;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
     { MPI_Send(&rank,1,MPI_INT,7,2,MPI_COMM_WORLD);
    else
     { if(rank == 7)
        { MPI_Recv(&r,1,MPI_INT,0,2,MPI_COMM_WORLD);
          printf("proces= %d r= %d\n",rank,r);
        }
     }
```

```
    return(0);
  }
```

Повторите пункты 2.1.1 и 2.1.2 с оператором запуска: `mpirun -np 8 hello`.

ПРИМЕР 2.4

Ветвь 0 пересылает информацию (свой номер) ветви 1, ветвь 1 принятый номер пересылает ветви 2, ветвь 2 принятый номер пересылает ветви 3 и т.д. по цепочке увеличения номеров. И наконец, ветвь 0 принимает пересылаемую информацию от ветви size-1 и выводит на экран свой номер и принятый номер, т.е. пересылка информации идет по "кольцу" компьютеров. Продолжаем усложнять предыдущую программу, используя те же операторы.

```
#include<stdio.h>
#include<mpi.h>
 int main(int argc, char **argv)
  { int size, rank, r;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
     { MPI_Send(&rank, 1, MPI_INT, rank+1, 2, MPI_COMM_WORLD);
       MPI_Recv(&r, 1, MPI_INT, size-1, 2, MPI_COMM_WORLD);
       printf("proces= %d r= %d\n",rank,r);
    else
     { MPI_Recv(&r, 1, MPI_INT, rank-1, 2, MPI_COMM_WORLD);
       MPI_Send(&r, 1, MPI_INT, (rank+1)%size, 2, MPI_COMM_WORLD);
     }
    return(0);
  }
```

В этом примере нужно обратить внимание на то, что алгоритм не зависит от числа компьютеров и программа может запускаться на любом допустимом количестве компьютеров.

В случае зависания системы нужно нажать клавиши Ctrl C: если зависание продолжается, нужно сделать повторный вход и удалить зависший процесс командой kill.

Контрольные вопросы к лабораторной работе № 1

1. Как скомпилировать С-программу на MPI?

2. Как запустить С-программу на MPI?

3. Действия системы по команде mpirun(…)?

4. Каким образом осуществляется загрузка программы в систему?

5. Какие типы функций парных системных взаимодействий имеются в MPI?

Лабораторная работа № 2
ПРОГРАММИРОВАНИЕ НА ТОПОЛОГИЯХ

Цель - практическое освоение программирования параллельных процессов, исполняющихся на декартовой топологии связей и топологии "граф". Освоение функций задания декартовой топологии и топологии "граф". В лабораторной работе приведены три примера. Примеры связаны с взаимодействиями параллельных процессов на декартовых структурах, а также приведен пример, имитирующий работу на топологии "звезда", у которой "листья", в свою очередь, связаны топологией "кольцо".

ПРИМЕР 2.5

В примере выполняется сдвиг данных соседним ветвям вдоль координат компьютеров на топологии "двумерный тор" на один шаг, т.е. все ветви параллельной программы передают данные соседним ветвям, например, в сторону увеличения координат компьютеров.

```
#include <mpi.h>
#include <stdio.h>
#define DIMS 2
int main(int argc, char** argv)
  { int        rank, size, i, A, B, dims[DIMS];
    int        periods[DIMS], sourc1, sourc2, dest1, dest2;
    int        reorder = 0;
    MPI_Comm   comm_cart;
    MPI_Status status;
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает общее количество ветвей */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 /* и свой номер: от 0 до (size-1) */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    A = rank;
    B = -1;
 /* Обнуляем массив dims и заполняем массив periods для топологии
   "двумерный тор" */
    for(i = 0; i < DIMS; i++) { dims[i] = 0; periods[i] = 1; }
 /* Заполняем массив dims, в котором указываются размеры решетки */
    MPI_Dims_create(size, DIMS, dims);
 /* Создаем топологию "двумерный тор" с коммуникатором comm_cart */
    MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder,
                                                    &comm_cart);
 /* Каждая ветвь находит своих соседей вдоль координат,
  * в направлении больших значений координат */
    MPI_Cart_shift(comm_cart, 0, 1, &sourc1, &dest1);
    MPI_Cart_shift(comm_cart, 1, 1, &sourc2, &dest2);
 /* Каждая ветвь передает свои данные (значение переменной A)
    соседней ветви с большим значением координаты и принимает данные
    в B от соседней ветви с меньшим значением координаты
    вдоль "кольца". */
 /* Каждая ветвь выводит свой ранг (он же и был послан соседней
  * ветви) и значение переменной B (ранг соседней ветви) */
```

```
    MPI_Sendrecv(&A, 1, MPI_INT, dest1, 2, &B, 1, MPI_INT, sourc1,
    2, comm_cart, &status);
    printf("rank = %d  B=%d\n", rank, B);
    MPI_Sendrecv(&A, 1, MPI_INT, dest2, 2, &B, 1, MPI_INT, sourc2,
                                          2, comm_cart, &status);
    printf("rank = %d  B=%d\n", rank, B);
 /* Все ветви завершают системные процессы, связанные с топологией
  * comm_cart, и завершают выполнение программы */
    MPI_Comm_free(&comm_cart);
    MPI_Finalize();
    return 0;
  }
```

ПРИМЕР 2.6

Аналогичен примеру 2.4, но с использованием топологии. Нулевая ветвь инициирует запуск данных вдоль "кольца" компьютеров, посланные данные последовательно "проходят" по всем компьютерам и возвращаются в нулевой компьютер, реализующий нулевую ветвь.

```
#include <mpi.h>
#include <stdio.h>
#define DIMS 1
int main(int argc, char** argv)
  { int        rank, size, i, A, B, dims[DIMS];
    int        periods[DIMS], source, dest;
    int        reorder = 0;
    MPI_Comm   comm_cart;
    MPI_Status status;
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает количество ветвей */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* Обнуляем массив dims и заполняем массив periods для топологии
    "кольцо" */
    for(i = 0; i < DIMS; i++) { dims[i] = 0; periods[i] = 1; }
 /* Заполняем массив dims, где указываются размеры (одномерной)
  * решетки */
     MPI_Dims_create(size, DIMS, dims);
 /* Создаем топологию "кольцо" с коммуникатором comm_cart */
    MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder,
                                                    &comm_cart);
 /* Каждая ветвь определяет свой номер: от 0 до (size-1) */
    MPI_Comm_rank(comm_cart, &rank);
    A = rank;
    B = -1;
 /* Каждая ветвь находит своих соседей вдоль кольца в направлении
  * больших значений рангов  */
    MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
 /* 0-ветвь инициирует передачу данных (значение своего ранга) вдоль
  * кольца и принимает это же значение от ветви size-1  */
    if(rank == 0)
     { MPI_Send(&A, 1, MPI_INT, dest, 12, comm_cart);
       MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
       printf("rank=%d B=%d \n", rank, B);
     }
```

```
   /* Работа всех остальных ветвей  */
      else
       { MPI_Recv(&B, 1, MPI_INT, source, 12, comm_cart, &status);
         MPI_Send(&B, 1, MPI_INT, dest, 12, comm_cart);
       }
   /* Все ветви завершают системные процессы, связанные с топологией
    * comm_cart, и завершают выполнение программы */
      MPI_Comm_free(&comm_cart);
      MPI_Finalize();
      return 0;
    }
```

ПРИМЕР 2.7

Из N запускаемых компьютеров создается топология "звезда", в которой нулевой компьютер является корневым, а "листья" соединены, кроме того, между собой в кольцо (см. рис. 2.1).

Корневой компьютер посылает данные "листьям", те в свою очередь обрабатывают полученную информацию и отсылают ее обратно в корневой компьютер. Корневой компьютер печатает некоторую информацию.

На "листьях" можно создать любую необходимую топологию, в данном случае создается топология "кольцо".

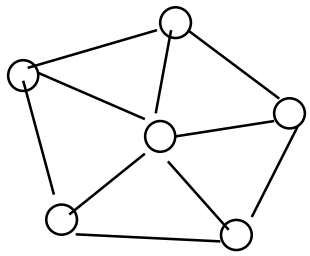


Рис. 2.1. Задаваемая в программе топология

