С.А. Лупин, М.А. Посыпкин

Технологии параллельного программирования

Допущено Учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению 230100 «Информатика и вычислительная техника»

Москва
ИД «ФОРУМ» — ИНФРА-М
2011

УДК 004(075.8)
ББК 32.973я73
Л85

Рецензенты:

проректор МИЭТ по научной работе, зав.кафедрой «Вычислительная техника» д.т.н., профессор *В. А. Бархоткин*;
первый заместитель директора Межведомственного Суперкомпьютерного центра, к.т.н. *Б. М. Шабанов*

Лупин С.А., Посыпкин М.А.

Л85 Технологии параллельного программирования. — М.: ИД «ФОРУМ»: ИНФРА-М, 2011. — 208 с. — (Высшее образование).

ISBN 978-5-8199-0336-0 (ИД «ФОРУМ»)
ISBN 978-5-16-003155-2 (ИНФРА-М)

Рассматриваются современные средства разработки параллельных программ для многопроцессорных и многоядерных систем с общей и распределенной памятью: библиотеки MPI, POSIX Threads, система OpenMP. Изложение материала построено по модели постепенного усложнения и базируется на примерах реализации различных вычислительных алгоритмов.

Книга предназначена для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника» и изучающих дисциплины «Основы параллельного программирования», и преподавателей указанных дисциплин. Кроме того, книга может быть полезна специалистам в области разработки приложений для многопроцессорных вычислительных систем.

УДК 004(075.8)
ББК 32.973я73

ISBN 978-5-8199-0336-0 (ИД «ФОРУМ»)
ISBN 978-5-16-003155-2 (ИНФРА-М)

© ИД «ФОРУМ», 2008
© Лупин С.А., Посыпкин М.А., 2008

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Последние десятилетия суперкомпьютеры находят свое применение при решении практически любых задач науки и техники. Среди таких задач — моделирование различных физических процессов, проблемы вычислительной химии и биологии, нанотехнологии, автоматизация проектирования и многие другие. Прогресс в области высокопроизводительных вычислений во многом определяет темп развития науки и техники, и, в конечном итоге, уровень технологического развития страны в целом. Поэтому можно с уверенностью утверждать, что создание и изучение методов разработки программ для суперкомпьютеров *является важнейшей областью современных* информационных технологий.

Чрезвычайно важным является подготовка специалистов в данной области. Для этого в программу многих высших учебных заведений введен курс «Параллельное программирование». Параллельное программирование является обобщающим термином, применяемым для обозначения технологий и методов разработки программ для суперкомпьютеров. Преподавание этой дисциплины на высоком уровне требует объединения усилий образовательных учреждений и крупнейших научных центров, в которых ведутся исследования в области параллельных вычислений, обладающих необходимыми вычислительными ресурсами мирового уровня. Такое сотрудничество было организовано Московским институтом электронной техники и крупнейшим российским вычислительным центром — Межведомственным суперкомпьютерным центром РАН. В течение шести последних лет студенты МИЭТ проходят обучение на базе МСЦ РАН. Одним из основных курсов программы обучения является курс «Параллельное программирование». Студенты посещают лекции специалистов МСЦ РАН и выполняют лабораторные работы под их руководством. Участие сотрудников МСЦ РАН дает возможность студентам осваивать передовые технологии параллельного программирования, получать представление о последних достижениях в данной области. Исключительно важной является также возмож-

ность апробации полученных знаний на современных образцах вычислительной техники, установленных в МСЦ РАН.

Данный учебник написан на основе лекций и практических занятий, проводимых в МСЦ РАН для студентов МИЭТ. В него вошли наиболее распространенные технологии и методы разработки параллельных приложений, освоение которых является необходимым условием для успешной работы в области создания программ для суперкомпьютеров.

Ректор МИЭТ,
член-корреспондент РАН,
д.т.н., профессор Чаплыгин Юрий Александрович

Первый заместитель директора
Межведомственного суперкомпьютерного центра РАН,
Лауреат государственной премии Российской Федерации,
к.т.н. Шабанов Борис Михайлович

ВВЕДЕНИЕ

Термин *параллельное программирование* означает достаточно широкую область, которая связана с организацией расчетов на вычислительных системах, состоящих из нескольких процессорных устройств. К таким системам относятся получившие в последнее время распространение многоядерные процессоры, многопроцессорные машины с общей памятью, высокопроизводительные вычислительные кластеры с распределенной памятью или гибридной архитектурой.

Параллельным вычислениям в последнее время уделяется большое внимание. Это связано главным образом с двумя факторами. Первый фактор обусловлен научно-техническим прогрессом, в результате которого появились новые области знаний, требующие применения методов математического моделирования. Сами модели также существенно усложнились. В итоге происходит неуклонное возрастание потребности в ресурсоемких расчетах, которые, в ряде случаев можно выполнить только на базе высокопроизводительной техники с помощью методов параллельных или распределенных вычислений.

Другой существенный фактор, в результате которого интерес к параллельным вычислениям существенно вырос, состоит в повсеместном распространении параллельных компьютеров. В последнее время многопроцессорные серверы можно часто встретить на средних и крупных предприятиях, в банках, исследовательских институтах и центрах. В связи с появлением многоядерных процессоров многие пользователи стали обладателями мини-суперкомпьютеров на своих рабочих местах.

Существенный прогресс в области сетевых технологий позволил использовать для параллельных вычислений локальные сети предприятий, учебные классы, сделал возможным создание дешевых вычислительных кластеров.

В итоге можно с уверенностью утверждать, что параллельные информационные технологии превратились из узкоспециальной дисциплины в необходимую составляющую комплекса знаний разработчика современного программного обеспечения.

Области применения параллельных вычислений

Параллельные вычисления применяются в областях, связанных с проведением больших расчетов:

- системах поддержки проектирования (CAD — Computer Aided Design). В таких системах необходимость осуществлять моделирование в реальном масштабе времени предъявляет высокие требования к производительности программного обеспечения. В результате применения параллельных информационных технологий удается существенно ускорить процесс проектирования и тем самым заметно снизить временные и трудовые затраты на разработку новой модели;
- инженерных приложениях. К этому классу относятся разнообразные задачи из области прочностного моделирования, моделирования аварийных ситуаций и многие другие;
- математическом моделировании физических процессов. В этот широкий класс входят задачи динамики жидкости и газа, электромагнитные и ядерные взаимодействия, процессы горения и т. п. Такие процессы, как правило, описываются системами уравнений в частных производных. Применяемые для решения таких задач разностные методы нередко требуют очень большого объема вычислений и памяти. Использование многопроцессорных систем позволяет повышать число улов сетки, тем самым увеличивая точность моделирования;
- моделировании глобальных процессов в науке о Земле. В первую очередь, это — задачи прогноза изменения климата, предсказание природных катаклизмов. Также большой вычислительной сложностью обладают различные геологические проблемы, связанные с анализом строения и процессов в недрах;
- вычислительной химии. Разнообразные задачи этой области направлены на изучение свойств вещества в различных состояниях. Широко применяемые методы молекулярной динамики также зачастую требуют существенных вычислительных ресурсов, что оправдывает применение параллельного программирования. К данной категории можно также отнести задачи, связанные с оптимальной конфигурацией протеинов, расшифровки ДНК и многие другие проблемы смежных с химией областей;
- бизнес-приложениях. К этой категории относятся задачи, связанные с анализом финансовых рынков и прогнозированием

курсов валют. Также распространены оптимизационные задачи, по формированию наилучшего варианта использования финансовых или иных ресурсов, построения оптимальных транспортных и телекоммуникационных сетей, размещения предприятий в регионе и многие другие задачи.

Мы привели лишь некоторые из многочисленных применений параллельного программирования. Следует также отметить, что список сфер применения параллельного программирования неуклонно расширяется последнее время.

Краткий обзор архитектуры параллельных систем

Идея параллельной обработки с целью ускорения процесса вычислений появилась достаточно давно, поэтому параллельная вычислительная техника развивается уже не один десяток лет. За этот период было создано много различных типов архитектур, подробное описание которых выходит за рамки нашего пособия. Обширная информация по этому вопросу содержится в монографиях [6, 11]. Мы же ограничимся только наиболее общей классификацией.

Общим для различных типов параллельных архитектур является наличие нескольких процессорных устройств. По типу взаимодействия процессорных устройств между собой параллельные системы подразделяются на системы с общей и распределенной памятью. В системах с общей памятью все процессоры имеют доступ к единому пространству памяти. Обмен данными между процессорами производится следующим образом: один процессор записывает данные по некоторому адресу, а другой считывает их. К этому классу относятся многопроцессорные рабочие станции и сервера, суперкомпьютеры с общей памятью (например HP Superdome), а также получившие широкое распространение в последнее время многоядерные процессоры.

Системы с общей памятью удобны с точки зрения разработки параллельных программ и обеспечивают высокую производительность, но при большом числе процессоров они очень дороги. Менее затратной альтернативой являются системы с распределенной памятью. В таких системах каждый процессор имеет доступ только к своей локальной памяти, а между собой процессоры

взаимодействуют с помощью передачи сообщений по сети. К этому классу относятся высокопроизводительные вычислительные кластеры, а также локальные сети. Не редки также ЭВМ с гибридной архитектурой, в которой узлы с общей памятью соединяются посредством сетевого оборудования.

Различия в архитектуре предопределяют и различия в парадигмах программирования. Для систем с общей памятью основной парадигмой является многопоточное программирование, при котором различные потоки управления осуществляют доступ к единому пространству памяти. Классическими инструментами для таких систем являются библиотека pthreads и пакет OpenMP.

Наиболее распространенным способом создания параллельных приложений для систем с распределенной памятью является организация процессов, взаимодействующих при помощи передачи сообщений. На этой парадигме основаны библиотеки MPI (Message Passing Interface), PVM (Parallel Virtual Machine) и многие другие, менее известные средства программирования.

Структура и целевая аудитория пособия

Учебное пособие предназначено для студентов, преподавателей, а также специалистов, чья область образовательной или профессиональной деятельности связана с тематикой параллельного программирования. Пособие содержит большое количество примеров и ориентировано, в первую очередь, на формирование практических навыков разработки программ.

Рассматриваются три среды, которые являются основными средствами создания параллельного программного обеспечения в настоящее время:

- библиотека MPI (гл. 1);
- библиотека POSIX Threads (гл. 2);
- среда программирования OpenMP (гл. 3).

В пособии есть три приложения, содержащие справочную информацию, полезную при разработке параллельных приложений.

Авторы выражают благодарность руководству Межведомственного суперкомпьютерного центра Российской академии наук, директору МСЦ РАН академику РАН Г.И. Савину и первому заместителю директора Б.М. Шабанову за возможность апробации курса на базовых кафедрах Московского физико-технического института и Московского института электронной техники и внимание к содержанию лекций и учебника. Авторы благодарны руководителю направления «Распределенные вычисления» в Институте системного анализа РАН, профессору А.П. Афанасьеву за плодотворные обсуждения материалов курса по параллельным вычислениям, ведущему научному сотруднику вычислительного центра РАН профессору И.Х. Сигалу и профессору университета г. Дублина А.Л. Ластовецкому.

1. ПАРАЛЛЕЛЬНЫЕ ПРОГРАММЫ НА ОСНОВЕ ПЕРЕДАЧИ СООБЩЕНИЙ

1.1. Параллельные процессы, взаимодействующие с помощью передачи сообщений

Наиболее распространенной технологией, применяемой для программирования многопроцессорных систем с распределенной памятью, является использование параллельных процессов, взаимодействующих с помощью передачи сообщений. Такая модель представляется также и наиболее естественной для подобных систем, так как передача сообщений по сети — практически единственный возможный способ взаимодействия процессов, выполняющихся на различных процессорах, не обладающих общей памятью.

Основным средством разработки программ в рассматриваемой парадигме является MPI (Message-Passing Interface) [17, 15]. В настоящее время MPI входит в стандартный комплект программного обеспечения практически любого многопроцессорного вычислительного комплекса.

В состав среды программирования MPI входит библиотека с интерфейсом для одного или нескольких языков программирования (обычно Fortran, Си и Си++), а также средства для запуска и сборки параллельного приложения. Интерфейс библиотеки соответствует стандарту версии 1.1, принятому в 1995 году или более позднему варианту версии 2.0, принятому в 1998 году. Оба документа доступны на сайте сообщества разработчиков стандарта [15]. Следует отметить, что подавляющее большинство MPI-приложений соответствуют стандарту версии 1.1, так как стандарт 2.0 доступен далеко не на всех параллельных системах.

1.2. Простейшая MPI-программа

Традиционно освоение новой системы программирования начинается с написания программы, печатающей «Hello, World!».

Мы тоже начнем изучение MPI с аналогичного примера, текст которого приведен на листинге 1.

```
1:  #include <stdio.h>
2:  #include <mpi.h>
3:  main(int argc, char* argv[])
4:  {
5:    MPI_Init(&argc, &argv);
6:    printf("Hello, World!\n");
7:    MPI_Finalize();
8:  }
```

Листинг 1. Простейшая MPI-программа

Рассмотрим подробно содержание этого примера. В строке 1 подключается заголовочный файл стандартной библиотеки ввода-вывода языка Си. В строке 2 подключается заголовочный файл `mpi.h`, содержащий определения функций, типов и констант MPI. Этот файл необходимо включать во все модули, использующие MPI.

Как и в любой программе языка Си, в строке 3 расположен заголовок функции `main`, которая является точкой входа в программу. Вызов функции `MPI_Init` в строке 5 нужен для инициализации MPI, а функция `MPI_Finalize` в строке 7 вызывается для завершения работы с библиотекой. Единственное содержательное действие, производимое рассматриваемой программой, состоит в выполнении оператора печати в строке 6.

Функция `MPI_Finalize` не имеет параметров. Функция `MPI_Init` принимает на вход адреса переменных, соответствующих аргументам командной строки.

```
int MPI_Init(int* pargc, char*** pargv)
  pargc – указатель на счетчик аргументов командной строки;
  pargv – указатель на массив аргументов командной строки.

int MPI_Finalize()
```

Перед тем как выполнить написанный пример, необходимо произвести его компиляцию с помощью следующей команды:

```
mpicc -o myex myex.c
```

В результате по файлу myex.c, содержащему текст программы, генерируется выполняемый файл myex, который можно запускать с помощью команды mpirun. Команда mpirun имеет несколько параметров, которые могут существенно отличаться для различных систем. Подробно способы запуска программы в различных версиях MPI описаны в приложении 1. На большинстве систем MPI-программу можно запустить, указав число запускаемых экземпляров программы с помощью опции -np, например, команда

```
mpirun -np 2 myex
```

запустит два экземпляра программы myex. Если выполнить указанную команду, то на терминал будет выведен следующий текст:

```
Hello, world!
Hello, world!
```

После выполнения команды mpirun было запущено два процесса, каждый из которых выполнил оператор печати, в результате дважды была напечатана фраза Hello, world!. Заметим, что все процессы выполняют одну и ту же программу, текст которой приведен на листинге 1.

Этот пример иллюстрирует модель SPMD (Single Program Multiple Data — «одна программа, разные данные»), которая является основной концепцией выполнения MPI-программы. Суть парадигмы состоит в том, что все задействованные процессоры вычислительной системы выполняют одну и ту же MPI-программу, которая может обрабатывать различные данные и выполняться по различным путям на разных процессорах. В рассмотренном примере данные и путь выполнения у разных экземпляров программы не различаются.