```
   /* Нулевая ветвь передает данные (значение своего ранга) остальным
    * ветвям, после обработки информации данные собираются в нулевой
   * ветви. Программа запускается на произвольном допустимом числе
   * компьютеров *#include <mpi.h>#include <stdio.h>#define DIMS 1

   int main(int argc, char** argv)
     {
       int        rankgr, rankc, size, sizel,i,v,j,key,color, A, *B, C;
       int
periods[DIMS],dims[DIMS],*index,*edges,source,dest,D[2];
       int        reord = 0;
       MPI_Comm   comm_cart, comm_gr, comm_0, comm_1;
       MPI_Status st;
       MPI_Init(&argc, &argv);
    /* Каждая ветвь узнает количество ветвей */
       MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Выделяем память под массивы для описания вершин (index) и
```

```
      ребер (edges) в топологии граф */
      index = (int *)malloc(size * sizeof(int));
      edges = (int *)malloc( ((size-1)+(size-1) * 3) * sizeof(int));

    /* Заполняем массивы для описания вершин и ребер для топологии
     * граф и задаем топологию "граф". */
      index[0] = size - 1;
      for(i = 0; i < size-1; i++)
        edges[i] = i+1;
        v = 0;
      for(i = 1; i < size; i++)
        { index[i] = (size - 1) + i * 3;
          edges[(size - 1) + v++] = 0;
          edges[(size - 1) + v++] = ((i-2)+(size-1))%(size-1)+1;
          edges[(size - 1) + v++] = i % (size-1) + 1;
         }

      MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,
      &comm_gr);  /* Каждая ветвь определяет свой номер:
      от 0 до (size-1) */

      MPI_Comm_rank(comm_gr, &rankgr);

     if(rankgr == 0)
      { color = MPI_UNDEFINED;
        key = 0;
        MPI_Comm_split(comm_gr, color, key, &comm_0);
      }
     else
      { color == 1;
        key = rankgr;
        MPI_Comm_split(comm_gr, color, key, &comm_l);

    /* Каждая ветвь узнает количество ветвей в comm_l
   */ MPI_Comm_size(comm_l, &sizel);
   /* В comm_l создаем теперь топологию "кольцо" comm_cart
   */
   /* Обнуляем массив dims и заполняем массив periods для
      топологии "кольцо" */ for(i = 0; i < DIMS; i++) { dims[i] = 0;
      periods[i] = 0; }
   /* Заполняем массив dims, где указываются размеры (одномерной)
      решетки */ MPI_Dims_create(sizel, DIMS, dims);
    /* Создаем топологию "кольцо" с communicator(ом) comm_cart */
       MPI_Cart_create(comm_l, DIMS, dims, periods, reord, &comm_cart);
    /* Каждая ветвь определяет свой номер в "кольце" */
       MPI_Comm_rank(comm_cart, &rankc); /* Каждая ветвь находит своих
     * соседей вдоль кольца, в направлении больших значений рангов */

        MPI_Cart_shift(comm_cart, 0, 1, &source, &dest);
      }

    /* 0-ветвь инициирует передачу данных (значение своего ранга) "ли-
стьям" *, которые образуют, в свою очередь, топологию "кольцо" */
    D[0] = 0;
    D[1] = 0; C = 0; A = 0; B = (int *)malloc(2 * size * sizeof(int));
         for(i = 0; i < 2*size; i++)
```

```
          B[i] = 0;

     /* Работа корневой ветви */
         if(rankgr == 0)
           { A = rankgr;
     /* Рассылка данных "листьям" */
           MPI_Bcast(&A, 1, MPI_INT, 0, comm_gr); /* Сбор данных от
"листьев" */ MPI_Gather(D, 2, MPI_INT, B, 2, MPI_INT, 0, comm_gr); for(i
= 0; i < 2*size; i++)
             printf(" B = %d\n",B[i]);
           }
     /* Работа всех остальных ветвей  */     else
           {
     /* Прием данных от корневой ветви */
           MPI_Bcast(&A, 1, MPI_INT, 0, comm_gr);
     /* Имитация обработки информации */ A = A + rankgr; D[0] = A;
           MPI_Sendrecv(&A,1,MPI_INT,dest,12,&C,1,MPI_INT,source,12,
                                                    comm_l,&st);
           D[1] = C;
     /* Посылка данных в корневую ветвь MPI_Gather(D, 2, MPI_INT, B, 2,
        MPI_INT, 0, comm_gr); /* Освобождение топологий в "листьях" */
           MPI_Comm_free(&comm_cart);
           MPI_Comm_free(&comm_l); } /* Все ветви завершают выполнение
           программы */ MPI_Comm_free(&comm_gr);
        MPI_Finalize();
        return 0;
       }
```

Контрольные вопросы к лабораторной работе № 2

1. Как задаются декартовы топологии?

2. Как задаются топологии "граф"?

3. С помощью функции задания топологии "граф" можно задать и декартову топологию. Почему декартовы топологии задаются отдельной функцией?

4. Как и зачем определять соседние компьютеры на декартовой топологии?

5. Для чего нужны топологии?

Лабораторная работа № 3
УМНОЖЕНИЕ МАТРИЦЫ НА ВЕКТОР И МАТРИЦЫ НА МАТРИЦУ

Цель - освоение методов распараллеливания алгоритмов решения задач, таких, как умножение матрицы на вектор и матрицы на матрицу. Эти задачи являются, в свою очередь, макрооперациями в итерационных задачах. В лабораторной работе приведены три примера. Два примера связаны с разными способами умножения матрицы на вектор и один - умножение матрицы на матрицу.

ПРИМЕР 2.8

Умножение матрицы на вектор в топологии "кольцо". Задана исходная матрица A и вектор B. Вычисляется произведение C = A x B, где A - матрица n1 x n2 и B - вектор n2. Исходные матрица и вектор предварительно разрезаны на полосы, и каждая ветвь генерирует свои части. Схема распределения данных по компьютерам приведена на рис. 2.2.

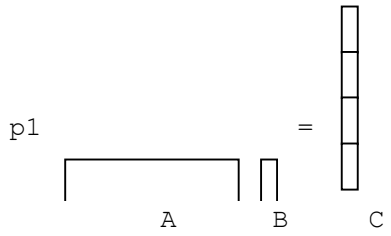


Рис. 2.2. Разрезание данных для параллельного алгоритма

Реализация алгоритма выполняется на системе из p1 компьютеров. Матрица A, вектор B и вектор C разрезаны на p1 горизонтальных полос. Здесь предполагается, что в память каждого компьютера загружается и может там находиться только одна полоса матрицы A и одна полоса матрицы B.

```
/* В примере предполагается, что количество строк матрицы A и B
 * делится без остатка на количество компьютеров в
 * системе.
 * В данном случае задачу запускаем на четырех компьютерах.
 */
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>
#include <sys/time.h>
/* Задаем в каждой ветви размеры полос матриц A и B. (Здесь
 * предполагается,
 * что размеры полос одинаковы во всех ветвях. */
#define M 16
#define N 4
/* NUM_DIMS - размер декартовой топологии. "кольцо" - одномерный
   тор. */
#define DIMS 1
#define EL(x) (sizeof(x) / sizeof(x[0][0]))
/* Задаем полосы исходных матриц. В каждой ветви в данном случае
 * они одинаковы */
static double A[N][M], B[N], C[N];
int main(int argc, char **argv)
  { int              rank,size,i,j,k,i1,d,sour,dest;
    int              dims[DIMS];
    int              periods[DIMS];
    int              new_coords[DIMS];
    int              reorder = 0;
```

```
        MPI_Comm         comm_cart;
        MPI_Status       st;
        int rt, t1, t2;
    /* Инициализация библиотеки MPI*/
        MPI_Init(&argc, &argv);
     /* Каждая ветвь узнает количество задач в стартовавшем
        приложении */
        MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Обнуляем массив dims и заполняем массив periods для топологии
      * "кольцо" */
        for(i=0; i < DIMS; i++) { dims[i] = 0; periods[i] = 1; }
    /*  Заполняем  массив  dims,  где  указываются  размеры  (одномерной)
решетки */
        MPI_Dims_create(size, DIMS, dims);
     /* Создаем топологию "кольцо" с communicator(ом) comm_cart */
        MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder,
                                                          &comm_cart);
     /* Каждая ветвь определяет свой собственный номер: от 0 до
        (size-1) */
        MPI_Comm_rank(comm_cart, &rank);
     /* Каждая ветвь находит своих соседей вдоль кольца, в направлении
      * меньших значений рангов */
        MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest);
     /* Каждая ветвь генерирует полосы исходных матриц A и B, полосы C
      * обнуляет */
        for(j = 0; j < N; j++)
         { for(i = 0; i < M; i++)
             A[j][i] = 3.0;       B[j] = 2.0;
           C[j] = 0.0;
         }
    /* Засекаем начало умножения матриц */
         t1 = MPI_Wtime();
      /* Каждая ветвь производит умножение своих полос матрицы
         и вектора */
      /* Самый внешний цикл for(k) - цикл по компьютерам */
         for(k = 0; k < size; k++)
         { d = ((rank + k)%size)*N;
     /* Каждая ветвь производит умножение своей полосы матрицы A на
      * текущую полосу матрицы B */
            for(j = 0; j < N; j++)
              { for(i1=0, i = d; i < d+N; i++, i1++)
                 C[j] += A[j][i] * B[i1];
              }
     /* Каждая ветвь передает своим соседним ветвям с меньшим рангом
      * полосы вектора B, т.е. полосы вектора B сдвигаются вдоль
      * кольца компьютеров */
            MPI_Sendrecv_replace(B, EL(V), MPI_DOUBLE, dest, 12, sour,
                                                  12, comm_cart, &st);
         }
    /* Умножение завершено. Каждая ветвь умножила свою полосу строк
      * матрицы A на все полосы вектора B. Засекаем время и
      * результат печатаем */
        t2 = MPI_Wtime();
        rt = t2 - t1;
         printf("rank = %d Time = %d\n", rank, rt);
     /* Для контроля печатаем первые N элементов
```

```
  * результата */
    for(i = 0; i < N; i++)
      printf("rank = %d RM = %6.2f\n"rank,C[i]);
 /* Все ветви завершают системные процессы, связанные с топологией
  * comm_cart, и завершают выполнение программы */
    MPI_Comm_free(&comm_cart);
    MPI_Finalize();
    return(0);
  }
```

Обратить внимание на способ "прокручивания" данных по процессорам и на формулу начального и конечного значения переменной цикла i (в самом внутреннем цикле).

ПРИМЕР 2.9

Умножение матрицы на вектор в топологии "полный граф". Задана исходная матрица A и вектор B. Вычисляется произведение C = A x B, где A - матрица n1 x n2 и B - вектор n2. Исходная матрица предварительно разрезана на полосы, как на рис. 2.2, и вектор B дублирован в каждой ветви. Каждая ветвь генерирует свои части. Программа делает следующее: умножает матрицу на вектор и получает распределенный по компьютерам результат - матрицу C; затем разрезанные части матрицы C соединяет в единый вектор, который записывается в вектор B во всех компьютерах. (Это модель одной итерации в итерационных алгоритмах.)

```
/* В примере предполагается, что количество строк матриц A и B
 * делится без остатка на количество компьютеров в
 * системе.
 * В данном случае задачу запускаем на четырех компьютерах.
 */
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>
#include <sys/time.h>
/* Задаем в каждой ветви размеры полос матриц A и B. (Здесь
 * предполагается,
 * что размеры полос одинаковы во всех ветвях. */
   #define M 16
   #define N 4
/* NUM_DIMS - размер декартовой топологии.
              "кольцо" - одномерный тор. */
   #define DIMS 1
   #define EL(x) (sizeof(x) / sizeof(x[0][0]))
/* Задаем полосы исходных матриц. В каждой ветви в данном случае
 * они одинаковы */
static double A[N][M], B[M], C[N];
int main(int argc, char** argv)
  { int rt, t1, t2, rank,size,i,j;
 /* Инициализация библиотеки MPI*/
```

```
        MPI_Init(&argc, &argv);
      /* Каждая ветвь узнает количество задач в стартовавшем приложении
*/
        MPI_Comm_size(MPI_COMM_WORLD, &size);
     /* Каждая ветвь определяет свой собственный номер: от 0 до
        (size-1) */
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     /* Каждая ветвь генерирует полосы исходных матриц A и B, полосы C
       * обнуляет */
        for(i = 0; i < M; j++) { for(j = 0; j < N; i++) { A[j][i] = 3.0;
                C[j] = 0.0;
              }
            B[i] = 2.0;
          } /* Засекаем начало умножения матриц */
        t1 = MPI_Wtime();
      /* Каждая ветвь производит умножение своих полос матрицы и
        вектора */
        for(j = 0; j < N; j++)
          { for(i = 0; i < M; i++)
              C[j] += A[j][i] * B[i];
           } /* Соединяем части вектора C и записываем их в вектор B
       * во все компьютеры */
        MPI_Allgather(C, N, MPI_DOUBLE, B, N, MPI_DOUBLE,
        MPI_COMM_WORLD);
      /* Каждая ветвь печатает время решения, нулевая ветвь печатает
        вектор B */
        t2 = MPI_Wtime();
        rt = t2 - t1;
        printf("rank = %d Time = %d\n",rank,rt);
        if(rank == 0)      { for(i = 0; i < M; i++)
             printf("B =  %6.2f\n",B[i]);      }
        MPI_Finalize();
        return(0);  }
```

ПРИМЕР 2.10

Умножение матрицы на матрицу в топологии "кольцо". Заданы две исходные матрицы A и B. Вычисляется произведение C = A x B, где A - матрица n1 x n2 и B - матрица n2 x n3. Матрица результатов C имеет размер n1 x n3. Исходные матрицы предварительно разрезаны на полосы и каждая ветвь генерирует свои части матриц. Схема распределения данных по компьютерам приведена на рис. 2.3.

Реализация алгоритма выполняется на "кольце" из p1 компьютеров. Матрицы разрезаны, как показано на рис. 2.3. Матрица A разрезана на p1 горизонтальных полос, матрица B разрезана на p1 вертикальных полос, и матрица результата C разрезана на p1 горизонтальные полосы. Здесь предполагается, что в память каждого компьютера загружается и может там находиться только одна полоса матрицы A и одна полоса матрицы B.

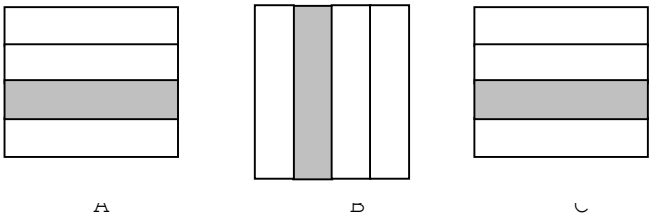


Рис. 2.3. Разрезание данных для параллельного алгоритма произведения двух матриц при вычислении на кольце компьютеров. Выделенные полосы расположены в одном компьютере

```
/* В примере предполагается, что количество строк матрицы A
 * и количество столбцов матрицы B делятся без остатка на количество
 * компьютеров в системе.
 * В данном случае задачу запускаем на восьми компьютерах.
 */
   #include<stdio.h>   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>
/* Задаем в каждой ветви размеры полос матриц A, B и C. (Здесь
 * предполагается,
 * что размеры полос одинаковы во всех ветвях). */
   #define M 320
   #define N 40
/* NUM_DIMS - размер декартовой топологии. "кольцо" - одномерный
   тор. */
   #define DIMS 1
   #define EL(x) (sizeof(x) / sizeof(x[0][0]))
/* Задаем полосы исходных матриц. В каждой ветви в данном случае
 * они одинаковы */
static double A[N][M], B[M][N], C[N][M];
int main(int argc, char **argv)
  { int        rank, size, i, j, k, i1, j1, d, sour, dest;
    int        dims[DIMS], periods[DIMS], new_coords[DIMS];
    int        reorder = 0;
    MPI_Comm   comm_cart;
    MPI_Status st;
    int rt, t1, t2;            /* Для засечения времени */
 /* Инициализация библиотеки MPI*/
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает количество задач в стартовавшем
    приложении */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* Обнуляем массив dims и заполняем массив periods для топологии
  * "кольцо" */
    for(i=0; i < DIMS; i++) { dims[i] = 0; periods[i] = 1; }
/* Заполняем массив dims, где указываются размеры
   (одномерной) решетки */
    MPI_Dims_create(size, DIMS, dims);
 /* Создаем топологию "кольцо" с communicator(ом) comm_cart */
    MPI_Cart_create(MPI_COMM_WORLD, DIMS, dims, periods, reorder,
                                                    &comm_cart);
 /* Каждая ветвь определяет свой собственный номер: от 0 до
    (size-1) */
```