Модель SPMD является удобной с точки зрения разработки параллельных программ, и, хотя следование этой модели не является обязательным требованием MPI, большинство MPI-приложений на сегодняшний день ей соответствуют.

В приведенном примере вывод на окно терминала не позволяет определить, какому процессу принадлежит любая из трех строк. Следующий вариант печатает перед каждой строкой пре-

фикс, соответствующий общему числу процессов и номеру процесса, выполняющего оператор печати:

```
1:  #include <stdio.h>
2:  #include <mpi.h>
3:  main(int argc, char* argv[])
4:  {
5:    int rank;
6:    int size;
7:    MPI_Init(&argc, &argv);
8:    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9:    MPI_Comm_size(MPI_COMM_WORLD, &size);
10:   printf("%d, %d: Hello, World!\n",
        size, rank);
11:   MPI_Finalize();
12:  }
```

Листинг 2. MPI-программа с печатью номеров и общего числа процессов

От предыдущего варианта данный пример отличается вызовами функций MPI_Comm_rank и MPI_Comm_size в строках 8–9. В результате выполнения этого вызова переменная rank получает значение, соответствующее номеру, которой присваивается процессу системой, а в переменную size будет записано общее число процессов приложения:

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

comm – коммуникатор;

size – указатель на область памяти, в которую будет сохранено общее число процессов в коммуникаторе comm;

rank – указатель на область памяти, в которую будет сохранен номер процесса, вызывающего функцию в коммуникаторе comm.

Первый параметр в вызове этих функций называется *коммуникатором* и служит для обозначения совокупности процессов, по отношению к которой вычисляется число процессор и номер

процесса, выполнившего вызов функции. В приведенном примере используется предопределенный коммуникатор MPI_COMM_WORLD, соответствующий всем процессам параллельной программы. Более подробно коммуникаторы будут рассмотрены в следующих разделах.

В результате выполнения данного примера на терминал будет вероятнее всего выведен один из двух вариантов:

```
   Вариант 1.          Вариант 2.
2,0: Hello, World!   2,1: Hello, World!
2,1: Hello, World!   2,0: Hello, World!
```

Очередность появления текстовых сообщений определяется тем, какой из процессов раньше выполнит оператор печати, что, вообще говоря, сложно предсказать заранее. В некоторых системах возможен, хотя и менее вероятен, вариант, при котором сообщения от обоих процессов будут смешаны произвольным образом. В этом случае воспринимать полученный текст будет значительно сложнее.

Понятно, что в более сложных программах, недетерминизм, возникающий при выводе на терминал или в файл, может вызывать серьезные затруднения при восприятии результатов работы программы. Одно из возможных решений этой проблемы, о котором пойдет речь в следующем разделе, заключается в том, чтобы собрать данные на одном из процессов и затем произвести их вывод на терминал или в файл.

1.3. Пересылка данных между двумя процессами

Программы, состоящие из процессов, которые не взаимодействуют между собой редко встречаются на практике, так как для организации параллельных вычислений необходим обмен данными между процессами. Мы начнем рассмотрение средств вазимодействия в MPI с так называемых *точечных обменов*. Это название происходит от английского «point-to-point communications», которое выражает тот факт, что у взаимодействия есть две точки: процесс-отправитель и процесс-получатель сообщения.

Точечные, так же как и обмены других типов, всегда производятся в пределах одного коммуникатора, который указывается в качестве параметра в вызовах функций обмена. Номера процес-

сов, принимающих участие в обменах, вычисляются по отношению к указанному коммуникатору. На листинге 3 приведен пример программы, описывающей поведение двух взаимодействующих процессов, один из которых посылает данные, а другой их принимает и выводит на печать.

```
1:  #include <stdio.h>
2:  #include <mpi.h>
3:  main(int argc, char* argv[])
4:  {
5:    int rank;
6:    MPI_Status st;
7:    char buf[64];
8:    MPI_Init(&argc, &argv);
9:    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10:   if(rank == 0) {
11:     sprintf(buf, "Hello from process 0");
12:     MPI_Send(buf, 64, MPI_CHAR, 1, 0,
          MPI_COMM_WORLD);
13:   } else {
14:     MPI_Recv(buf, 64, MPI_CHAR, 0, 0,
          MPI_COMM_WORLD, &st);
15:     printf("Process %d received %s \n",
    rank, buf);
16:   }
17: MPI_Finalize();
18: }
```

Листинг 3. Пример точечной пересылки

Выполнение данного примера приводит к следующему результату:[1]

```
> mpirun -np 2 ./ex3
Process 1 received Hello from process 0
>
```

[1] Здесь и далее в тексте книги символ > употребляется для обозначения приглашения среды программирования (командной строки). На вашей системе приглашение может выглядеть по-другому.

Рассмотрим подробнее текст примера. Вначале (строки 8, 9) с помощью функций, рассмотренных ранее, производится инициализация библиотеки и в переменной `rank` запоминатся номер процесса. Далее следует оператор `if`, первая ветвь (строки 11, 12) которого выполняется процессом с номером 0, а вторая — всеми остальными процессами (в нашем случае — процессом с номером 1):

Процесс с рангом 0	Все остальные процессы (в нашем случае — только процесс с номером 1)
`11: sprintf(buf, "Hello from process 0");` `12: MPI_Send(buf, 64, MPI_CHAR, 1, 0, MPI_COMM_WORLD);`	`14: MPI_Recv(buf, 64, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &st);` `15: printf("Process %d received %s \n", rank, buf);`

В результате действий, произведенных первым процессом в буфер `buf` помещается текст приветствия (строка 11), который затем (строка 12) пересылается процессу с номером 1. Процесс 1 получает (строка 14) приветственное сообщение, сформированное и отправленное процессом 0, и распечатывает его на терминал (строка 15).

Для передачи данных в примере, приведенном на листинге 3, применяются функции точечных обменов. Отправка сообщения осуществляется функцией `MPI_Send`. Первым входным параметром функции является адрес области памяти, в которой хранятся передаваемые данные. В примере на листинге 3 этой областью является массив `buf`. Следующие два параметра служат для описания передаваемых данных. Предполагается, что данные хранятся в памяти подряд: указывается число элементов и тип каждого элемента. В нашем примере пересылается весь массив `buf`. Так как этот массив состоит из 64 элементов, то в качестве параметров `count` и `datatype` указываются 64 и тип `MPI_CHAR` соответственно:

Отправка сообщения:

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

`buf` – указатель на начало буфера передаваемых данных;
`count` – число элементов передаваемых данных;
`datatype` – MPI-тип элементов передаваемых данных;
`dest` – номер процесса, которому передаются данные;
`tag` – тэг сообщения;
`comm` – коммуникатор, в пределах которого передаются данные.

Прием сообщения:

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

`buf` – указатель на начало буфера, в котором будут сохранены полученные данные;
`count` – максимальное число элементов данных в сообщении;
`datatype` – MPI-тип элементов данных в сообщении;
`source` – номер процесса, от которого осуществляется прием данных;
`tag` – ожидаемый тэг сообщения;
`comm` – коммуникатор, в пределах которого передаются данные;
`status` – указатель на структуру, в которую будет записан статус завершения операции;

Указываемый тип элементов, составляющих сообщения, называется *MPI-типом*. MPI-тип является объектом данных типа `MPI_Datatype` и должен соответствовать типу данных, хранимых в области `buf`. В табл. 1 приводятся предопределенные MPI-типы и соответствующие им типы языка Си.

Параметр `dest` функции `MPI_Send` определяет ранг процесса, которому предназначено сообщение. Параметр `tag` задает так называемый *тэг сообщения*, который представляет собой целое число, передаваемое вместе с сообщением и проверяемое при его приеме. В примере на листинге 3 ранг процесса-получателя задается равным 1, а тэг — равным 0. Последним параметром

функции MPI_Send является коммуникатор, в пределах которого передается сообщение.

Таблица 1. Предопределенные типы MPI

Тип MPI	Тип Си
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
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long double

Для приема сообщения служит функция MPI_Recv. Первый параметр buf задает начало буфера для сохранения данных, пришедших в сообщении. Второй параметр count задает максимальное число элементов данных, которое предполагается принять. Обычно это число выбирается таким образом, чтобы сохранение данных не привело к переполнению буфера. Третий параметр задает тип элементов данных. В качестве четвертого параметра source передается номер процесса, от которого ожидается сообщение. Сообщения, пришедшие от процесса с другим номером, не принимаются. Следующий параметр задает тэг ожидаемого сообщения. Так же как и в случае с номером, сообщения, имеющие тэг, отличный от указанного, не принимаются. Предпоследний параметр comm — это коммуникатор, в пределах которого выполняется пересылка сообщения.

Завершает список параметр status, который является указателем на структуру типа MPI_Status, которая заполняется после завершении операции приема и содержит информацию о том, каким образом завершилась операция. Поля этой структуры будут подробно рассмотрены в ближайших разделах.

1.4. Численное интегрирование: параллельная реализация на основе MPI

Рассмотренное в предыдущих разделах подмножество MPI оказывается практически достаточным для реализации параллельного численного интегрирования. Простейшим способом приближенного вычисления значения определенного интеграла является *метод прямоугольников* [10]. Отрезок интегрирования разбивается на некоторое количество интервалов (рис. 1). На интервале (x_i, x_{i+1}) площадь под кривой $f(x)$ аппроксимируется к площади прямоугольника с основанием (x_i, x_{i+1}) и высотой $f(x_i + 0{,}5h)$, где $h = x_{i+1} - x_i$ — длина шага разбиения.

Несомненным достоинством данного метода с точки зрения распараллеливания является возможность независимого вычисления площадей разных прямоугольников. Это наблюдение подсказывает естественную параллельную реализацию, в которой каждый процесс вычисляет сумму площадей некоторого подмножества прямоугольников. Если производительности процессоров, составляющих параллельную систему, совпадают, то целесообразно разделить работу между ними поровну.

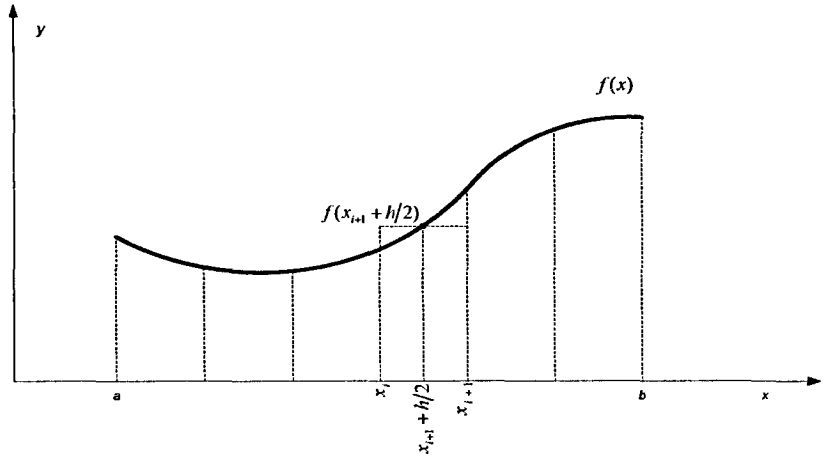


Рис. 1. Интегрирование методом прямоугольников

Одна из возможных реализаций такого распределения заключается в вычислении на процессоре с номером i площадей трапеций с номерами $i, i+p, i+2p, \ldots, i+mp$, где $m = \left[\frac{n-i}{p}\right]$, а n и p — общее число трапеций в разбиении и процессоров соответственно. Текст MPI-программы, реализующей данный алгоритм, приведен на листинге 4.

```
1:  #include <mpi.h>
2:  #include <stdio.h>
3:  double f(double x)
4:  {
5:    return 4./(1 + x * x);
6:  }
7:  main(int argc, char* argv[])
8:  {
9:    int r;
10:   int p;
11:   int i;
12:   double sum;
13:   double h;
14:   MPI_Status st;
15:   double t;
16:   int n = 1000000000;
17:   double a = 0.0;
18:   double b = 1.0;
19:   MPI_Init(&argc, &argv);
20:   MPI_Comm_rank(MPI_COMM_WORLD, &r);
21:   MPI_Comm_size(MPI_COMM_WORLD, &p);
22:   if(r == 0)
23:     t = MPI_Wtime();
24:   MPI_Barrier(MPI_COMM_WORLD);
25:   sum = 0.0;
26:   h = (b - a) / n;
27:   for(i = r; i < n; i += p)
28:     sum += f(a + (i + 0.5) * h);
29:   sum *= h;
30:   if(r != 0)
31:   MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
       MPI_COMM_WORLD);
```



```
32:   if(r == 0) {
33:     double s;
34:     for(i = 1; i < p; i ++) {
35:       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD, &st);
36:       sum += s;
37:     }
38:     t = MPI_Wtime() - t;
39:     printf("Integral value = %lf.  Time
          = %lf sec.\n", sum, t);
40:   }
41:   MPI_Finalize();
42: }
```

Листинг 4. Программа на MPI для численного интегрирования

Рассмотрим данную программу подробно. Вначале определяется подынтегральная функция `f`. Заметим, что одну и ту же программу можно использовать для вычисления интеграла от различных функций, определив функцию `f` по-другому. Функция `main` начинается с уже знакомой нам последовательности вызовов инициализации, определения числа запущенных процессов и номера процесса. Далее (строки 22, 23) производится замер времени с помощью функции `MPI_Wtime` на нулевом процессе. Эта функция возвращает время в секундах на момент вызова, представленное как число с плавающей точкой.

Вызов `MPI_Barrier` в строке 24 необходим для синхронизации всех процессов. Синхронизация нужна для точного определения времени работы всей программы. Функция `MPI_Barrier` блокирует вызывающий процесс до тех пор, пока все процессы, входящие в коммуникатор, переданный в качестве аргумента, также не выполнят этот вызов.

Операторы в строках 25–29 вычисляют сумму площадей прямоугольников. Этот код выполняется всеми процессами. Процесс с номером `r` вычисляет сумму только части прямоугольников с номерами `r, r+p`, и т. д. Для получения итогового результата значения, вычисленные процессами, необходимо просуммировать. Для этого все процессы кроме нулевого пересылают вычисленное значе-

ние нулевому (строка 31). Прием и суммирование присланных значений осуществляется операторами в строках 34–37 нулевым процессом, который распечатывает результат и затраченное время (строка 39).

На рис. 2 приведена зависимость времени работы от количества процессоров, задействованных в вычислениях. Расчеты проводились на суперкомпьютере MVS-15000 BM, который имеет следующие технические характеристики:

Процессор	PowerPC 970 2.2 GHz
Количество процессоров	918
Коммуникационное оборудование	Myrinet 2000

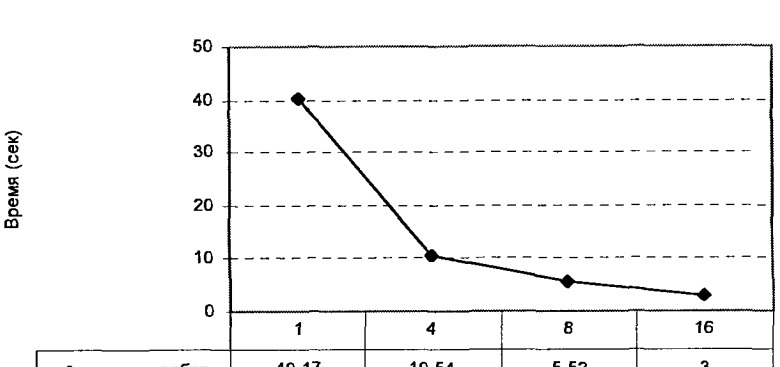


Рис. 2. Зависимость времени работы параллельной программы численного интегрирования от количества процессоров

Результаты расчетов демонстрируют существенное снижение времени работы при увеличении количества процессоров в решающем поле. Это подтверждает целесообразность применения параллельных вычислений для рассмотренной задачи.

1.5. Семантика точечных обменов

При рассмотрении базовых функций обмена сообщениями не уточнялось, каким именно способом передается сообщение процессу-приемнику от процесса-отправителя. Если процесс-приемник вызвал функцию приема ранее или в момент вызова функции посылки сообщения процессом-отправителем, то данные могут передаваться сразу, без задержки, так как процесс-получатель и процесс-отправитель оба готовы к передаче сообщений. В противном случае возможны следующие варианты развития ситуации:

1) ситуация рассматривается как ошибочная и передаваемое сообщение теряется;
2) процесс-отправитель блокируется до момента вызова парной операции приема;
3) сообщение буферизуется, процесс-отправитель продолжает свою работу.

Каждая из перечисленных реакций имеет свои достоинства и недостатки. Преимуществом первого варианта реализации обмена является минимизация накладных расходов на пересылку: не требуется задерживать процесс или организовывать буфер для его временного хранения. Вместе с тем, в таком варианте приложение с большой вероятностью попадает в ошибочное состояние, и при этом ошибка может исчезать и вновь появляться от запуска к запуску. Ошибки такого рода достаточно сложно выявлять и устранять. Несмотря на столь серьезные недостатки, MPI предусматривает такой вариант пересылки: для это служит специальная функция `MPI_Rsend`. Она имеет точно такие же параметры, как и функция `MPI_Send`. Применять эту функцию следует только в том случае, если гарантированно известно, что процесс-приемник ожидает сообщение к моменту вызова `MPI_Rsend`.

Существенным достоинством варианта 2 является то, что процессу-отправителю не требуется промежуточный буфер для хранения сообщения. Платой за такую экономию памяти может быть потеря времени за счет блокировки процесса-отправителя. MPI предусматривает специальную функцию `MPI_Ssend`, которая реализует именно этот вариант точечной пересылки. Ее параметры также не отличаются от параметров функции `MPI_Send`.

Так как скорость работы является наиболее важной характеристикой параллельной программы, то разработчики отдают предпочтение функциям, не вызывающим блокировку процесса. При этом (вариант 3) сообщение буферизуется, т.е. сохраняется до момента отправки в специально-выделенной области памяти. Если используется системный буфер, то программисту нет необходимости заботится о его выделении, но такой буфер может переполниться. Функция MPI_Send использует системный буфер следующим образом: сообщение сохраняется, если оно может быть сохранено, т.е. если размер сообщения не превосходит размера свободного места в системном буфере. В противном случае процесс-отправитель блокируется до тех пор, пока такое место освободится, или до того момента, когда будет выполнена парная операция. При этом выбор между буферизацией и блокировкой процесса производится системой MPI.

Если требуется в обязательном порядке обеспечить буферизацию сообщения, то следует пользоваться функцией MPI_Bsend, которая имеет те же параметры, что и MPI_Send. Эта функция *всегда* сохраняет сообщение в буфере, но в отличие от MPI_Send память под буфер предоставляется не системой, а выделяется самим пользователем.

1.6. Организация буферизованных пересылок

Функция MPI_Bsend всегда помещает данные в буфер. Выделение буфера является задачей приложения: перед тем как осуществлять пересылки, необходимо выделить область памяти необходимого размера. Для этого, прежде всего, нужно вычислить этот размер. Вычисление размера производится по формуле:

размер_буфера = число_сообщений ×
× размер_памяти_для_одного_сообщения.

Очевидно, что такой объем буфера необходим для хранения *всех* сообщений, пересылаемых процессом-отправителем. Реальной задействованный объем памяти может быть существенно меньше.

Размер памяти, требуемой для хранения одного сообщения, вычисляется с помощью функции MPI_Pack_size():


```
int MPI_Pack_size(int count, MPI_Datatype,
MPI_Comm comm, int* size)
```

count – число элементов данных;
datatype – тип одного элемента данных;
comm – коммуникатор, в рамках которого осуществляется обмен сообщениями;
size – итоговый размер.

Для получения размера сообщения, возвращаемого через последний параметр, функции передается число и тип передаваемых элементов данных, а также коммуникатор, по которому планируется пересылка. Дополнительно с каждым сообщением сохраняется некоторая служебная информация, для хранения которой требуется объем памяти, определяемый константой MPI_BSEND_OVERHEAD.

Таким образом, для получения размера памяти, которую нужно выделить для буферизации одного сообщения, сначала вычисляется размер сообщения с помощью функции MPI_Pack_size(), к которому затем добавляется MPI_BSEND_OVERHEAD.

После того как размер вычислен, память может быть выделена с помощью стандартных функций управления памятью: malloc, calloc и т.п. О выделенной памяти требуется сообщить библиотеке MPI при помощи вызова функции MPI_Buffer_attach:

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

buffer – адрес буфера;
size – размер буфера;

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

buffer_address – адрес буфера;
size – размер буфера.

В качестве первого аргумента этой функции передается адрес выделенного буфера, а в качестве второго — его размер в байтах.

Когда буфер больше не нужен, процесс должен выполнить вызов MPI_Buffer_detach перед тем, как освободить выделенную память. Первый и второй параметры этой функции являются выходными. По адресу, переданному в качестве первого аргумента, будет записан адрес буфера, а по адресу, переданному в качестве второго аргумента, — его размер.

Стандартная последовательность действий для организации буферизованной пересылки сообщений следующая:

1. Вычислить необходимый объем буфера (MPI_Pack_size).
2. Выделить память под буфер (malloc).
3. Зарегистрировать буфер в системе (MPI_Buffer_attach).
4. Выполнить пересылки.
5. Отменить регистрацию буфера (MPI_Buffer_dettach)
6. Освободить память, выделенную под буфер (free).

Важно отметить особенности работы с буфером:

- буфер всегда один и организуется на процессе-отправителе;
- для изменения размера буфера сначала следует вызвать MPI_Buffer_detach, затем выделить новый блок памяти нужного размера и зарегистрировать его с помощью вызова MPI_Buffer_attach;
- функция MPI_Buffer_detach блокирует процесс до того момента, когда все буферизованные сообщения будут отправлены, поэтому ее следует всегда вызывать перед освобождением памяти, выделенной под буфер – это предотвращает ситуацию, при которой освобождается память, в которой находятся не отправленные сообщения.

Фрагмент кода на листинге 5 иллюстрирует использование буферизованных обменов сообщениями. В строках 1, 2 вычисляется размер буфера, требуемый для пересылки M сообщений типа MPI_INT. В строках 4, 5 производится выделение памяти под буфер и регистрация буфера в системе. В цикле, расположенном в строках 5–8, посылается M сообщений, каждое из которых содержит в точности один элемент целого типа. В строках 9–11 снимается регистрация буфера и освобождается соответствующая память.


```
1:  MPI_Pack_size(1, MPI_INT,
      MPI_COMM_WORLD,&msize)
2:  blen = M * (msize + MPI_BSEND_OVERHEAD);
3:  buf = (int*) malloc(blen);
4:  MPI_Buffer_attach(buf, blen);
5:  for(i = 0; i < M; i ++) {
6:    n = i;
7:    MPI_Bsend(&n, 1, MPI_INT, 1, i,
        MPI_COMM_WORLD);
8:  }
9:  MPI_Buffer_detach(&abuf, &ablen);
10: free(abuf);
```

Листинг 5. Посылка данных с буферизацией.

1.7. Прием сообщения по шаблону

В MPI предусмотрена возможность указывать в качестве номера процесса-отправителя и тэга ожидаемого сообщения так называемые шаблоны — предопределенные константы MPI_ANY_SOURCE (для номера процесса) и MPI_ANY_TAG (для тэга). Если идентификатор MPI_ANY_SOURCE подставляется в качестве аргумента, соответствующего номеру процесса-отправителя при вызове функции приема, то сообщение будет принято в независимости от того, каков номер процесса, пославшего сообщение. Аналогично, если MPI_ANY_TAG подставляется в качестве аргумента, соответствующего тэгу ожидаемого сообщения, то сообщение принимается в независимости от его тэга.

Номер процесса, от которого пришло сообщение, может быть определен на основании последнего аргумента в вызове функции MPI_Recv (см. стр. 17). Этот аргумент указывает на структуру типа MPI_Status, поля которой получат значения в результате вызова функции приема сообщения. Структура MPI_Status имеет три обязательных поля с именами MPI_SOURCE, MPI_TAG, MPI_ERROR, которые содержат номер процесса, пославшего сообщения, тэг принятого сообщения и код ошибки, если передача сообщения прошла некорректно.

Поле MPI_ERROR избыточно при обычном вызове функции приема сообщения, когда код ошибки совпадает с кодом возврата функции. Оно предназначено главными образом для применения в функциях, завершающих выполнение нескольких операций приема, каждая из которых может вызвать ошибку. Такие функции рассматриваются в дальнейших разделах курса.

Поля MPI_SOURCE и MPI_TAG нужны для того, чтобы в случае использования шаблонов определить номер процесса-отправителя и тэг пришедшего сообщения. Если номер процесса-отправителя и тэг указаны точно в аргументах при вызове функции приема сообщения, то поля MPI_SOURCE и MPI_TAG дублируют эту информацию.

Применение шаблонов полезно при реализации параллельных программ, требующих *динамической балансировки нагрузки*. В таких программах ход вычислений нельзя предсказать заранее, он становится понятным только в процессе ее работы. Вследствие этого невозможно заранее спланировать распределение нагрузки по процессорам и пересылки оптимальным образом: работу приходится перераспределять по ходу вычислений. В частности, в программах подобного рода зачастую неизвестны последовательность и направление пересылок, поэтому применение шаблонов становится единственно возможным путем реализации таких программ. Пример такого приложения рассматривается в следующем разделе.

1.8. Стратегия управляющий—рабочие (master—slave): адаптивная квадратура

Одним из наиболее распространенных подходов к организации вычислений в параллельном программировании является стратегия управляющий—рабочие [12]. Эта стратегия предполагает, что все множество процессов параллельной программы разбивается на два подмножества (рис. 3). Первое подмножество состоит из одного процесса, называемого *управляющим*. Второе подмножество содержит все прочие процессы, которые называются *рабочими*.

Задачей управляющего процесса является распределение вычислений между рабочими процессами. Управляющий процесс посылает задачи свободным рабочим процессам для обработки. После решения своей задачи рабочий процесс посылает управ-

ляющему соответствующее уведомление, становится свободным и получает новое задание от управляющего процесса. В наиболее простом варианте список задач для посылки рабочим процессам генерируется самим управляющим процессом. Также возможно пополнение списка в процессе вычислений задачами, присланными с рабочих процессов. Такой подход характерен для задач древовидного поиска, в частности в методе ветвей и границ.

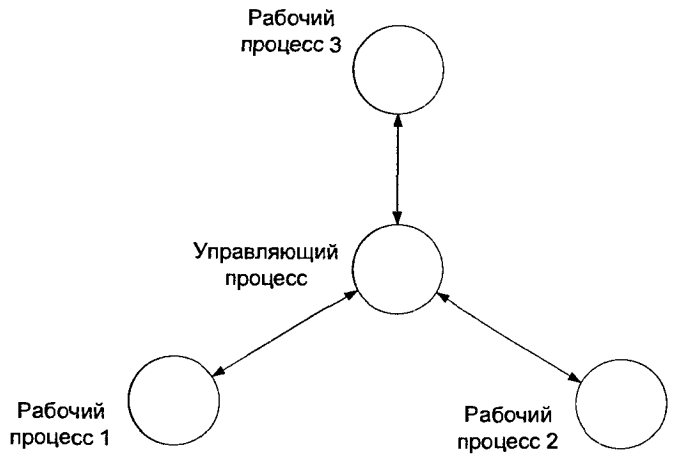


Рис. 3. Взаимодействие процессов в схеме управляющий—рабочие

Задачи, посылаемые рабочим процессам, могут быть различной трудоемкости, поэтому заранее неизвестно, какой из рабочих процессов быстрее справится со своей задачей. Для достижения максимальной эффективности распараллеливания новую задачу должен получать рабочий процесс, который первым освободился. Это можно обеспечить только с помощью приема по шаблону, который позволяет ожидать сообщение сразу от группы процессов.

Стратегия управляющий—рабочие находит свое применение при параллельной реализации различных проблем оптимизации, криптоанализа, биоинформатики и многих других. Одним из алгоритмов, который может быть эффективно реализован с помощью данной стратегии, является *адаптивная квадратура*, которая представляет собой алгоритм численного интегрирования с автоматическим выбором шага [10].

Адаптивная квадратура основана на том наблюдении, что гранулярность разбиения, необходимая для достижения заданной точности интегрирования, зависит от характера изменения функции. Так, например, оценка погрешности метода трапеций на отрезке $[a,b]$ зависит от величины шага интегрирования h и максимального значения абсолютной величины второй производной на отрезке интегрирования: $|\Psi| \leqslant \frac{h^2(b-a)}{12} \max_{x\in[a,b]} |f''(x)|$. Чем больше абсолютная величина второй производной, т.е. чем интенсивнее меняется первая производная, тем больше погрешность. Из этого следует, что для достижения заданной точности шаг целесообразно выбирать с учетом абсолютной величины второй производной.

Возможный подход к автоматизации выбора шага в случае, когда вычисление второй производной невозможно или слишком трудоемко, состоит в следующем: на каждом следующем шаге отрезок интегрирования делится пополам – добавляется еще одна точка, совпадающая с серединой отрезка. Если новое приближенное значение интеграла отличается от предыдущего не более чем на заданное число δ, то считается, что требуемая точность достигнута, и процесс вычислений можно останавливать. В противном случае процедура повторяется для каждого из двух отрезков, построенных в результате разбиения (рис. 4).

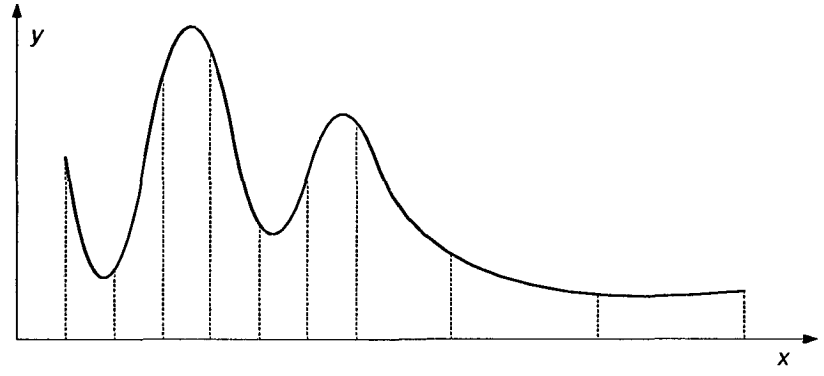


Рис. 4. Адаптивная квадратура:
шаг выбирается в зависимости от характеристик изменения функции

Основанием для этого метода служит *правило Рунге*: $I_i - I_{h,i} \approx \frac{I_{h/2,i} - I_{h,i}}{7}$, где I_i — значение интеграла на отрезке с номером i, $I_{h,i} = \frac{f(x_{i+1}) + f(x_i)}{2} h_i$ — приближенное значение интеграла, вычисленное по двум точкам, $I_{h/2,i} = \frac{f(x_{i+1}) + 2f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_i)}{2} h_i$ — приближенное значение интеграла, вычисленное по трем точкам. Таким образом, для достижения заданной точности ε параметр δ следует выбирать исходя из соотношения $\delta = 7\varepsilon$. Если для первоначального отрезка $[a,b]$ интегрирования задана точность ε, то для некоторого отрезка длиной h_i, полученного в результате таких разбиений, требуемая точность должна составлять $\varepsilon_i = \frac{\varepsilon h_i}{b-a}$ (вклад погрешности пропорционален длине отрезка).