```
   MPI_Comm_rank(comm_cart, &rank);
/* Каждая ветвь находит своих соседей вдоль кольца, в направлении
 * меньших значений рангов */
   MPI_Cart_shift(comm_cart, 0, -1, &sour, &dest);
/* Каждая ветвь генерирует полосы исходных матриц A и B, полосы C
 * обнуляет */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { A[i][j] = 3.141528;
            B[j][i] = 2.812;
            C[i][j] = 0.0;
          }
      }
 /* Засекаем начало умножения матриц */
    t1 = MPI_Wtime();
 /* Каждая ветвь производит умножение своих полос матриц */
 /* Самый внешний цикл for(k) - цикл по компьютерам */
    for(k = 0; k < size; k++)
      {
/* Каждая ветвь вычисляет координаты (вдоль строки)
   для результирующих
 * элементов матрицы C, которые зависят от номера цикла k и
 * ранга компьютера. */
       d = ((rank + k)%size)*N;
/* Каждая ветвь производит умножение своей полосы матрицы A на
 * текущую полосу матрицы B */
       for(j = 0; j < N; j++)          { for(i1 = 0, j1 = d;
       j1 < d+N; j1++, i1++)
             { for(i = 0; i < M; i++)
                 C[j][j1] += A[j][i] * B[i][i1];
             }
       }
/* Умножение полосы строк матрицы A на полосу столбцов матрицы B в
 * каждой ветви завершено */
/* Каждая ветвь передает своим соседним ветвям с меньшим рангом
 * вертикальные полосы матрицы B, т.е. полосы матрицы B сдвигаются
 * вдоль кольца компьютеров */
     MPI_Sendrecv_replace(B, EL(B), MPI_DOUBLE, dest, 12, sour, 12,
                                                  comm_cart, &st);
      }
/* Умножение завершено. Каждая ветвь умножила свою полосу строк
 * матрицы A на все полосы столбцов матрицы B. Засекаем время и
 * результат печатаем */
   t2 = MPI_Wtime();
   rt = t2 - t1;
    printf("rank = %d Time = %d\n", rank, rt);
/* Для контроля печатаем первые четыре элемента первой строки
 * результата */
    if(rank == 0)
      { for(i = 0; i < 1; i++)
          for(j = 0; j < 4; j++)
              printf("C[i][j] = %f\n", C[i][j]);       }
/* Все ветви завершают системные процессы, связанные с топологией
 * comm_cart, и завершают выполнение программы */
    MPI_Comm_free(&comm_cart);
    MPI_Finalize();
```

```
    return(0);
}
```

Следует обратить внимание на способ "прокручивания" данных по процессорам и на формулу начального и конечного значения переменной цикла j1.

Контрольные вопросы к лабораторной работе № 3

1. Как распределяются данные матрицы при умножении матрицы на вектор?

2. Как распределяется вектор в случае распределения вектора по компьютерам?

3. Как лучше представить в памяти вторую матрицу при умножении матрицы на матрицу для ускорения времени решения задачи?

Лабораторная работа № 4
РЕШЕНИЕ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ МЕТОДОМ ГАУССА И ПРОСТОЙ ИТЕРАЦИЕЙ

Цель - освоение методов распараллеливания алгоритмов решения СЛАУ методом Гаусса и методом простой итерации. В лабораторной работе приведены три примера. Два примера связаны с разными способами решения СЛАУ методом Гаусса и один - решение СЛАУ методом простой итерации.

Рассматриваемые здесь два алгоритма решения СЛАУ методом Гаусса связаны с разными способами представления *данных* (матриц коэффициентов и правых частей) в распределенной памяти мультикомпьютера. Хотя данные распределены в памяти мультикомпьютера в каждом алгоритме по-разному, но оба они реализуются на одной и той же топологии связи компьютеров - "полный граф".

ПРИМЕР 2.11

В алгоритме, представленном в данном примере, исходная матрица коэффициентов `A` и вектор правых частей `F` разрезаны горизонтальными полосами, как показано на рис. 2.4. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер и т.д., последняя полоса – в `p1` компьютер. В примере предполагается, что матрица коэффициентов `A` и вектор правых частей `F` разрезаны на части заранее и каждая ветвь генерирует свои части матрицы и вектора правых частей.

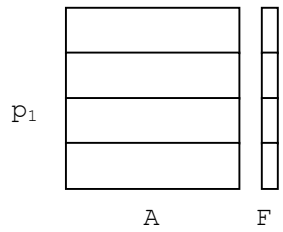


Рис. 2.4. Разрезание данных для параллельного алгоритма 1 решения СЛАУ методом Гаусса

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

После прямого хода полосы матрицы A в каждом узле будут иметь вид (рис. 2.5).

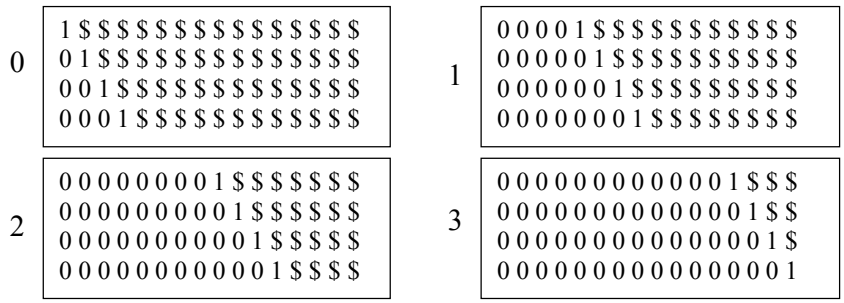


Рис. 2.5. Вид полос после прямого хода в алгоритме 1 решения СЛАУ методом Гаусса

Пример приведен для четырех узлов; $ – вещественные числа.

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

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

```
#define N 50
#define tegD 1
#define EL(x) (sizeof(x) / sizeof(x[0]))
/* Описываем массивы для полос исходной матрицы - MA и вектор V
 * для приема данных. Для простоты вектор правой части уравнений
 * присоединяем дополнительным столбцом к матрице коэффициентов.
 * В этом дополнительном столбце и получим результат. */
double MA[N][M+1], V[M+1], MAD, R;
int main(int args, char **argv)
  { int size, MyP, i, j, v, k, d, p;
    int       *index, *edges;
    MPI_Comm   comm_gr;
    MPI_Status status;
    int rt, t1,t2;
    int reord = 1;
 /* Инициализация библиотеки */
    MPI_Init(&args, &argv);
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
 /* и свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);
 /* Выделяем память под массивы для описания вершин и ребер
  * в топологии полный граф */
    index = (int *)malloc(size * sizeof(int));
    edges = (int *)malloc(size*(size-1) * sizeof(int));
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф". */
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,
                                                    &comm_gr);
/* Каждая ветвь генерирует свою полосу матрицы A и свой отрезок
 * вектора правой части, который присоединяется дополнительным
 * столбцом к A.
 * Нулевая ветвь генерирует нулевую полосу, первая ветвь - первую
 * полосу и т.д. (По диагонали исходной матрицы числа = 2,
 * остальные числа = 1). */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((N*MyP+i) == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }
 /* Каждая ветвь засекает начало вычислений и производит
    вычисления */
    t1 = MPI_Wtime();
 /* Прямой ход */
```

```
/* Цикл p - цикл по компьютерам. Все ветви, начиная с нулевой,
 * последовательно
 * приводят к диагональному виду свои строки. Ветвь, приводящая
 * свои строки к диагональному виду, назовем активной, строку,
 * с которой производятся вычисления, также назовем активной. */
   for(p = 0; p < size; p++)
     {
/* Цикл k - цикл по строкам. (Все ветви "крутят" этот цикл). */
       for(k = 0; k < N; k++)
         { if(MyP == p)
             {
/* Активная ветвь с номером MyP == p приводит свои строки к
 * диагональному виду.
 * Активная строка k передается ветвям с номером большим, чем
   MyP */
         MAD = 1.0/MA[k][N*p+k];
         for(j = M; j >= N*p+k; j--)
           MA[k][j] = MA[k][j] * MAD;
         for(d = p+1; d < size; d++)
           MPI_Send(&MA[k][0], M+1, MPI_DOUBLE, d, tegD, comm_gr);
             for(i = k+1; i < N; i++)
               { for(j = M; j >= N*p+k; j--)
                   MA[i][j] = MA[i][j]-MA[i][N*p+k]*MA[k][j];
               }
           }
/* Работа принимающих ветвей с номерами MyP > p */
          else if(MyP > p)
            {  MPI_Recv(V, EL(V), MPI_DOUBLE, p, tegD, comm_gr,
                                                    &status);
               for(i = 0; i < N; i++)
                 { for(j = M; j >= N*p+k; j--)
                     MA[i][j] = MA[i][j]-MA[i][N*p+k]*V[j];
                 }
            }
         }        /* for k */
     }            /* for p */
/* Обратный ход */
/* Циклы по p и k аналогичны, как и при прямом ходе. */
   for(p = size-1; p >= 0; p--)
     { for(k = N-1; k >= 0; k--)
         {
/* Работа активной ветви */
         if(MyP == p)
          { for(d = p-1; d >= 0; d--)
              MPI_Send(&MA[k][M], 1, MPI_DOUBLE, d, tegD, comm_gr);
             for(i = k-1; i >= 0; i--)
               MA[i][M] -= MA[k][M]*MA[i][N*p+k];
           }
/* Работа ветвей с номерами MyP < p */
          else
            { if(MyP < p)
                { MPI_Recv(&R, 1, MPI_DOUBLE, p, tegD, comm_gr,
                                                    &status);
                  for(i = N-1; i >= 0; i--)
                    MA[i][M] -= R*MA[i][N*p+k];
                }
```

```
            }
          }                  /* for k */
      }                      /* for p */
/* Все ветви засекают время и печатают */
   t2 = MPI_Wtime();
   rt = t2 - t1;
   printf("MyP = %d Time = %d\n", MyP, rt);
/* Все ветви печатают для контроля свои первые четыре значения
 * корня */
   printf("MyP = %d %f %f %f %f\n",MyP,MA[0][M],MA[1][M],
                                   MA[2][M],MA[3][M]);
/* Все ветви завершают выполнение */
   MPI_Comm_free(&comm_gr);
   MPI_Finalize();
   return(0);
 }
```

ПРИМЕР 2.12

В алгоритме, представленном в данном примере, исходная матрица коэффициентов A и вектор правых частей F разрезаны горизонтальными полосами с шириной полосы в одну строку, как показано на рис. 2.6. Каждая полоса загружается в соответствующий компьютер: нулевая полоса – в нулевой компьютер, первая полоса – в первый компьютер, и т.д., p_1-я строка в компьютер p_1 (где p_1 - количество компьютеров в системе). Затем p1+1–я строка снова помещается в компьютер 0, p1+2-я строка – в компьютер 1, и т.д. В примере предполагается, что матрица A и вектор правых частей F разрезаны на части заранее и каждая ветвь генерирует свои части матрицы.

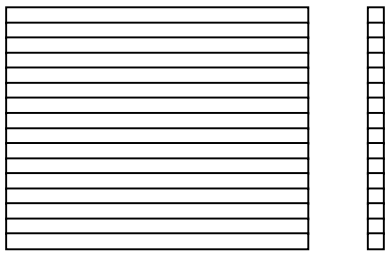


Рис. 2.6. Разрезание данных для параллельного алгоритма 2 решения СЛАУ методом Гаусса

При таком распределении данных соответствующим этому распределению должен быть и алгоритм. Строку, которая вычитается из всех остальных строк (после предварительного деления на нужные коэффициенты), назовем текущей строкой. Алгоритм прямого хода заключается в следующем. Сначала текущей строкой является строка с индексом 0 в компьютере 0, затем строка с индексом 0 в компьютере 1 (здесь не нужно путать общую нумерацию строк во всей матрице и индексацию строк в каждом компьютере; в каждом компьютере индексация строк в массиве начинается с нуля) и т.д., и наконец, строка с индексом 0

в последнем по номеру компьютере. После этого цикл по компьютерам повторяется и текущей строкой становится строка с индексом 1 в компьютере 0, затем строка с индексом 1 в компьютере 1 и т.д. После прямого хода полосы матрицы в каждом компьютере будут иметь вид, показанный на рис. 2.7. Рисунок приведен для четырех узлов; $ – вещественные числа.

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

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

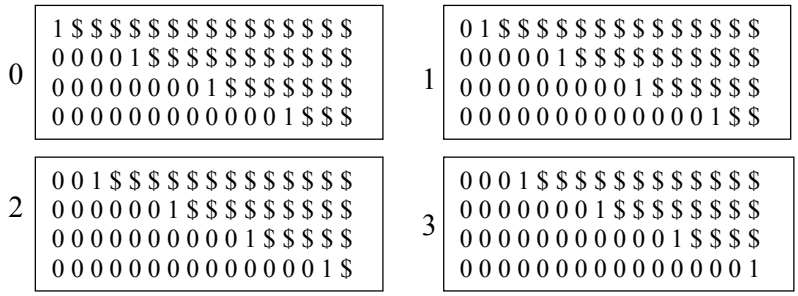


Рис. 2.7. Вид полос после прямого хода в алгоритме 2 решения СЛАУ методом Гаусса

```
/ * Решение СЛАУ методом Гаусса. Распределение данных - циклическими
  * горизонтальными полосами.
  * (Запуск задачи на восьми компьютерах).
  */
#include<stdio.h>
#include<mpi.h>
#include<sys/time.h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора
 * правой части. (Предполагаем, что размеры данных делятся без
 * остатка на количество компьютеров.) */
#define M 400
#define N 50
#define tegD 1
/* Описываем массив для циклических полос исходной матрицы - MA
 * и вектор V для приема данных. Для простоты вектор правой части
 * уравнений присоединяем дополнительным столбцом к матрице
 * коэффициентов. В этом дополнительном столбце и получим
   результат. */
double MA[N][M+1], V[M+1], MAD, R;
int main(int args, char **argv)
  { int        size, MyP, i, j, v, k, k1, p;
    int       *index, *edges;
    MPI_Comm   comm_gr;
```

```
    int rt, t1, t2;
    int reord = 1;
 /* Инициализация библиотеки */
    MPI_Init(&args, &argv);
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
/* Выделяем память под массивы для описания вершин и ребер
  * в топологии полный граф */
    index = (int *)malloc(size * sizeof(int));
    edges = (int *)malloc(size*(size-1)*sizeof(int));
 /* Заполняем массивы для описания вершин и ребер для топологии
  * полный граф и задаем топологию "полный граф". */
    for(i = 0; i < size; i++)
      { index[i] = (size - 1)*(i + 1);
        v = 0;
        for(j = 0; j < size; j++)
          { if(i != j)
              edges[i * (size - 1) + v++] = j;
          }
      }
    MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,
                                                    &comm_gr);
 /* Каждая ветвь определяет свой номер (ранг) */
    MPI_Comm_rank(MPI_COMM_WORLD, &MyP);
 /* Каждая ветвь генерирует свои циклические полосы матрицы A и свой
  * отрезок вектора правой части, который присоединяется
  * дополнительным столбцом к A.
  * Нулевая ветвь генерирует следующие строки исходной матрицы:

  * 0, size, 2*size, 3*size, и т.д. Первая ветвь - строки: 1,
  * 1+size, 1+2*size,1+3*size и т.д.
  * Вторая ветвь - строки: 2, 2+size, 2+2*size, 2+3*size и т.д.
  * (По диагонали исходной матрицы - числа = 2,
    остальные числа = 1). */
    for(i = 0; i < N; i++)
      { for(j = 0; j < M; j++)
          { if((MyP+size*i) == j)
              MA[i][j] = 2.0;
            else
              MA[i][j] = 1.0;
          }
        MA[i][M] = 1.0*(M)+1.0;
      }
 /* Каждая ветвь засекает начало вычислений и производит
    вычисления */
    t1 = MPI_Wtime();
 /* Прямой ход */
 /* Цикл k - цикл по строкам. Все ветви, начиная с нулевой,
  * последовательно приводят к диагональному виду свои строки.
  * Ветвь, приводящая свои строки к диагональному виду, назовем
  * активной, строку, с которой производятся вычисления, также
  * назовем активной. */
    for(k = 0; k < N; k++)
      {
 /* Цикл p - цикл по компьютерам. (Все ветви "крутят" этот цикл). */
        for(p = 0; p < size; p++)
```