Так как плавность поведения функции может отличаться на разных отрезках интегрирования, то соответственно может различаться и время работы алгоритма адаптивной квадратуры. Если просто распределить отрезок интегрирования поровну между рабочими процессами, то возможен существенный дисбаланс в загрузке, который приводит к потере производительности. Поэтому для балансировки загрузки узлов вычислительного комплекса целесообразно применить стратегию управляющий—рабочие, которая устроена следующим образом. Отрезок интегрирования разбивается на достаточно большое число интервалов одинаковой длины. Управляющий посылает рабочему процессу номер отрезка интегрирования, на котором рабочий процесс вычисляет приближенное значение интеграла, используя адаптивную квадратуру, и отсылает вычисленное значение управляющему процессу, после чего получает новое задание. На листинге 6 приведен текст MPI-программы, реализующей стратегию управляющий—рабочие для метода адаптивной квадратуры. Рассмотрим подробно действия, производимые в этой программе.

В начале программы подключается заголовочные файлы математической библиотеки и MPI, а также определяется подынтегральная функция `f`:

```
1:  #include <math.h>
2:  #include <mpi.h>
3:  #define MYABS(A) (((A) < 0)?(-(A)):(A))
4:  double f(double x)
5:  {
6:    return sin(1. / x);
7:  }
```

Следующая функция adint вычисляет приближенное значение Iold интеграла на отрезке (left, right), затем приближенное значение интеграла, вычисленное по трем точкам (левой границе, середине отрезка и правой границе), которое сохраняется по адресу nint, переданному функции в качестве аргумента. Если приближение по трем точкам отличается от приближения по двум точкам менее чем на заданную величину eps, то функция возвращает 1, в противном случае возвращается 0:

```
8:  int adint(double (*f) (double), double
      left, double right, double eps, double
      *nint)
9:  {
10:   double mid;
11:   double h;
12:   double Iold;
13:   double Ileft;
14:   double Iright;
15:   h = 0.5 * (right - left);
16:   mid = 0.5 * (right + left);
17:   Iold =  h * (f(left) + f(right));
18:   Ileft = 0.5 * h * (f(left) + f(mid));
19:   Iright = 0.5 * h * (f(mid) + f(right));
20:   *nint = Ileft + Iright;
21:   if(MYABS(Iold - *nint) < eps)
22:     return 1;
23:   else
24:     return 0;
25: }
```


Рекурсивная функция recadint вычисляет приближенное значение интеграла методом адаптивной квадратуры, используя при этом функцию adint. Если функция adint возвращает 1, то требуемая точность достигнута и функция возвращает вычисленное приближение (срока 29–31). В противном случае отрезок делится пополам и на каждой из половин вновь вызывается функция recadint с уменьшенным вдвое допустимым значением погрешности (строки 36–37). Функция возвращает сумму вычисленных приближений (строка 38):

```
26: double recadint(double (*f)(double),
      double left, double right, double eps)
27: {
28:   double I;
29:  if(adint(f, left, right, eps, &I)) {
30:     return I;
31:   } else {
32:     double Ileft;
33:     double Iright;
34:     double mid;
35:     mid = 0.5 * (right + left);
36:     Ileft = recadint(f, left, mid, 0.5 *
          eps);
37:     Iright = recadint(f, mid, right, 0.5 *
          eps);
38:     return Ileft + Iright;
39:   }
40: }
```

Собственно параллельная часть программы заключена в функции main. Процесс с рангом 0 выполняет роль управляющего, все остальные процессы являются рабочим. После инициализации следует определение ранга вызывающего процесса и общего числа процессов, участвующих в вычислении (строки 47–48). В строках 49–50 вычисляется длина отрезка разбиения и номер (начиная с нуля) последнего из этих отрезков. В строке 51 производится замер времени до начала выполнения основных вычислений с помощью функции MPI_Wtime. Затем управляющий процесс в цикле выполняет следующие действия (строки 55–57): принимает от любого из рабочих процессов вычисленное значе-

ние интеграла, прибавляет его к общей сумме и посылает освободившемуся процессу новое задание – номер очередного отрезка для вычисления приближенного значения интеграла. Цикл заканчивает свою работу, когда все отрезки отправлены с управляющего процесса рабочим для обработки (n = 0) и все рабочие процессы завершили вычисления (s = p - 1).

Рабочий процесс выполняет вычисления приближенного значения интеграла на присланном ему отрезке интегрирования: в строке 66 управляющему процессу посылается вычисленное на предыдущей итерации значение приближения, затем принимается номер очередного отрезка интегрирования (строка 67), для которого вычисляется приближенное значение интеграла (строка 69). Отрицательное значение номера отрезка интегрирования воспринимается как сигнал к остановке цикла (разрыв цикла в строке 71). В строках 75–76 производится замер времени счета, которое распечатывается вместе с вычисленным значением.

```
41: main(int argc, char* argv[])
42: {
43:   int r, p, n = 100000, s = 0;
44:   double a = 0.0001, b = 1.0, eps =
        0.0001, I = 0., h, t;
45:   MPI_Status st;
46:   MPI_Init(&argc, &argv);
47:   MPI_Comm_rank(MPI_COMM_WORLD, &r);
48:   MPI_Comm_size(MPI_COMM_WORLD, &p);
49:   h = (b - a) / n;
50:   n --;
51:   t = MPI_Wtime();
52:   if(r == 0) {
53:     while(s != (p-1)) {
54:       double Islave;
55:       MPI_Recv(&Islave, 1, MPI_DOUBLE,
            MPI_ANY_SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &st);
56:       I += Islave;
57:       MPI_Send(&n, 1, MPI_INT,
            st.MPI_SOURCE,0,
            MPI_COMM_WORLD);
58:       if(n >= 0)
```



```
59:         n --;
60:       else
61:         s += 1;
62:     }
63:   } else {
64:     int m;
65:     while(1) {
66:       MPI_Send(&I, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
67:       MPI_Recv(&m, 1, MPI_INT, 0,
            MPI_ANY_TAG,
            MPI_COMM_WORLD, &st);
68:       if(m >= 0)
69:         I = recadint(f,a+h*m,a+h*(m+1),
              eps*h/(b-a));
70:       else
71:         break;
72:     }
73:   }
74:   if(r == 0) {
75:     t = MPI_Wtime() - t;
76:     printf("Integral value: %lf,
            time = %lf\n", I, t);
77:   }
78:   MPI_Finalize();
79: }
```

Листинг 6. MPI-программа адаптивной квадратуры

Данные, полученные на суперкомпьютере MVS 15000BM, доказывают, что предложенный алгоритм позволяет получать существенное ускорение за счет распараллеливания (рис. 5).

Следует заметить, что схема управляющий—рабочие позволяет проводить балансировку нагрузки систем, состоящих из узлов различной производительности. Действительно, если разделить поровну работу между узлами с разной производительностью, то быстрые узлы закончат ее скорее и будут простаивать в ожидании завершения работы медленных. При применении схемы управляющий—рабочие быстрые узлы сообщат о завершении расчетов управляющему процессу и получат новое задание. Таким образом простоев удается избежать. Для эффективного применения метода

необходимо, чтобы общее задание было разделено на количество частей, превосходящее число узлов в системе в несколько раз.

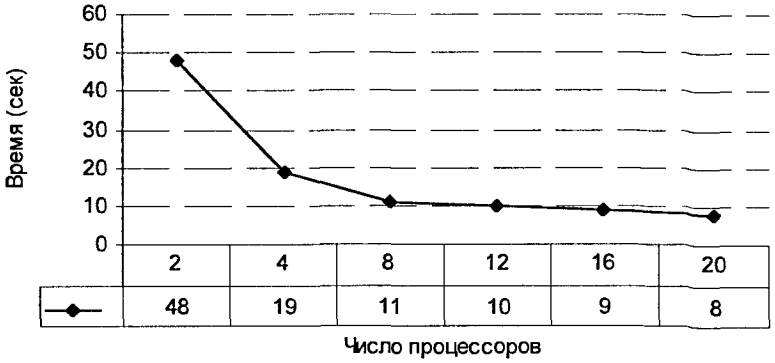


Рис. 5. Зависимость времени работы параллельной программы адаптивного численного интегрирования от количества процессоров

1.9. Отложенные пересылки данных

Синхронные операции отправки и приема данных всегда блокируют выполнение процесса, вызывающего эту операцию, до момента выполнения парной операции. В ряде приложений подобные задержки могут служить существенным препятствием для достижения максимальной производительности. Наибольшие задержки возможны при использовании функции `MPI_Ssend`, предполагающей синхронный режим передачи данных. Другие виды пересылок позволяют избежать задержки на стороне отправителя сообщения, но не позволяют сделать это на процессе-приемнике, так как функция приема сообщения `MPI_Recv` всегда предполагает блокировку. В целях преодоления перечисленных проблем в MPI предусмотрен механизм *отложенных операций.*

Идея отложенной операции заключается в том, что передача или прием сообщения разбивается на *инициализацию* и *завершение*. В результате инициализации процесс получает дескриптор отложенной операции, который затем используется в качестве

аргумента операции завершения. Рассмотрим функций инициализации операций приема и отправки сообщений:

Отправка сообщения:

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

`buf` – указатель на начало буфера передаваемых данных;
`count` – число элементов передаваемых данных;
`datatype` – MPI-тип элементов передаваемых данных;
`dest` – номер процесса, которому передаются данные;
`tag` – тэг сообщения;
`comm` – коммуникатор, в пределах которого передаются данные;
`request` – указатель на дескриптор операции.

Прием сообщения:

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

`buf` – указатель на начало буфера, в котором будут сохранены полученные данные;
`count` – максимальное число элементов данных в сообщении;
`datatype` – MPI-тип элементов данных в сообщении;
`source` – номер процесса, от которого осуществляется прием данных;
`tag` – ожидаемый тэг сообщения;
`comm` – коммуникатор, в пределах которого передаются данные;
`status` – указатель на структуру, в которую будет записан статус завершения операции;
`request` – указатель на дескриптор операции.

От стандартных функций обмена отложенные операции отличаются наличием дополнительного параметра `request`, который имеет тип указателя на тип `MPI_Request`. По адресу, передаваемому в качестве этого параметра, сохраняется дескриптор инициализированной операции, который может быть использован в программе в дальнейшем для контроля над этой операцией. В этом случае говорят, что полученный дескриптор *связан* с отложенной операцией.

Наиболее простая функция для завершения отложенной операции носит название `MPI_Wait`. Первым ее параметром является дескриптор отложенной операции, а вторым – указатель на область памяти, в которой будет сохранен статус завершения операции. Функция `MPI_Wait` блокирует выполнение процесса до момента завершения операции.

При успешном завершении операции обмена функция `MPI_Wait` освобождает память, занятую запросом и записывает по адресу, переданному через параметр `request`, константу `MPI_REQUEST_NULL`, обозначающую нулевой запрос. Если в качестве параметра передается нулевой дескриптор или дескриптор, не связанный с какой-либо операцией, то вызов функции `MPI_Wait` сразу же завершается с пустым статусом.

Ожидание завершения одной операции:

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

`request` – указатель на дескриптор отложенной операции;
`status` – указатель на статус завершения.

Проверка завершения одной операции:

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

`flag` – указатель индикатор завершения операции;
`request` – указатель на дескриптор отложенной операции;
`status` – указатель на статус завершения.

В MPI также предусмотрена возможность проверить завершенность операции без блокирования процесса. Для этой цели

служит функция `MPI_Test`. В отличие от `MPI_Wait`, эта функция не блокирует процесс. Если операция, соответствующая дескриптору `request`, не завершилась, то по указателю `flag` записывается 0. В этом случае значение статуса завершения операции не определено. Если операция успешно завершилась, то по указателю `flag` будет записано значение, отличное от нуля, память, занятая запросом `request`, освобождена, а по указателю `status` будет записан статус завершения операции.

В некоторых случаях нужно ожидать завершение двух или более отложенных операций. Для этих целей служат следующие функции:

Ожидание завершения одной операции:

```
int MPI_Waitany(int count, MPI_Request * ar-
ray_of_requests, int * index, MPI_Status *
status)
```

`count` – число дескрипторов отложенных операций;
`array_of_requests` – массив дескрипторов отложенных операций;
`index` – указатель на индекс завершенной операции;
`status` – указатель на статус завершения.

Проверка завершения одной операции:

```
int MPI_Testany(int count, MPI_Request * ar-
ray_of_requests, int * index, int* flag,
MPI_Status * status)
```

`flag` – указатель на индикатор завершения операции;
`array_of_requests` – массив дескрипторов отложенных операций;
`index` – указатель на индекс завершенной операции;
`status` – указатель на статус завершения.

Ожидание завершения множества операций:

```
int MPI_Waitall(int count, MPI_Request * ar-
ray_of_requests, MPI_Status * array_of_statuses)
```

`count` – количество ожидаемых операций;
`array_of_requests` – массив дескрипторов отложенных операций; `array_of_statuses` – массив статусов завершения операции.

1.10. Коммуникаторы и группы

Любое достаточно большое приложение редко состоит из одной функции. Как правило, помимо основной функции main, программа также содержит еще несколько. Функции, часто используемые различными приложениями, нередко объединяют в библиотеки.

В рассмотренных выше примерах также присутствовали функции. Но эти функции выполняли только последовательный код и не содержали действий, связанных с параллельными вычислениями. Очевидно, что существует потребность и в функциях, выполняющих параллельный код. Вызов такой функции выполняется процессами, входящими в некоторое подмножество, которое может в частном случае совпадать со всем вычислительным пространством, но может и отличаться от него. Проблема состоит в определении способа, которым следует передать функции информацию о том подмножестве вычислительного пространства, на котором производится ее вызов.

Поясним сказанное на следующем примере. Выделим из ранее рассмотренного примера численного интегрирования (листинг 4) функцию вычисления интеграла quad (листинг 7). Очевидно, что реализованная таким образом функция quad не будет корректно работать на множестве процессов, отличном от всего вычислительного пространства, так как все взаимодействия производятся в рамках коммуникатора MPI_COMM_WORLD, который соответствует всему вычислительному пространству. Следовательно, ее можно выполнять только на всем вычислительном пространстве. Снять это ограничение можно, если найти способ передачи функции quad информацию о множестве процессов, которые выполняют ее вызов. Сделать это можно, используя механизм *коммуникаторов*, который предусмотрен в MPI именно для идентификации подмножеств вычислительного пространства.

```
1:  double quad(double a, double b, int n)
2:  {
3:    int r, p, i;
```



```
4:    double h, sum;
5:    MPI_Status st;
6:    MPI_Comm_rank(MPI_COMM_WORLD, &r);
7:    MPI_Comm_size(MPI_COMM_WORLD, &p);
8:    sum = 0.0;
9:    h = (b - a) / n;
10:   for(i = r; i < n; i += p)
11:     sum += f(a + (i + 0.5) * h);
12:   sum *= h;
13:   if(r != 0)
14:     MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
          MPI_COMM_WORLD);
15:   if(r == 0) {
16:     double s;
17:     for(i = 1; i < p; i ++) {
18:       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
            MPI_COMM_WORLD, &st);
19:       sum += s;
20:     }
21:     return sum;
22:   }
23: }
```

Листинг 7. Функция вычисления интеграла

Коммуникатор — это объект, идентифицирующий некоторое подмножество вычислительного пространства. Предопределенный коммуникатор MPI_COMM_WORLD существует с начала выполнения параллельного приложения и соответствует множеству всех процессов вычислительного пространства. Коммуникаторы, соответствующие другим подмножествам вычислительного пространства, необходимо создавать с помощью специальных функций, которые рассматриваются далее. Так же как и коммуникатор MPI_COMM_WORLD, эти коммуникаторы можно использовать для организации обменов данными, передавая их в качестве параметров в функции посылки и приема сообщений. Внутри коммуникатора процессы нумеруются последовательными целыми числами, начиная с нуля.