```
        { if(MyP == p)
            {
/* Активная ветвь с номером MyP == p приводит свою строку с
 * номером k к диагональному виду.
 * Активная строка - k передается всем ветвям. */
             MAD = 1.0/MA[k][size*k+p];
             for(j = M; j >= size*k+p; j--)
               MA[k][j] = MA[k][j] * MAD;
             for(j = 0; j <= M; j++)
               V[j] = MA[k][j];
             MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
             for(i = k+1; i < N; i++)
               { for(j = M; j >= size*k+p; j--)
                   MA[i][j] = MA[i][j]-MA[i][size*k+p]*MA[k][j];
               }
            }
/* Работа принимающих ветвей с номерами MyP < p */
          else if(MyP < p)
            { MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
              for(i = k+1; i < N; i++)
                { for(j = M; j >= size*k+p; j--)
                    MA[i][j] = MA[i][j] - MA[i][size*k+p]*V[j];
                }
            }
/* Работа принимающих ветвей с номерами MyP > p */
           else if(MyP > p)
             { MPI_Bcast(V, M+1, MPI_DOUBLE, p, comm_gr);
                for(i = k; i < N; i++)
                  { for(j = M; j >= size*k+p; j--)
                     MA[i][j] = MA[i][j] - MA[i][ size*k+p]*V[j];
                  }
             }
        }          /*for p */
     }             /*for k */
/* Обратный ход */
/* Циклы по k и p аналогичны, как и при прямом ходе. */
    for(k1 = N-2, k = N-1; k >= 0; k--,k1--)
      { for(p = size-1; p >= 0; p--)
          { if(MyP == p)
              {
/* Работа активной ветви */
               R = MA[k][M];
               MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
               for(i = k-1; i >= 0; i--)
                 MA[i][M] -= MA[k][M]*MA[i][size*k+p];
              }
/* Работа ветвей с номерами MyP < p */
           else if(MyP < p)
             { MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
               for(i = k; i >= 0; i--)
                 MA[i][M] -= R*MA[i][ size*k+p];
             }
/* Работа ветвей с номерами MyP > p */
           else if(MyP > p)
             { MPI_Bcast(&R, 1, MPI_DOUBLE, p, comm_gr);
                for(i = k1; i >= 0; i--)
```

```
                    MA[i][M] -= R*MA[i][size*k+p];
                }
            }               /* for p */
        }                   /* for k */
/* Все ветви засекают время и печатают */
   t2 = MPI_Wtime();
   rt = t2 - t1;
   printf("MyP = %d Time = %d\n", MyP, rt);
/* Все ветви для контроля печатают свои первые четыре значения
 * корня */
   printf("MyP = %d %f %f %f %f\n", MyP, MA[0][M],MA[1][M],
                                        MA[2][M],MA[3][M]);
/* Все ветви завершают выполнение */
   MPI_Comm_free(&comm_gr);
   MPI_Finalize();
   return(0);
 }
```

ПРИМЕР 2.13

Здесь рассматривается параллельный алгоритм решения СЛАУ методом простой итерации. Приближенные решения (итерации) системы линейных уравнений последовательно находятся по формуле (1).

$$y_{k+1}^{(i)} = y_k^{(i)} - t(\sum_{j=1}^{N} a_{i,j} y_k^{(i)} - f_k^{(i)}),...i=1,2,...N.....(1)$$

Для решения этой задачи на параллельной системе исходную матрицу коэффициентов A разрезаем на p_1 горизонтальные полосы по строкам, как показано на рис. 2.3, где p_1 – количество компьютеров в системе. Аналогично, горизонтальными полосами разрезаются вектор b (правая часть) и вектора y_0 (начальное приближение), y_k (текущее приближение) и y_{k+1} (следующее приближение). Полосы последовательно распределяются по соответствующим компьютерам системы, как и в описанном выше первом алгоритме умножения матрицы на вектор.

Здесь выражение $\sum_{j=1}^{N} ai, jy_k^{(i)}$ есть умножение матрицы на вектор, параллельный алгоритм которого представлен в примере 7. Таким образом, этот алгоритм является составной частью описываемого в данном пункте алгоритма. В каждом компьютере системы вычисляется "свое" подмножество корней. Поэтому после нахождения приближенных значений корней на очередном шаге итерации в каждом компьютере проверяется выполнение следующего условия для "своих" подмножеств корней:

$$\| y_{k+1}^{(i)} - y_k^{(i)} \| \le e.....(2)$$

Это условие в некоторых компьютерах системы в текущий момент может выполняться, а в некоторых нет. Но условием завершения работы каждого компьютера является обязательное выполнение условия (2) во всех компьютерах. Таким образом, прежде чем завершить работу, при выполнении условия (2) каждый компьютер должен предварительно узнать, во всех ли компьютерах выполнилось условие (2). И если условие (2) не выполнилось хотя бы в одном компьютере, то все компьютеры должны продолжить работу. Это обстоятельство связано с тем, что в операции умножения матрицы на вектор участвуют все компьютеры, взаимодействуя друг с другом. И цепочку этих взаимодействий прерывать нельзя, если хотя бы в одном из компьютеров не выполнится условие (2). При неыполнении условия (2) каждый процесс передает всем остальным процессам полученную итерацию своих корней. И тем самым вектор Y полностью восстанавливается в каждом процессе для выполнения операции его умножения на матрицу коэффициентов на следующем шаге итерации.

```
/*
 * Решение СЛАУ методом простой итерации. Распределение данных -
 * горизонтальными полосами. (Запуск задачи на четырех компьютерах).
 */
#include<stdio.h>
#include<mpi.h>
#include<time.h>
#include<sys/time.h>
/* Каждая ветвь задает размеры своих полос матрицы MA и вектора
 * правой части.
 * (Предполагаем, что размеры данных делятся без остатка
 * на количество компьютеров.) */
#define M 16
#define N 4
#define EL(x) (sizeof(x) / sizeof(x[0]))
#define ABS(X) ((X) < 0 ? -(X) : (X))
/* Задаем необходимую точность приближенных корней */
#define E 0.0001
/* Задаем шаг итерации */
#define T 0.1
/* Описываем массивы для полос исходной матрицы - MA, вектора
 * правой части - F, значения приближений на предыдущей
 * итерации - Y и текущей - Y1, результата умножения матрицы
 * коэффициентов на вектор - S и всего вектора значения
 * приближений на предыдущей итерации - V. */
static double MA[N][M], F[N], Y[N], Y1[N], S[N], V[M];
int main(int argc, char **argv)
  { int         i, j, z, H, MyP, size, v;
    int        *index, *edges;
    MPI_Comm    comm_gr;
    int rt, t1 t2;              /* Для засечения времени */
    int reord = 1;
 /* Инициализация библиотеки */
    MPI_Init(&argc, &argv);
 /* Каждая ветвь узнает размер системы */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
/* Выделяем память под массивы для описания вершин и ребер
 * в топологии полный граф */
   index = (int *)malloc(size * sizeof(int));
   edges = (int *)malloc(size*(size-1)*sizeof(int));
/* Заполняем массивы для описания вершин и ребер для топологии
 * полный граф и задаем топологию "полный граф". */
   for(i = 0; i < size; i++)
     { index[i] = (size - 1)*(i + 1);
       v = 0;
       for(j = 0; j < size; j++)
         { if(i != j)
             edges[i * (size - 1) + v++] = j;
         }
     }
   MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, reord,
                                                    &comm_gr);
/* каждая ветвь определяет свой номер (ранг) */
   MPI_Comm_rank(comm_gr, &MyP);
/* Каждая ветвь генерирует свои полосы матрицы A и свой отрезок
 * вектора правой части.
 * (По диагонали исходной матрицы - числа = 2,
   остальные числа = 1). */
   for(i = 0; i < N; i++)
     { for(j = 0; j < M; j++)
         { if((N*MyP + i) == j)
             MA[i][j] = 2.0;
           else
             MA[i][j] = 1.0;
         }
       F[i] = M + 1;
     }
/* Каждая ветвь засекает начало вычислений и производит
   вычисления */
   t1 = MPI_Wtime();
/* Каждая ветвь задает начальное приближение корней. */
   for(i = 0; i < N; i++)
     Y1[i] = 0.8;
/* Начало вычислений */
   do
     { for(i = 0; i < N; i++)
         { S[i] = 0.0;
           Y[i] = Y1[i];
         }
/* В каждой ветви формируем весь вектор предыдущей итерации
 * и умножаем матрицу коэффициентов на этот вектор */
       MPI_Allgather(Y, EL(Y), MPI_DOUBLE, V, EL(Y), MPI_DOUBLE,
                                                    comm_gr);
       for(j = 0; j < N; j++)
         for(i = 0; i < M; i++)
           S[j] += MA[j][i] * V[i];
       z = 0;        /* Флаг завершения вычислений всеми ветвями */
       for(i = 0; i < N; i++)
         { Y1[i] = Y[i] - T*(S[i] - F[i]);
           if(ABS(ABS(Y1[i]) - ABS(Y[i])) > E)
           z = 1;
         }
```