Для обеспечения независимости от подмножества вычислительного пространства, на котором производится вызов, функции quad необходимо передавать коммуникатор, соответствующий

тому множеству процессов, на котором она вызывается. Пример такой реализации представлен на листинге 8: в качестве первого параметра comm в функцию quad передается коммуникатор. В строке 45 показано, каким образом можно вызывать эту функцию.

```
1:  #include <math.h>
2:  #include <mpi.h>
3:  #include <stdio.h>
4:  double f(double x)
5:  {
6:    return 4/(1 + x * x);
7:  }
8:  double quad(MPI_Comm comm, double a, double
      b, int n)
9:  {
10:   int r, p, i;
11:   double h, sum;
12:   MPI_Status st;
13:   MPI_Comm_rank(comm, &r);
14:   MPI_Comm_size(comm, &p);
15:   sum = 0.0;
16:   h = (b - a) / n;
17:   for(i = r; i < n; i += p)
18:     sum += f(a + (i + 0.5) * h);
19:   sum *= h;
20:   if(r != 0)
21:     MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
        comm);
22:   if(r == 0) {
23:     double s;
24:     for(i = 1; i < p; i ++) {
25:       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
          comm, &st);
26:       sum += s;
27:     }
28:     return sum;
29:   }
30: }

31: main(int argc, char* argv[])
32: {
33:   int r;
```



```
34:   int p;
35:   double t, sum;
36:   int n = 10000000;
37:   double a = 0.0;
38:   double b = 1.0;
39:   MPI_Init(&argc, &argv);
40:   MPI_Comm_rank(MPI_COMM_WORLD, &r);
41:   MPI_Comm_size(MPI_COMM_WORLD, &p);
42:   if(r == 0)
43:     t = MPI_Wtime();
44:   MPI_Barrier(MPI_COMM_WORLD);
45:   sum = quad(MPI_COMM_WORLD, a, b, n);
46:   if(r == 0) {
47:     t = MPI_Wtime() - t;
48:     printf("Integral value = %lf. Time = %lf
        sec.\n", sum, t);
49:   }
50:   MPI_Finalize();
51: }
```

Листинг 8. Программа на MPI для численного интегрирования: вычисление интеграла выделено в отдельную функцию

В программе, приведенной на листинге 8, можно было бы использовать и первый вариант функции quad, так как вызов функции производится на всем вычислительном пространстве. В то же время, вызвать функцию quad на подмножестве, отличном от всего вычислительного пространства, возможно только в рамках реализации, представленной на листинге 8. Для того чтобы привести пример такого использования, нам потребуются функции для создания и удаления коммуникаторов.

Один из наиболее общих способов создания коммуникатора основан на разбиении существующего коммуникатора на несколько. Для этого предназначена функция MPI_Comm_split:

Разбиение коммуникатора на несколько:

```
int MPI_Comm_split(MPI_Comm comm, int color, int
key, MPI_Comm *newcomm)
```

comm – исходный коммуникатор;

color – индикатор разбиения;

key – порядок присвоения номеров в создаваемых коммуникаторах;

newcomm – указатель на область памяти, для сохранения создаваемых коммуникаторов.

Создание коммуникатора по группе процессов:

```
int MPI_Comm_create(MPI_Comm comm,MPI_Group
group, MPI_Comm *newcomm)
```

comm – исходный коммуникатор;

group – группа, по которой создается коммуникатор;

newcomm – область для сохранения результата операции.

Освобождение коммуникатора:

```
int MPI_Comm_free(MPI_Comm *comm)
```

Первый параметр comm служит для передачи идентификатора разбиваемого коммуникатора. Функция должна вызываться на всех процессах, входящих в этот коммуникатор. Параметр color служит для выделения создаваемых коммуникаторов: процессы с одинаковым значением color образуют один из создаваемых коммуникаторов. Параметр key задает частичный порядок на множестве процессов каждого из создаваемых коммуникаторов. Ранги процессам в создаваемом коммуникаторе присваиваются в порядке возрастания значения key. Последний параметр newcomm указывает на область памяти, в которой сохраняются идентификаторы созданных коммуникаторов. В результате выполнения функции MPI_Comm_split коммуникатор comm разбивается на несколько коммуникаторов, количество которых совпадает с числом различных значений параметра color.

В качестве иллюстрации использования параметров key и color рассмотрим следующий фрагмент кода:

```
MPI_Comm_rank(MPI_COMM_WORLD, &r);
MPI_Comm_split(MPI_COMM_WORLD, r%2, r%2 ? 0
: - r, & newcomm);
```

В результате выполнения этой последовательности операторов будет создано два новых коммуникатора, содержащих процессы с четными и нечетными номерами из коммуникатора MPI_COMM_WORLD соответственно. При этом номера процессов, вошедших в первый коммуникатор, будут упорядочены так же, как номера в коммуникаторе MPI_COMM_WORLD, так как значения параметра key у них всех одинаковы. Процессы, вошедшие во второй коммуникатор, будут иметь обратный порядок, в соответствии со значением ключей:

MPI_COMM_WORLD	0	1	2	3	4	5
Значения COLOR	0	1	0	1	0	1
Значения key	0	0	–2	0	–4	0
Коммуникатор 1	–	0	–	1	–	2
Коммуникатор 2	2	–	1	–	0	–

В некоторых случаях требуется создать несколько новых коммуникаторов из части процессов, входящих в исходный. Для этого следует указать на процессах, которые не войдут ни в один из создаваемых коммуникаторов, значение MPI_UNDEFINED в качестве параметра color. На этих процессах по адресу newcomm будет записано значение MPI_COMM_NULL.

Следует отметить, что коммуникаторы, созданные в программе, следует удалять с помощью функции MPI_Comm_free. Функция должна вызываться на всех процессах, входящих в удаляемый коммуникатор. Пренебрежение своевременным освобождением созданных коммуникаторов может привести к переполнению памяти и, как следствие, к аварийному завершению программы.

На листинге 9 приведен фрагмент кода, в котором функция quad вызывается на двух непересекающихся коммуникаторах, созданных при помощи функции MPI_Comm_split. В функцию quad добавлен дополнительный параметр f, через который передается указатель на интегрируемую функцию. Это позволяет использовать quad для вычисления интегралов различных функций в одной и той же программе. В строке 48 производится создание коммуникатора с помощью вызова функции MPI_Comm_split рассмотренным ранее способом. Полученные коммуникаторы сохраняются в переменной newcomm, значение которой передается в качестве второго параметра функции quad.

В строках 52–55 производится вызов функции quad на созданных коммуникаторах. При этом на процессах, входящих в первый коммуникатор в качестве первого параметра функции quad, передается функция f1, а на процессах второго коомуникатора – функция f2. Отрезки интегрирования также отличаются.

```
1:  #include <math.h>
2:  #include <mpi.h>
3:  #include <stdio.h>
4:  #include <math.h>
5:  double f1(double x)
6:  {
7:    return 4./(1. + x * x);
8:  }
9:  double f2(double x)
10: {
11:   return  sin(x);
12: }
13: double quad(double (*f) (double),
       MPI_Comm comm, double a, double b, int n)
14: {
15:   int r, p, i;
16:   double h, sum;
17:   MPI_Status st;
18:   MPI_Comm_rank(comm, &r);
19:   MPI_Comm_size(comm, &p);
20:   sum = 0.0;
21:   h = (b - a) / n;
22:   for(i = r; i < n; i += p)
23:     sum += f(a + (i + 0.5) * h);
24:   sum *= h;
25:   if(r != 0)
26:     MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
         comm);
27:   if(r == 0) {
28:     double s;
29:     for(i = 1; i < p; i ++) {
30:       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
            comm, &st);
31:       sum += s;
```



```
32:     }
33:   }
34: }

35: main(int argc, char* argv[])
36: {
37:   int r, p, newr;
38:   double t, sum;
39:   int n = 10000000;
40:   double a = 0.0;
41:   double b = 1.0;
42:   double c = 0.0;
43:   double d = 3.14;
44:   MPI_Comm newcomm;
45:   MPI_Init(&argc, &argv);
46:   MPI_Comm_rank(MPI_COMM_WORLD, &r);
47:   MPI_Comm_size(MPI_COMM_WORLD, &p);
48:   MPI_Comm_split(MPI_COMM_WORLD, r % 2,
        r % 2 ? 0 : - r, &newcomm);
49:   if(r == 0)
50:     t = MPI_Wtime();
51:   MPI_Barrier(MPI_COMM_WORLD);
52:   if(r % 2)
53:     sum = quad(f1, newcomm, a, b, n);
54:   else
55:     sum = quad(f2, newcomm, c, d, n);
56:   MPI_Comm_rank(newcomm, &newr);
57:   if(newr == 0) {
58:     t = MPI_Wtime() - t;
59:     printf("Integral value = %lf.
          Time = %lf sec.\n", sum, t);
60:     fflush(stdout);
61:   }
62:   MPI_Comm_free(&newcomm);
63:   MPI_Finalize();
64: }
```

Листинг 9. Программа на MPI для численного интегрирования: вычисления на двух непересекающихся коммуникаторах

При сборке примера на листинге 9 необходимо подключить математическую библиотеку. На большинстве UNIX-систем для этого необходимо указать опцию -lm:

```
mpicc -o ex4 ex4.c -lm,
```

где ex4 — имя выполняемого модуля примера, а ex4.c — имя исходного файла. После завершения приложения на экране печатаются значения интегралов и времена, затраченные на их вычисление.

Альтернативный механизм создания коммуникаторов основан на использовании понятия *группы процессов*. Группа является объектом MPI, соответствующим упорядоченному множеству идентификаторов процессов. В отличие от коммуникатора, группа связана не с процессами приложения, а только с их идентификаторами, и является локальным объектом по отношению к процессу.

Процессы в группе, так же как и в коммуникаторе, нумеруются последовательно, начиная с нуля. Таким образом, группа задает отображение идентификаторов процессов в их номера. Количество процессов в группе и номер процесса может быть определен с помощью функций MPI_Group_size и MPI_Group_rank.

Одному и тому же множеству процессов могут соответствовать две различные группы, задающие на этом множестве различную нумерацию. Поэтому две группы могут считаться одинаковыми только при условии, что в них входят одни и те же процессы в одном и том же порядке. Сравнение двух групп на совпадение осуществляется при помощи функции MPI_Group_compare. Для установления соответствия между номерами, присвоенными одним и тем же процессам в различных группах, предназначена функция MPI_Group_translate_ranks.

Определение размера группы:

```
int MPI_Group_size(MPI_Group group, int *size)
```

group – группа;

size – указатель на область памяти для записи информации о количестве процессов в группе.

Определение номера процесса, выполняющего вызов функции, в группе:

```
int MPI_Group_rank(MPI_Group group, int *rank)
```


group – группа;

rank – указатель на область памяти для сохранения номера процесса.

Установление соответствия между номерами процессов в различных группах:

```
int MPI_Group_translate_ranks (MPI_Group group1,
int n, int *ranks1, MPI_Group group2, int *ranks2)
```

group1 – первая группа;

n – число элементов массивов ranks1 и ranks2;

ranks1 – массив номеров процессов в первой группе;

group2 – вторая группа;

ranks2 – массив для сохранения номеров процессов во второй группе.

Эта функция заполняет массив ranks2 номерами процессов в группе group2, которые имеют номера, перечисленные в ranks1 в группе group1.

Сравнение двух групп процессов:

```
int MPI_Group_compare(MPI_Group group1,MPI_Group
group2, int *result)
```

group1 – первая группа;

group2 – вторая группа;

result – указатель на область памяти для сохранения результата.

Если группа group1 содержит те же процессы, что и группа group2 и порядок процессов в этих группах совпадает, группы считаются одинаковыми, и по адресу result записывается константа MPI_IDENT, в противном случае результатом будет MPI_UNEQUAL.

Для формирования новых групп из созданных ранее в MPI предусмотрен набор операций MPI_Group_union, MPI_Group_intersection, MPI_Group_difference, которые являются аналогами теоретико-множественных операций (объединение, пересечение и разность) с учетом порядка. Также выделена специальная группа MPI_GROUP_EMPTY, не содержащая процессов. Следует отличать идентификатор MPI_GROUP_EMPTY от идентификатора MPI_GROUP_NULL, который соответствует несуществующей группе и служит аналогом нулевого указателя. Функции MPI_Group_

incl и MPI_Group_excl позволяют создать новую группу из заданного набора номеров существующей группы. Так же как и коммуникаторы, память, отведенную под группы процессов, необходимо удалять с помощью функции MPI_Group_free.

Объединение двух групп:

```
int MPI_Group_union(MPI_Group gr1, MPI_Group g2,
MPI_Group* gr3)
```

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции.

Набор процессов, входящих в gr3 получается объединением процессов, входящих в gr1 и gr2, причем элементы группы gr2, не вошедшие в gr1, следуют за элементами gr1.

Пересечение двух групп:

```
Int MPI_Group_intersection(MPI_Group gr1, MPI_
Group g2, MPI_Group* gr3)
```

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции.

Группа gr3 составлена из процессов, входящих в gr1 и в gr2, расположенных в том же порядке, что и в gr1.

Разность двух групп:

```
int MPI_Group_difference(MPI_Group gr1,
MPI_Group gr2, MPI_Group* gr3)
```

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции.

Группа gr3 составлена из процессов, входящих в gr1, но не входящих в gr2, расположенных в том же порядке, что и в gr1.

Переупорядочивание (с возможным удалением) процессов в существующей группе:

```
int MPI_Group_incl(MPI_Group group, int n, int*
ranks, MPI_Group* newgroup)
```

group – исходная группа;

n – число элементов в массиве ranks;

ranks – массив номеров процессов, из которых будет создана новая группа;

newgroup – указатель на область для сохранения результата операции.

Созданная группа newgroup содержит элементы группы group, перечисленные в массиве ranks: i-й процесс создаваемой группы newgroup совпадает с процессом, имеющим номер ranks[i] в группе group.

Удаление процессов из группы:

```
int MPI_Group_excl(MPI_Group group, int n, int*
ranks, MPI_Group* newgroup)
```

group – исходная группа;

n – число элементов в массиве ranks;

ranks – массив номеров удаляемых процессов;

newgroup – указатель на область для сохранения результата операции.

В результате выполнения этой операции создается новая группа newgroup, получаемая удалением из исходной группы процессов с номерами, перечисленными в массиве ranks.