```
/* Суммируем все флаги (по всем ветвям) и результат записываем в H
 * в каждой ветви */
      MPI_Allreduce(&z, &H, 1, MPI_INT, MPI_SUM, comm_gr);
    }
   while(H > 0);
/* Все ветви засекают время и печатают */
   t2 = MPI_Wtime();
   rt = t2 - t1;
   printf("MyP = %d Time = %d\n", MyP, rt);
/* Все ветви для контроля печатают свои первые четыре значения
 * корня */
   printf("Rez MyP = %d Y0=%f Y1=%f Y2=%f Y3=%f\n", MyP,Y[0],Y[1],
                                                    Y[2],Y[3]);
/* Все ветви завершают выполнение */
   MPI_Comm_free(&comm_gr);
   MPI_Finalize();
   return(0);
 }
```

Контрольные вопросы к лабораторной работе № 4

1. В каком из приведенных алгоритмов Гаусса вычислительная нагрузка более равномерно распределена по компьютерам и почему?

Лабораторная работа № 5
ПРОИЗВЕДЕНИЕ ДВУХ МАТРИЦ В ТОПОЛОГИИ "ДВУМЕРНАЯ РЕШЕТКА"

Цель - закрепление знаний раздела конструирования MPI–типов данных. В этом примере исходные и результирующая матрицы разрезаются в нулевой ветви, и затем части матриц рассылаются во все другие ветви. Схема распределения данных по компьютерам приведена ниже.

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

Последовательные стадии вычисления иллюстрируются на рис. 2.8.

1. Матрица `A` распределяется по горизонтальным полосам вдоль координаты `(x,0)`.

2. Матрица `B` распределяется по вертикальным полосам вдоль координаты `(0,y)`.

3. Полосы `A` распространяются в измерении `y`.

4. Полосы `B` распространяются в измерении `x`.

5. Каждый процесс вычисляет одну подматрицу произведения.

6. Матрица C собирается из (x,y) плоскости.

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

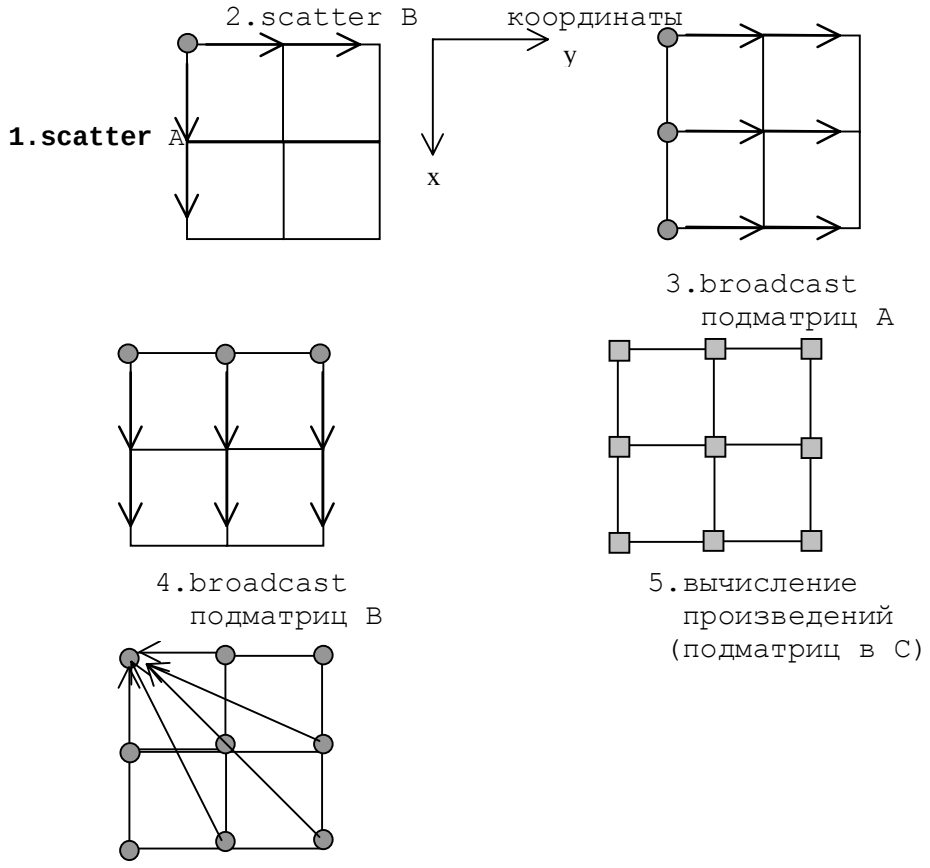


Рис. 2.8. Стадии вычисления произведения матриц в 2D параллельном алгоритме

ПРИМЕР 2.14

```
/* Произведение двух матриц в топологии "двумерная решетка"
   компьютеров */
/* В примере предполагается, что количество строк матрицы A и
 * количество столбцов матрицы B делятся без остатка на количество
 * компьютеров в системе. В данном примере задачу запускаем
   на четырех компьютерах и на решетке 2x2. */
   #include<stdio.h>
   #include<stdlib.h>
   #include<mpi.h>
   #include<time.h>
   #include<sys/time.h>
/* NUM_DIMS - размер декартовой топологии. "двумерная решетка"
   0xP1 */
   #define NUM_DIMS 2
   #define P0 2
```

```
   #define P1 2
/* Задаем размеры матриц A = MxN, B = NxK и C = MxK (Эти размеры
 * значимы в ветви 0)
 */
   #define M 8
   #define N 8
   #define K 8
   #define A(i,j) A[N*i+j]
   #define B(i,j) B[K*i+j]
   #define C(i,j) C[K*i+j]

/* Подпрограмма, осуществляющая перемножение матриц */

PMATMAT_2(n, A, B, C, p, comm)
 /* Аргументы A, B, C, n, p значимы в данном случае только
    в ветви 0 */
    int *n;                          /* Размеры исходных матриц */
    double *A, *B, *C;     /* Исходные матрицы: A[n[0]][n[1]],
                                                B[n[1]][n[2]],
                                                C[n[0]][n[2]]; */
 /* Данные */
    int *p;
 /* размеров решетки компьютеров. p[0] соответствует n[0], p[1]
  * соответствует n[2] и произведение p[0]*p[1] будет эквивалентно
  * размеру группы comm */
 /* Коммуникатор для процессов, участвующих в умножении матрицы */
    на матрицу MPI_Comm comm;

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

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

    MPI_Datatype typeb, typec, types[2];

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

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

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

```
   MPI_Bcast(n, 3, MPI_INT, 0, pcomm);
   MPI_Bcast(p, 2, MPI_INT, 0, pcomm);

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

/* Нахождение коммуникаторов для подрешеток 1D для рассылки полос
 * матриц A и B */
   for(i = 0; i < 2; i++)
     { for(j = 0; j < 2; j++)
         remains[j] = (i == j);
         MPI_Cart_sub(comm_2D, remains, &comm_1D[i]);
     }
/* Во всех ветвях задаем подматрицы (полосы) */
/* Здесь предполагается, что деление без остатка */
   nn[0] = n[0]/p[0];
   nn[1] = n[2]/p[1];

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

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

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

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

    /* Вычисление размера подматрицы BB и смещений каждой
     * подматрицы в матрице B. Подматрицы BB упорядочены в B
     * в соответствии с порядком номеров компьютеров в решетке,
     * так как массивы расположены в памяти по строкам, то
```

```
      * подматрицы BB в памяти (в B) должны располагаться
      * в следующей последовательности:  BB0, BB1,.... */
        dispb =  (int *)malloc(p[1] * sizeof(int));
        countb = (int *)malloc(p[1] * sizeof(int));
        for(j = 0; j < p[1]; j++)
          { dispb[j] = j;
            countb[j] = 1;
          }

    /* Задание типа данных для подматрицы CC в C */
        MPI_Type_vector(nn[0], nn[1], n[2], MPI_DOUBLE, &types[0]);
    /* и корректируем размер диапазона */
        MPI_Type_struct(2, blen, disp, types, &typec);
        MPI_Type_commit(&typec);
     /* Вычисление размера подматрицы CC и смещений каждой
      * подматрицы в матрице C. Подматрицы CC упорядочены в C
      * в соответствии с порядком номеров компьютеров в решетке,
      * так как массивы расположены в памяти по строкам, то подматрицы
      * CC в памяти (в C) должны располагаться в следующей
      * последовательности: CC0, CC1, CC2, CC3, CC4, CC5, CC6, CC7. */
        dispc =  (int *)malloc(p[0] * p[1] * sizeof(int));
        countc = (int *)malloc(p[0] * p[1] * sizeof(int));
        for(i = 0; i < p[0]; i++)
          { for(j = 0; j < p[1]; j++)
              { dispc[i*p[1]+j] = (i*p[1]*nn[0] + j);
                countc[i*p[1]+j] = 1;
              }
          }
      }         /* Нулевая ветвь завершает подготовительную работу */

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

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

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

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

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

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

    for(i = 0; i < nn[0]; i++)
      { for(j = 0; j < nn[1]; j++)
```

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

 /* 6. Сбор всех подматриц CC в ветви 0 */
    MPI_Gatherv(CC, nn[0]*nn[1], MPI_DOUBLE, C, countc, dispc, typec,
                                             0, comm_2D);

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

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

    return 0;
  }

 /* Главная программа */

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

```
   MPI_Cart_create(MPI_COMM_WORLD, NUM_DIMS, dims, periods, reorder,
   &comm);
/* В первой ветви выделяем в памяти место для исходных матриц */
   if(MyP == 0)
     {
    /* Задаем размеры матриц и размеры двумерной решетки
       компьютеров */
       n[0] = M;
       n[1] = N;
       n[2] = K;
       p[0] = P0;
       p[1] = P1;

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

    /* Генерируем в первой ветви исходные матрицы A и B, матрицу C
       обнуляем */
       for(i = 0; i < M; i++)
         for(j = 0; j < N; j++)
           A(i,j) = i+1;
       for(j = 0; j < N; j++)
         for(k = 0; k < K; k++)
           B(j,k) = 21+j;
       for(i = 0; i < M; i++)
         for(k = 0; k < K; k++)
           C(i,k) = 0.0;
     }           /* Подготовка матриц ветвью 0 завершена */
/* Засекаем начало умножения матриц во всех ветвях */
   gettimeofday(&tv1, (struct timezone*)0);
/* Все ветви вызывают функцию перемножения матриц */
   PMATMAT_2(n, A, B, C, p, comm);
/* Умножение завершено. Каждая ветвь умножила свою полосу строк
 * матрицы A на полосу столбцов матрицы B. Результат находится
 * в нулевой ветви. Засекаем время и результат печатаем */
   gettimeofday(&tv2, (struct timezone*)0);
   dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec -
   tv1.tv_usec; printf("MyP = %d Time = %d\n", MyP, dt1);
/* Для контроля 0-я ветвь печатает результат */

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

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

   if(MyP == 0)
     { free(A);
       free(B);
       free(C);
```

```
    }

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

Контрольные вопросы к лабораторной работе № 5

1. Какой из приведенных алгоритмов перемножения матриц (в лабораторных работах № 3 и № 5) более эффективен и почему?

ОСНОВНЫЕ ТЕРМИНЫ

АСИНХРОННЫЙ ОБМЕН ДАННЫМИ - буферизованный обмен данными между процессами. Передающий процесс помещает посылаемые данные в специальный буфер. Принимающий процесс берет эти данные из буфера (после их поступления) по мере необходимости или переходит в состояние ожидания, если нужные данные не поступили.

СИНХРОННЫЙ ОБМЕН ДАННЫМИ - небуферизованный обмен данными между процессами. Передающий и принимающий (принимающие) процесс(ы) осуществляет обмен данными только после их выхода на соответствующие команды обмена данными.

ВИРТУАЛЬНЫЙ - не имеющий физического воплощения или воспринимаемый иначе, чем реализован.

ВИРТУАЛЬНЫЙ КАНАЛ. В сети коммутации пакетов - средства, обеспечивающие передачу пакетов между двумя узлами с сохранением исходной последовательности, даже если пакеты пересылаются по разным физическим маршрутам.

ТОПОЛОГИЯ СЕТИ - структура сети связи с учетом дисциплины соединений узлов. Примерами форм топологии сети являются: звезда, кольцо, решетка, тор, гиперкуб, дерево и т.д.

ВИРТУАЛЬНАЯ ТОПОЛОГИЯ - совокупность ресурсов, которые эмулируют поведение реальной топологии сети, например: звезды, кольца, решетки, тора и т.д.

ПРОЦЕСС - логическая единица, определяемая некоторым программным кодом, назначенным на выполнение определенному процессору, участком памяти, содержащим входные и промежуточные данные (контекстом процесса) и дескриптором процесса, содержащим информацию о состоянии выделенных процессу ресурсов.

MIMD-система - (от Multiple Instruction Multiple Data) - архитектура параллельной ЭВМ с несколькими потоками команд и несколькими потоками данных. Организация вычислительной системы с несколькими однородными или разнородными процессорами, каждый из которых выполняет свои команды над своими данными.

СИСТЕМА С ПЕРЕДАЧЕЙ СООБЩЕНИЙ - MIMD-система, состоящая из нескольких процессоров, каждый из которых имеет доступ только к своей локальной памяти, и коммуникационной сети, объединяющей эти процессоры в единую систему. Обмен данными между процессорами системы происходит с помощью посылки сообщений по коммуникационным каналам.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. *Snir M., Otto S. W., Huss-Lederman S., Walker D., and Dongarra J.* MPI: The Complete Reference. MIT Press. Boston, 1996.
2. Малышкин В.Э., Вшивков В.А., Краева М.А. О реализации метода частиц на мультипроцессорах. Новосибирск, 1995. 37 с. (Препринт / РАН. Сиб. отдние. ВЦ; 1052).
3. Евреинов Э.В., Косарев Ю.Г. Однородные универсальные вычислительные системы высокой производительности. Новосибирск, 1966. 308 с.
4. Миренков Н.Н. Параллельное программирование для многомодульных вычислительных систем. М., 1989. 320 с.
5. Малышкин В.Э. Линеаризация массовых вычислений // Системная информатика / Под ред. В.Е.Котова. Новосибирск, 1991. № 1. С. 229–259.
6. Корнеев В.Д. Система и методы программирования мультикомпьютеров на примере вычислительного комплекса PowrXplorer. Новосибирск, 1998. 56 с. (Препринт / РАН. Сиб. отд-ние. ИВМиМГ; 1123).
7. Корнеев В.Д. Параллельные алгоритмы решения задач линейной алгебры. Новосибирск, 1998. 27 с. (Препринт / РАН. Сиб. отд-ние. ИВМиМГ; 1124).
8. Корнеев В.Д**.** Параллельное программирование в MPI. Новосибирск, 2000. 220 с.
9. *Dongarra J., Otto S. W., Snir M., and Walker D.,* An Introduction to the MPI Standard. Technical report CS-95-274. University of Tennessee, January 1995.

ОГЛАВЛЕНИЕ

В.Д. Корнеев

Параллельное программирование в MPI

Редактор, корректор В.Н. Чулкова
Компьютерная верстка И.Н. Ивановой

Подписано в печать 27.11.2002. Формат 60x84/16. Бумага тип.
Усл. печ. л. 6,04. Уч.-изд. л. 5,2. Тираж 150 экз. Заказ

Оригинал-макет подготовлен
в редакционно-издательском отделе ЯрГУ

Отпечатано на ризографе.

Ярославский государственный университет
150000 Ярославль, ул. Советская, 14.