Получение коммуникатора по группе:

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group
*group)
```

comm – коммуникатор;

group – указатель на область памяти для сохранения полученной группы.

Освобождение памяти, отведенной для группы:

```
int MPI_Group_free(MPI_Group* group)
```

group – идентификатор освобождаемой группы.

Понятия «группы» и «коммуникатора» тесно связаны: каждому коммуникатору соответствует группа входящих в него процессов. Эта группа может быть получена с помощью функции MPI_Comm_group. По группе можно создать коммуникатор с помощью функции MPI_Comm_create (см. стр. 45). В качестве первого аргумента этой функции передается идентификатор исходного коммуникатора, а в качестве второго — группа, обра-

зующая подмножество процессов этого коммуникатора. В результате создается новый коммуникатор, объединяющий процессы из этой группы. Результат операции сохраняется в области памяти, указатель на которую передается в качестве третьего аргумента. Функция `MPI_Comm_create` должна вызываться на всех процессах, входящих в исходный коммуникатор.

В данном разделе были рассмотрены наиболее употребительные функции, которые, однако, не исчерпывают всех возможностей, предоставляемых MPI для управления коммуникаторами и группами. В частности, не были рассмотрены атрибуты коммуникаторов, топологии, интер-коммуникаторы. Подробное описание этих возможностей можно найти в [17].

1.11. Коллективные взаимодействия процессов

До настоящего момента мы рассматривали взаимодействия, в которых участвуют ровно два процесса — отправитель и получатель сообщения. Такие обмены относятся к классу *«точка-точка»* (point-to-point communications), их иногда также называют *точечными обменами.* Аппарат точечных обменов позволяет эффективно программировать широкий класс приложений и, в принципе, достаточен для выражения любого параллельного алгоритма. Однако на практике зачастую появляются задачи, в которых требуются взаимодействия, вовлекающие сразу несколько процессов. Такие операции называются *коллективными взаимодействиями.*

Введение коллективных операций преследует две основные цели. Первая состоит в облегчении процесса программирования за счет более выразительных средств. В частности, сокращается объем исходного теста, упрощается восприятие программы. Вторая цель состоит в повышении производительности приложения, которое достигается за счет оптимизации пересылок данных.

Поясним сказанное на примере. Пусть требуется выполнить рассылку значения переменной целого типа с процесса под номером 0 остальным процессам. Фрагмент кода, использующий точечные пересылки, выполняющий эти действия, и альтернативный вариант с использованием коллективной операции приведены на листинге 10.

Рассылка данных с помощью точечных пересылок:

```
1:  if(r == 0){
2:    for(i = 1; i < s; i ++)
3:      MPI_Send(&a, 1, MPI_INT, i, 0,
          MPI_COMM_WORLD);
4:  } else
5:    MPI_Recv(&a,1,MPI_INT,0,0,MPI_COMM_WORLD,
        &st);
```

Рассылка данных с использованием коллективной операции:

```
      MPI_Bcast(&a, 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

Листинг 10. Рассылка данных с помощью точечных обменов и коллективной операции

Реализация рассылки, приведенная в строках 1–5, не только занимает больше места в программе, но и, как правило, менее эффективна, чем соответствующая библиотечная функция. Причина этого заключается в том, что функция, предоставляемая библиотекой, может реализовать данную операцию более эффективно. На рис. 6 представлены две возможные реализации рассылки сообщения от процесса с номером 0 оставшимся 7 процессам. Реализация, представленная на рис. 6, *а*, соответствует фрагменту кода в строках 1–6 листинга 10. Если предположить, что на одну пересылку процесс 0 расходует единичный квант времени, то рассылка сообщения по семи процессам требует 7 таких квантов. Более экономная реализация рассылки (рис 6, *б*), занимает 3 временных кванта.

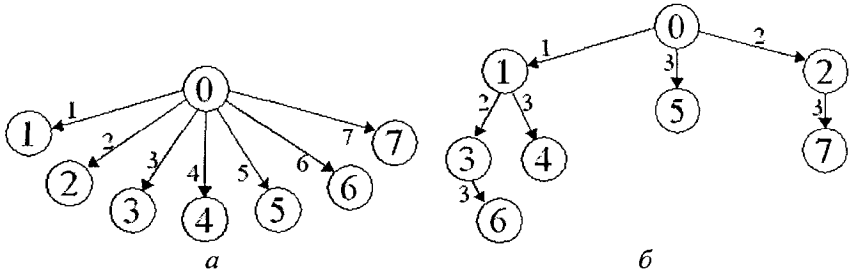


Рис. 6. Возможные варианты рассылки сообщения. Вершины соответствуют процессам. Дуги помечены номерами временных квантов

В предложенных реализациях учитывается только время, которое тратит процесс, посылающий сообщение, но не учитываются сетевые задержки. Количество пересылок, которые можно производить одновременно зависит от устройства коммуникационной среды многопроцессорного вычислительного комплекса. Хорошая реализация коллективной операции должна также учитывать характеристики коммуникационной среды.

1.11.1. Барьерная синхронизация

Простейшая коллективная операция `MPI_Barrier` принимает на вход лишь один аргумент — коммуникатор — и вызывает синхронизацию всех процессов, входящих в него. Более точно семантика функции формулируется следующим образом: вызывающий процесс блокируется до тех пор, пока все процессы, входящие в коммуникатор, переданный в качестве аргумента, не выполнят вызов `MPI_Barrier`.

```
int MPI_Barrier(MPI_Comm * comm)
```

`comm` – коммуникатор, объединяющий синхронизуемые процессы;

Функция барьерной синхронизации обычно применяется для точного измерения времени выполнения параллельной программы (см. пример на листинге 4) или для преодоления сложностей, вызванных недетерминированным поведением.

1.11.2. Основные коллективные операции синхронизации, рассылки и сбора данных

Так же как и рассмотренная функция барьерной синхронизации, любая коллективная операция должна вызываться всеми процессами, входящими в коммуникатор, передаваемый ей в качестве параметра. Ситуация, в которой это условие не выполнено, считается ошибочной. При этом выполнение коллективной операции, отличной от `MPI_Barrier`, не обязано, хотя и может, вызывать синхронизацию участвующих в ее выполнении процессов. Конкретный вариант поведения определяется реализацией. Поэтому переносимая программа не должна предполагать синхронизацию процессов, выполняющих коллективную операцию, но также должна корректно работать в ситуации, когда такая синхронизация имеет место.

Перейдем теперь к более подробному рассмотрению основных функций для коллективной рассылки и приема данных. Функция `MPI_Bcast` предназначена для рассылки одной копии данных группе процессов. Данные располагаются в памяти процесса с номером `root` в коммуникаторе `comm` по адресу `buffer` и состоят из `count` элементов типа `datatype`. В результате выполнения операции все процессы получат эти данные и сохранят их по адресу `buffer`:

```
int MPI_Bcast(void* buffer, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```

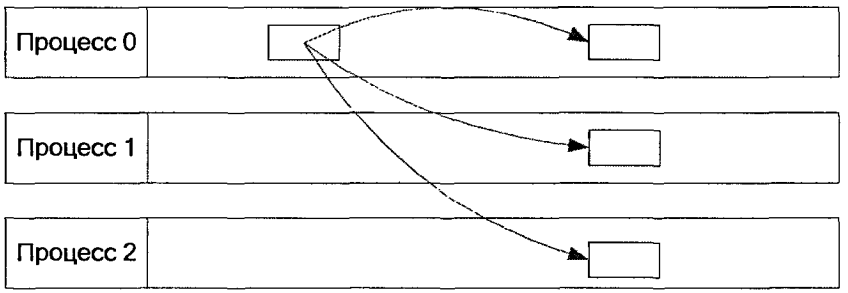
`buffer` – на процессе с рангом `root` – указатель на посылаемые данные, на других процессах – указатель на область памяти для сохранения принятых данных;

`count` – количество пересылаемых элементов данных;

`datatype` – тип пересылаемых элементов данных;

`root` – номер процесса, с которого рассылаются данные;

`comm` – коммуникатор, объединяющий процессы, участвующие в пересылке;



Зачастую требуется рассылать различные, а не одинаковые копии данных различным процессам. Для этого предназначена функция `MPI_Scatter`. Данная функция рассылает блоки дан-

ных одинакового размера, содержащие sendcount элементов типа sendtype, расположенные подряд начиная с адреса sendbuf на процессе с рангом root, всем остальным процессам, входящим в коммуникатор comm. Число блоков совпадает с числом процессов в коммуникаторе comm, передаваемым в качестве последнего параметра:

```
int MPI_Scatter(void* sendbuffer, int sendcount,
MPI_Datatype sendtype, void* recvbuffer, int
recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

sendbuffer – указатель на посылаемые данные (существенен только на процессе с рангом root);

sendcount – количество элементов в пересылаемом блоке данных;

sendtype – тип элементов пересылаемых данных;

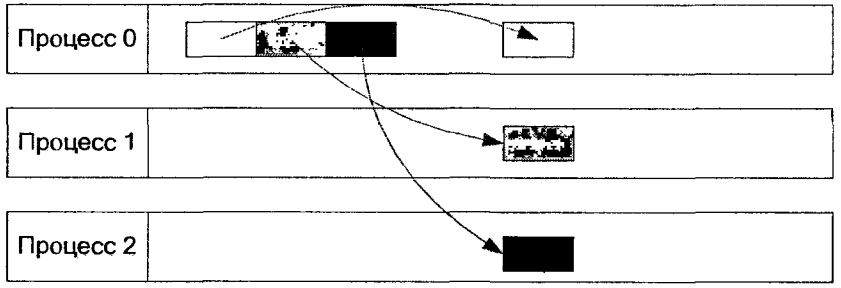
recvbuffer – указатель на область памяти для сохранения принятых данных;

recvcount – количество элементов в блоке принимаемых данных;

recvtype – тип принимаемых элементов данных;

root – номер процесса, с которого рассылаются данные;

comm – коммуникатор, объединяющий процессы, участвующие в коллективной операции.



Важно отметить, что для функции MPI_Scatter, как и для других коллективных операций, объем данных в посылаемом сообщении, определяемый параметрами sendcount и sendtype,

должен в точности совпадать с объемом данных, указанным для принимаемого сообщения, который задается параметром recvcount и recvtype. Эта особенность отличает коллективные операции от точечных пересылок, в которых объем данных, указанных в функции приема сообщения может превышать объем пришедшего сообщения.

Функция MPI_Gather выполняет операцию, обратную по отношению к функции MPI_Scatter, т.е. собирает одинаковые блоки данных, пересылаемые с разных процессов, в массив на одном из них:

```
int MPI_Gather(void* sendbuffer, int send-
count, MPI_Datatype sendtype, void* recvbuffer,
int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

sendbuffer – указатель на посылаемые данные;

sendcount – количество элементов в пересылаемом блоке данных;

sendtype – тип элементов пересылаемых данных;

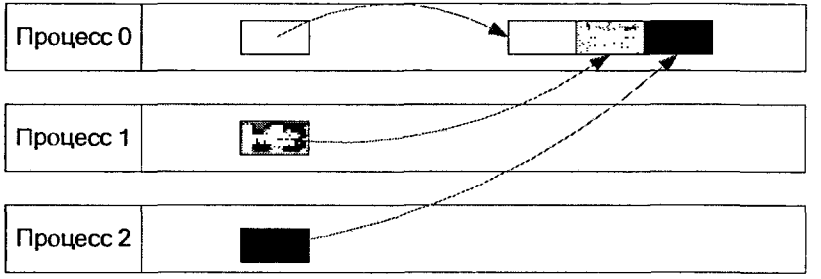
recvbuffer – указатель на область памяти для сохранения принятых данных (существенен только на процессе с рангом root);

recvcount – количество элементов в блоке принимаемых данных;

recvtype – тип принимаемых элементов данных;

root – номер процесса, на котором выполняется прием данных;

comm – коммуникатор, объединяющий процессы, участвующие в коллективной операции.



В качестве иллюстрации применения операций `MPI_Bcast`, `MPI_Gather` и `MPI_Scatter` рассмотрим параллельную реализацию метода Якоби решения системы линейных уравнений.

Приведем краткое описание алгоритма Якоби. (Более подробное описание можно найти в монографии [10].) Пусть дана система линейных уравнений вида

$$Ax = f, \tag{1}$$

где $A = (a_{ij})$ — квадратная матрица размерности m; $x = (x_1, \ldots, x_m)^T$, $f = (f_1, \ldots, f_m)^T$ — вектора переменных и правой части соответственно. Если все a_{ii} отличны от нуля, то система (1) может быть переписана в следующем виде: $x_i = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j - \sum_{j=i+1}^{m} \frac{a_{ij}}{a_{ii}} x_j + \frac{f_i}{a_{ii}}$, $i = 1, 2, \ldots, m$. Такая запись удобна для применения итерационных методов решения системы (1).

Метод Якоби состоит из следующих действий:

1. Произвольным образом выбирается начальное приближение $(x_1^0, \ldots, x_m^0)$ к решению.

2. На n-м шаге очередное приближение вычисляется по формуле $x_i^{n+1} = -\sum_{j=1}^{i-1} \frac{a_{ij}}{a_{ii}} x_j^n - \sum_{j=i+1}^{m} \frac{a_{ij}}{a_{ii}} x_j^n + \frac{f_i}{a_{ii}}$, $i = 1, 2, \ldots, m$.

3. Проверяется условие окончания процесса вычислений $\left\| x^{n+1} - x^n \right\| = \sqrt{\sum_{i=1}^{n} (x_i^{n+1} - x_i^n)^2} \leqslant \varepsilon$. Если оно выполнено или выполнено заданное количество итераций, то процесс вычислений заканчивается. В противном случае последовательность шагов 2–3 повторяется вновь.

Вычисление очередного приближения можно переписать в матричной форме следующим образом:

$$x^{n+1} = -Bx^n + g,$$

где элементы матрицы B и вектора g вычисляются в соответствии с формулой

$$b_{ij} = \begin{cases} a_{ij}/a_{ii} \text{ при } i \neq j, \\ 0 \text{ при } i = j; \end{cases} \qquad g_i = \frac{f_i}{a_{ij}}. \tag{2}$$

Можно показать, что если A — симметричная положительно-определенная матрица с диагональным преобладанием, т.е. $a_{ii} > \sum_{j \neq i} |a_{ij}|$, то метод Якоби сходится. Другими словами, для таких матриц условие окончания вычислений будет выполнено после конечного числа итераций при любом начальном приближении.

Рассмотрим матрицу A, состоящую из элементов следующего вида: $a_{ii} = 2m$, $a_{ij} = 1$ при $i \neq j$. Так как условие диагонального преобладания для нее выполнено, то метод Якоби сходится. Эту матрицу возьмем в качестве входных данных для рассматриваемых далее примеров. Компоненты правой части f подберем таким образом, чтобы вектор (1, 2, ..., m) был решением системы (1). Для этого необходимо положить $f_i = \frac{m(m+1)}{2} + i(2m-1)$. В этом случае коэффициенты матрицы B и вектора g согласно формуле (2) имеют следующий вид:

$$b_{ij} = \begin{cases} 1/a_{ii} \text{ при } i \neq j, \\ 0 \text{ при } i = j; \end{cases} \qquad g_i = \frac{m+1}{4} + i\left(1 - \frac{1}{2m}\right). \tag{3}$$

Формула (3) используется в функции инициализации исходных данных, приведенной на листинге 11. Помимо инициализации матрицы B и вектора g функция `init` также инициализирует вектор x, который является начальным приближением к решению. Так как при таких исходных данных метод Якоби сходится при любом начальном приближении, то выбор в качестве начального вектора (1, 1, ..., 1) корректен.

```
1:  void init(int m, double* B, double* g,
       double* x)
2:  {
3:    int i, j;
4:    for(i = 0; i < m; i ++) {
5:      g[i]=((double)m+1)/4.+(1.-1./
          (double)(2*m))*(i+1);
6:      x[i] = 1.;
7:      B[i * m + i] = 0.0;
8:    }
9:    for(i = 0; i < m; i ++) {
10:     for(j = 0; j < m; j ++)
11:       B[i * m + j] = 1./(double)(2 * m);
12:   }
13: }
```

Листинг 11. Инициализация исходных данных для метода Якоби

На листинге 12 приведен код другой вспомогательной функции `evalDiff`, которая применяется для вычисления выражения $\|v-u\| = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}$, используемого при оценке погрешности.

```
1:  double evalDiff(double* u, double* v, int m)
2:  {
3:    int i;
4:    double a = 0.0;
5:    for(i = 0; i < m; i ++) {
6:      double b;
7:      b = v[i] - u[i];
8:      a += b * b;
9:    }
10:   return sqrt(a);
11: }
```

Листинг 12. Вычисление нормы разности векторов

В алгоритме можно выделить еще одну вспомогательную функцию, выполняющую вычисление очередного приближения по формуле $x^{n+1} = -Bx^n + g$:

```
1:  void matvec(double* B, double* g, double*
       xold, double* x, int k, int m)
2:  {
3:    int i;
4:    for(i = 0; i < k; i ++) {
5:      int j;
6:      double a = 0.;
7:      double* row = B + i * m;
8:      for(j = 0; j < m; j ++)
9:        a += row[j] * xold[j];
10:     x[i] = -a + g[i];
11:   }
12: }
```

Листинг 13. Функция вычисления очередного приближения

В качестве аргументов в функцию передается матрица `B`, вектор `g`, адреса первого элемента предыдущего приближения `xold` (x_n) и первого элемента очередного приближения `x` (x_{n+1}). Также передается количество строк `k` и столбцов матрицы `m`. В рассматриваемом примере матрица квадратная, и можно было бы ограничиться одним параметром, но в параллельном варианте потребуется вариант этой функции для матрицы произвольного вида.

Функции `init`, `evalDiff` и `matvec` используются как последовательным, так и параллельным вариантами реализации метода Якоби. Последовательный вариант приведен на листинге 14. Рассмотрим его более подробно. В строках 5–6 производится разбор параметров командной строки: получают значения размерность задачи `m` и требуемая точность `eps`. Далее в строках 7–10 выделяется память для матрицы `B`, вектора `g` , которые используются для получения нового значения приближения по формуле $x^{n+1} = -Bx^n + g$, векторов `x` и `xold`, предназначенных для хранения очередного (x^{n+1}) и предыдущего (x^n) приближений. Далее в строке 11 аллокированные массивы инициализируются с помощью функции `init`.

```
1:  main(int argc, char* argv[])
2:  {3:    int m, I = 0;
4:    double *B, *g, *x, *xold, eps, diff, t;
5:    m = atoi(argv[1]);
6:    eps = atof(argv[2]);
7:    B = (double*)malloc(m * m *
sizeof(double));
8:    g = (double*)malloc(m * sizeof(double));
9:    xold = (double*)malloc(m *
sizeof(double));
10:   x = (double*)malloc(m * sizeof(double));
11:   init(m, B, g, xold);
12:   t = time(NULL);
13:   do {
14:     matvec(B, g, xold, x, m, m);
15:     diff = evalDiff(xold, x, m);
16:     I ++;
17:     printf("diff=%lf, eps = %lf\n", diff,
eps);
18:     memcpy(xold, x, m * sizeof(double));
19:   } while ((diff >= eps) && (I <= MAX-
ITERS));
20:   t = time(NULL) - t;
21:   printf("%d iterations consumed %lf
        seconds\n", I, t);
22: }
```

Листинг 14. Последовательный вариант реализации метода Якоби

Основной цикл (строки 13–19) вычисляет очередное приближение по формуле $x^{n+1} = -Bx^n + g$ до тех пор, пока не выполнено одно из условий останова: норма разности $\|x^{n+1} - x^n\|$ стала меньше заданной точности eps или выполнено более определенного числа итераций.

Несложно заметить, что вычисление различных компонентов очередного приближения может производиться параллельно. Это наблюдение оказывается полезным при параллельной реализации: каждый процессор вычисляет определенную часть элементов очередного приближения. Вычисления организуются по следующей схеме. На первом этапе исходные данные задачи — матрица B и вектор g — рассылаются по рабочим процессам: все получают одинаковые части матрицы и вектора, которые сохраняются в массивах Bloc и gloc. Иллюстрация для матрицы 6 × 6 приведена на рис. 7: каждый из трех процессов получает две строки матрицы B и два элемента вектора g.

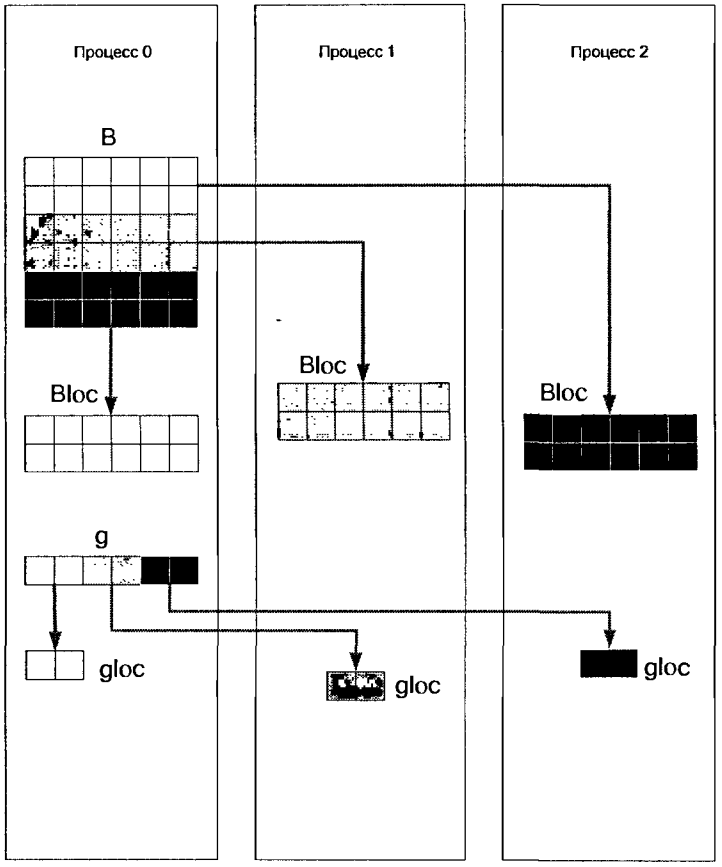


Рис. 7. Рассылка исходных данных задачи в параллельном варианте метода Якоби

Первая часть программы, содержащая объявление переменных, инициализацию и рассылку исходных данных приведена на листинге 15. В строке 11 вычисляется число строк chunk, обрабатываемое каждым процессором. В ситуации когда все процессоры имеют одинаковую производительность, целесообразно разделить строки между процессорами одинаковыми порциями, равными отношению числа строк матрицы m к количеству процессоров np. В строках 14–16 число строк матрицы m, точность eps и количество строк chunk для обработки каждым процессом рассылаются с процесса 0 всем остальным chunk. В строках 17–22 производится выделение памяти для хранения следующих массивов:

g – вектор g,

x – очередное приближение (x^{n+1}),

xold – предыдущее приближение (x^n),

Bloc – массив для хранения локальной части матрицы B,

gloc – массив для хранения локальной части вектора g,

xloc – массив для хранения локальной части вектора очередного приближения x^{n+1}.

В строках 27–28 с помощью MPI_Scatter производится рассылка массивов B и g.

```
1:  main(int argc, char* argv[])
2:  {
3:    int m,np,rk,chunk,i,I = 0;
4:    double *B,*Bloc,*g,*gloc,*x,*xloc,*xold,
        t,diff, eps;
5:    MPI_Init(&argc, &argv);
6:    MPI_Comm_size(MPI_COMM_WORLD, &np);
7:    MPI_Comm_rank(MPI_COMM_WORLD, &rk);
8:    if(rk == 0) {
9:      m = atoi(argv[1]);
10:     eps = atof(argv[2]);
11:     chunk = m / np;
12:     B = (double*)malloc(m*m*sizeof(double));
13:   }
14:   MPI_Bcast(&m, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
```



```
15:   MPI_Bcast(&eps,1,MPI_DOUBLE,
        0,MPI_COMM_WORLD);
16:   MPI_Bcast(&chunk, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
17:   g=(double*)malloc(m * sizeof(double));
18:   x=(double*)malloc(m * sizeof(double));
19:   xold=(double*)malloc(m * sizeof(double));
20:   Bloc=(double*)malloc(chunk*m*
        sizeof(double));
21:   gloc=(double*)malloc(chunk *
        sizeof(double));
22:   xloc=(double*)malloc(chunk *
        sizeof(double));
23:   if(rk == 0){
24:     init(m, B, g, xold);
25:     t = MPI_Wtime();
26:   }
27:   MPI_Scatter(B,m*chunk,MPI_DOUBLE,Bloc,
        m*chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
28:   MPI_Scatter(g,chunk,MPI_DOUBLE,gloc,
        chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

Листинг 15. Параллельный вариант реализации метода Якоби: инициализация и рассылка исходных данных

Следующий этап связан с выполнением итераций метода Якоби. Также как и в последовательном варианте, в цикле, представленном на листинге 16, вычисляется очередное приближение по формуле $x^{n+1} = -Bx^n + g$. Коммуникационная схема одной итерации цикла представлена на рис. 8. При этом каждый процесс вычисляет с помощью определенной ранее функции matvec только часть приближения, соответствующую переданной ему части строк матрицы B, формируя не весь вектор очередного приближения, а только его фрагмент, сохраняемый в массиве xloc. Затем фрагменты, сформированные на разных процессах, объединяются на нулевом процессе в вектор очередного приближения x с помощью вызова функции MPI_Gather в строке 32. Далее в строках 33–36 производится вычисление нормы разности предыдущего и очередного приближения $\|x^{n+1} - x^n\|$ и копирование

очередного приближения на место предыдущего, которое рассылается всем процессам в начале следующей итерации цикла.

Следует отметить, что в отличие от последовательного варианта в случае выполнения условия завершения, цикл следует разрывать на всех процессах. Так как норма разности приближений вычисляется только на процессе 0, то ее значение необходимо разослать всем процессам. Для этого в строке 40 производится вызов функции `MPI_Bcast`, которая рассылает значения переменной `diff`, в которой было сохранено значение нормы разности приближений на процессе 0.

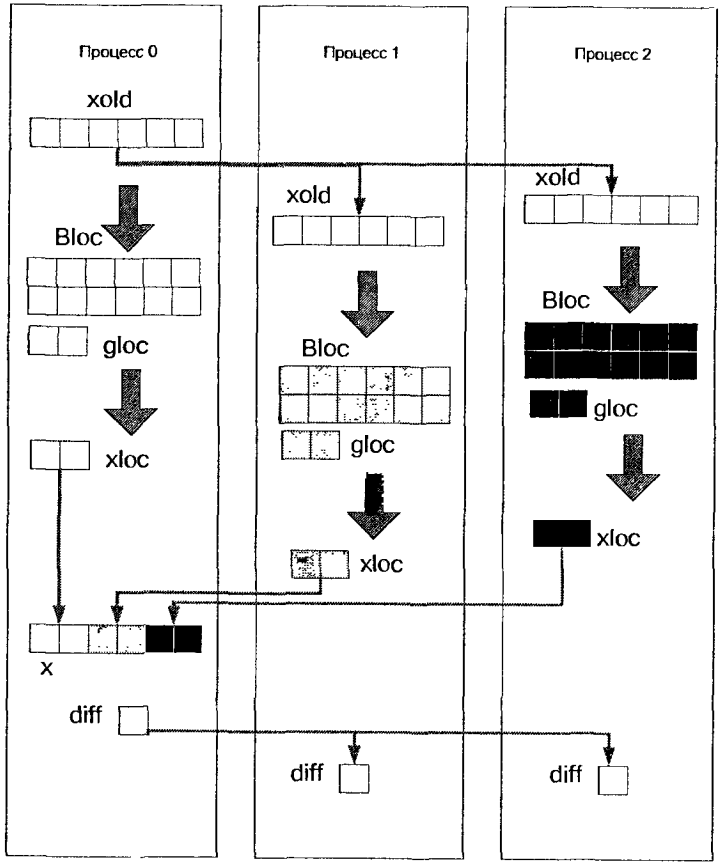


Рис. 8. Коммуникационная схема одной итерации цикла параллельного варианта метода Якоби


```
29:   do {
30:     MPI_Bcast(xold,m,MPI_DOUBLE,0,
          MPI_COMM_WORLD);
31:     matvec(Bloc, gloc, xold, xloc, chunk, m);
32:     MPI_Gather(xloc, chunk, MPI_DOUBLE,
          x,chunk,MPI_DOUBLE,0,MPI_COMM_WORLD);
33:     if(rk == 0) {
34:       diff = evalDiff(xold, x, m);
35:       memcpy(xold, x, m * sizeof(double));
36:     }
37:     MPI_Bcast(&diff,1,MPI_DOUBLE,0,
          MPI_COMM_WORLD);
38:     I ++;
39:   } while((diff >= eps) && (I <= MAX-
          ITERS));
```

Листинг 16. Параллельный вариант реализации метода Якоби: основной цикл

Последний фрагмент кода параллельного варианта метода Якоби выполняет печать результата и времени работы приложения (листинг 17).

```
40:   if(rk == 0) {
41:     t = MPI_Wtime() - t;
42:     printf("%d iterations consumed %lf
          sec\n",I,t);
43:   }
44:   MPI_Finalize();
45: }
```

Листинг 17. Параллельный вариант реализации метода Якоби: завершение

Заметим, что приведенная параллельная реализация будет корректно работать только при условии, что число строк матрицы `m` нацело делится на число процессоров `np`. Если это условие не выполнено, то возникают сложности с использованием функций

MPI_Scatter и MPI_Gather, требующих указания одинакового размера блока пересылаемых данных на всех процессах, участвующих в обмене данными. В этом случае полезными оказываются функции MPI_Scatterv (для рассылки) и MPI_Gatherv (для сбора), позволяющие пересылать блоки разной длины. В отличие от функций MPI_Scatter и MPI_Gather, эти функции позволяют указывать для каждого блока пересылаемых данных размер и смещение от начала буфера.

Важно заметить, что смещение от начала буфера указывается не в байтах, а в элементах пересылаемого типа данных. Это означает, что для функции MPI_Gatherv буфер для приема сообщений и для MPI_Scatterv буфер для отправки сообщений указывают на первый элемент массива однородных компонентов типа recvtype и sendtype соответственно.

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

sendbuffer – указатель на посылаемые данные (существенен только на процессе с рангом root);

sendcounts – массив, i-й элемент которого содержит число элементов в i-м пересылаемом блоке данных;

displs – массив, i-й элемент которого содержит смещение, измеренное в элементах типа sebdtype, начала i-го блока данных по отношению к началу буфера sendbuf;

sendtype – тип элементов пересылаемых данных;

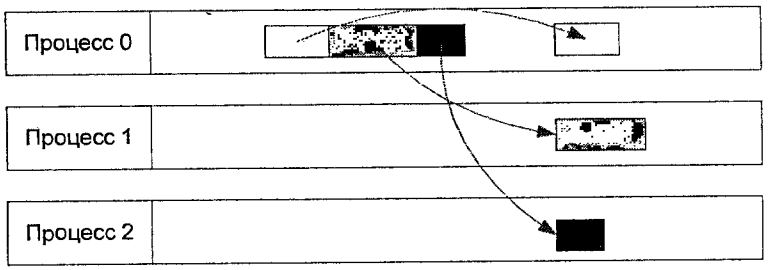
recvbuf – указатель на область памяти для сохранения принятых данных;

recvcount – количество элементов в блоке принимаемых данных;

recvtype – тип принимаемых элементов данных;

root – номер процесса, с которого рассылаются данные;

comm – коммуникатор, объединяющий синхронизуемые процессы.



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

sendbuf – указатель на посылаемые данные;

sendcount – количество элементов в пересылаемом блоке данных;

sendtype – тип элементов пересылаемых данных;

recvbuf – указатель на на область памяти для сохранения принятых данных (существеннен только на процессе с рангом root);

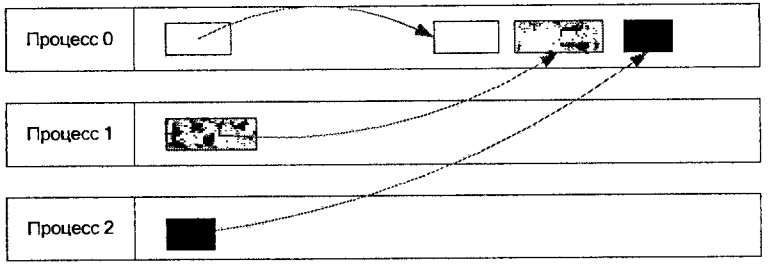
recvcounts – массив, i-й элемент которого задает число данных в i-м блоке принимаемых данных;

displs – массив, i-й элемент которого задает смещение, измеренное в элементах типа recvtype, i-го блока принимаемых данных от начала буфера recbuf;

recvtype – тип принимаемых элементов данных;

root – номер процесса, на котором выполняется прием данных;

comm – коммуникатор, объединяющий синхронизуемые процессы.



Рассмотренный параллельный вариант метода Якоби можно теперь переписать с применением функций MPI_Scatterv и MPI_Gatherv (листинг 18). В основном он отличается от предыдущего варианта необходимостью подготовки массивов смещений и размеров пересылаемых блоков для использования функциями MPI_Scatterv и MPI_Gatherv.

В цикле, расположенном в строках 26–32, производится вычисление размеров (массив chunks) и смещений (массив disps) для рассылки блоков матрицы B. Количество строк, пересылаемых в блоке, равняется величине chunk для всех процессов, кроме последнего, который получает все оставшиеся строки. Сама рассылка осуществляется функцией MPI_Scatterv в строке 34.

В строках 35–41 аналогично готовятся массивы размеров и смещений блоков для рассылки вектора g и сборки элементов вектора x. Заметим, что массивы g и x , а также локальные копии их фрагментов — массивы gloc и xloc — имеют одинаковую длину и тип элементов, поэтому для их рассылки и сборки используются одни и те же массивы смещений и размеров. Рассылка фрагментов массива gloc производится в строке 43 функцией MPI_Scatterv. Формирование вектора x на процессе с рангом 0 производится функцией MPI_Gatherv, вызываемой в строке 48.

```
1:  main(int argc, char* argv[])
2:  {
3:    int m,np,rk,chunk,i,I = 0,*chunks,*disps;
4:    double *B,*Bloc,*g,*gloc,*x,*xloc,*xold,
        t,diff,eps;
5:    MPI_Init(&argc, &argv);
6:    MPI_Comm_size(MPI_COMM_WORLD, &np);
7:    MPI_Comm_rank(MPI_COMM_WORLD, &rk);
8:    if(rk == 0) {
9:      m = atoi(argv[1]);
10:     eps = atof(argv[2]);
11:     chunk = m / np;
12:     B = (double*)malloc(m*m*sizeof(double));
13:   }
```



```
14:   MPI_Bcast(&m, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
15:   MPI_Bcast(&eps,1,MPI_DOUBLE,0,
        MPI_COMM_WORLD);
16:   MPI_Bcast(&chunk,1,MPI_INT,0,
        MPI_COMM_WORLD);
17:   chunks = (int*) malloc(np * sizeof(int));
18:   disps = (int*) malloc(np * sizeof(int));
19:   g = (double*)malloc(m * sizeof(double));
20:   x = (double*)malloc(m * sizeof(double));
21:   xold = (double*)malloc(m *
              sizeof(double));
22:   if(rk == 0){
23:     init(m, B, g, xold);
24:     t = MPI_Wtime();
25:   }
26:   for(i = 0; i < np; i ++) {
27:     disps[i] = i * chunk * m;
28:     if(i == (np - 1))
29:       chunks[i] = (m - (np - 1) * chunk)*m;
30:     else
31:       chunks[i] = chunk * m;
32:   }
33:   Bloc = (double*)malloc(chunks[rk]*
        sizeof(double));
34:   MPI_Scatterv(B,chunks,disps,
        MPI_DOUBLE,Bloc,chunks[rk],MPI_DOUBLE,
        0, MPI_COMM_WORLD);
35:   for(i = 0; i < np; i ++) {
36:     disps[i] = i * chunk;
37:     if(i == (np - 1))
38:       chunks[i] = m - (np - 1) * chunk;
39:     else
40:       chunks[i] = chunk;
41:   }
42:   gloc = (double*)malloc(chunks[rk]*
        sizeof(double));
43:   MPI_Scatterv(g,chunks,disps,MPI_DOUBLE,
        gloc,chunks[rk],MPI_DOUBLE,0,
        MPI_COMM_WORLD);
44:   xloc=(double*)malloc(chunks[rk]
        *sizeof(double));
```

```
45:   do {
46:     MPI_Bcast(xold,m,MPI_DOUBLE,0,
          MPI_COMM_WORLD);
47:     matvec(Bloc,gloc,xold,xloc,
          chunks[rk],m);
48:     MPI_Gatherv(xloc,chunks[rk],MPI_DOUBLE,
          x,chunks,disps,MPI_DOUBLE,0,
          MPI_COMM_WORLD);
49:     if(rk == 0) {
50:       diff = evalDiff(xold, x, m);
51:       printf("diff = %lf,
          eps = %lf\n",diff, eps);
52:       memcpy(xold, x, m * sizeof(double));
53:     }
54:     MPI_Bcast(&diff,1,MPI_DOUBLE,0,
          MPI_COMM_WORLD);
55:     I ++;
56:   } while((diff >= eps) && (I <= MAXITERS));
57:   if(rk == 0) {
58:     t = MPI_Wtime() - t;
59:     printf("%d iterations consumed %lf
          sec\n", I, t);
60:   }
61:   MPI_Finalize();
62: }
```

Листинг 18. Параллельный вариант реализации метода Якоби с использованием функций `MPI_Scatterv` и `MPI_Gatherv`

Заметим, что на каждой итерации цикла выполняется две операции коллективного взаимодействия: рассылка копий `xold` в строке 46 и сборка вектора `x` в строке 48. Эти две операции можно объединить в одну с помощью функции `MPI_Allgatherv`. В отличие от функции `MPI_Gatherv`, вектор собранных данных формируется на всех процессах коммуникатора, в рамках которого выполняется коллективная операция:


```
int MPI_Allgatherv(void *sendbuf, int send-
count, MPI_Datatype sendtype, void *recvbuf,
int *recvcounts, int *displs, MPI_Datatype recv-
type, MPI_Comm comm )
```

`sendbuf` – указатель на посылаемые данные;

`sendcount` – количество элементов в пересылаемом блоке данных;

`sendtype` – тип элементов пересылаемых данных;

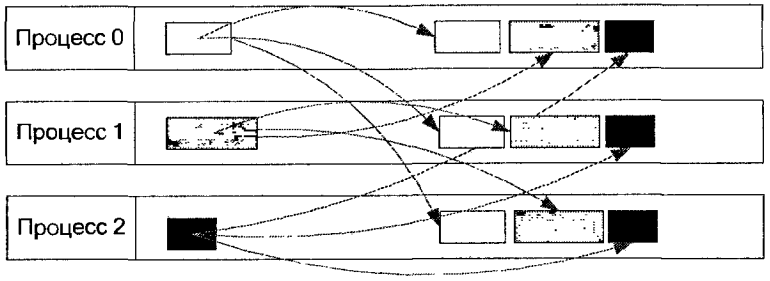
`recvbuf` – указатель на область памяти для сохранения принятых данных (существеннен только на процессе с рангом `root`);

`recvcounts` – массив, i-й элемент которого задает число данных в i-м блоке принимаемых данных;

`displs` – массив, i-й элемент которого задает смещение, измеренное в элементах типа `recvtype`, i-го блока принимаемых данных от начала буфера `recbuf`;

`recvtype` – тип принимаемых элементов данных;

`comm` – коммуникатор, объединяющий синхронизуемые процессы.



После применения функции `MPI_Allgatherv` основной цикл приобретет вид, представленный на листинге 19. Остальной код программы при этом останется неизменным.

```
1:  MPI_Bcast(xold,m,MPI_DOUBLE,0,
      MPI_COMM_WORLD);
2:  do{
3:    matvec(Bloc,gloc,xold,xloc,chunks[rk],m);
4:    MPI_Allgatherv(xloc,chunks[rk],MPI_DOUBLE,
```

```
            x,chunks,disps,MPI_DOUBLE,
            MPI_COMM_WORLD);
5:      diff = evalDiff(xold, x, m);
6:      memcpy(xold, x, m * sizeof(double));
7:      I ++;
8:  } while((diff >= eps) && (I <= MAXITERS));
```

Листинг 19. Основной цикл в параллельном варианте реализации метода Якоби с применением функции `MPI_Allgatherv`

В результате выполнения операции `MPI_Allgatherv` (строка 4) новое приближение `x` формируется из фрагментов `xloc` на всех процессах, участвующих в вычислениях (рис. 9). Поэтому норму разности приближений можно вычислять на всех процессах (строка 6). Также на каждом процессе производится копирование очередного приближения на место предыдущего (строка 7). Для обеспечения первой итерации цикла начальное значение предыдущего приближения рассылается по все процессам перед циклом (строка 1).

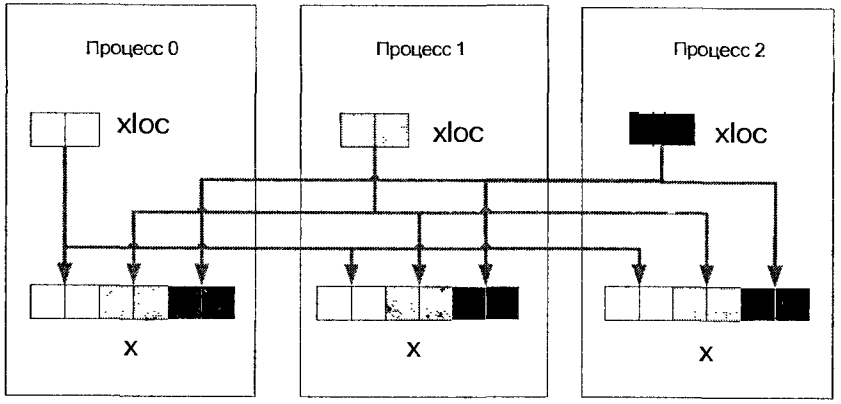


Рис. 9. Коммуникационная схема одной итерации цикла параллельного варианта метода Якоби с применением функции `MPI_Allgatherv`

На рис. 10 приведены результаты вычислительного эксперимента, проведенного на суперкомпьютере MVS 15000 BM для задачи размерностью 14000 и точности 0.0001. График показывает зависимость времени (в секундах) от количества участвующих

в вычислениях процессоров для варианта параллельной реализации с помощью функции `MPI_Allgatherv`. В качестве времени работы на одном процессоре взято время работы последовательного алгоритма на одном из узлов кластера.

График показывает, что время работы снижается по мере увеличения числа процессоров до 8. Далее наблюдается стабилизация времени работы, что свидетельствует о том, что накладные расходы на передачу данных стали сравнимыми со временем вычислений. Измерения показывают, что основное время занимает рассылка матрицы `B`. Если матрицу `B` не рассылать, а формировать на каждом процессе соответствующую ее часть, то ускорение будет значительно выше.

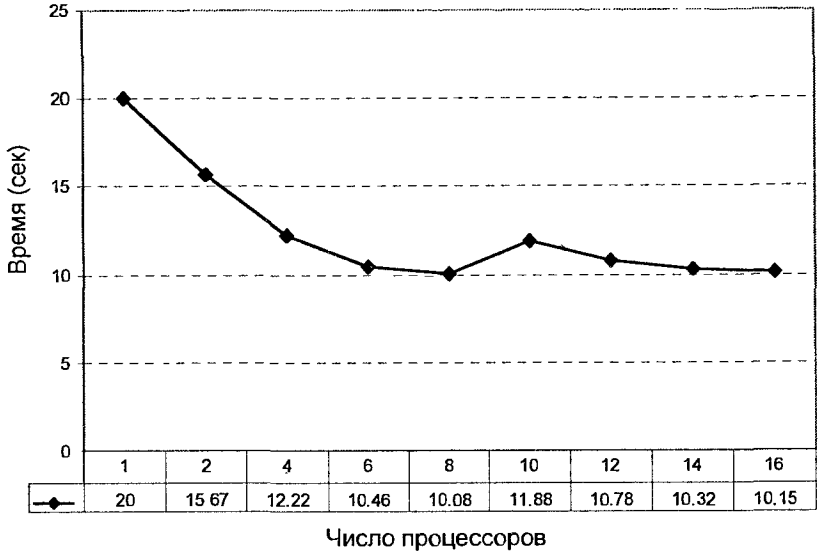


Рис. 10. График зависимости времени работы в секундах от количества процессоров, участвующих в вычислениях для параллельной реализации метода Якоби с помощью `MPI_Allgatherv`

Существует также ряд других коллективных операций, выполняющих сборку и рассылку данных. Информацию о них можно найти в [17].

1.11.3. Коллективные редуктивные операции

Рассмотренные в предыдущем разделе операции предназначены для пересылки данных между процессами. В некоторых случаях требуется не только переслать данные, но и выполнить над ними некоторые вычисления. Для таких случаев MPI предусматривает так называемые редуктивные операции, которые формируют результат с помощью применения заданной бинарной операции к операндам, пересылаемым с различных процессов.

Начнем рассмотрение редуктивных операций с функции `MPI_Reduce`. Данная функция вычисляет результат применения бинарной операции к операндам, расположенным на процессах, входящих в коммуникатор, по которому выполняется редукция. Если в выполнении операции принимают участие n процессов, $A_1,\ldots,A_n$ — операнды, $\circ$ — бинарная операция, то результатом выполнения редукции будет значение выражения $A_1 \circ A_2 \circ \ldots \circ A_n$. Операнды берутся из буферов отправки, а результат сохраняется на одном из процессов, номер которого передается в функцию через параметр `root`:

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

`sendbuf` – адрес буфера сообщения для отправки;

`recvbuf` – адрес буфера для сохранения результата операции;

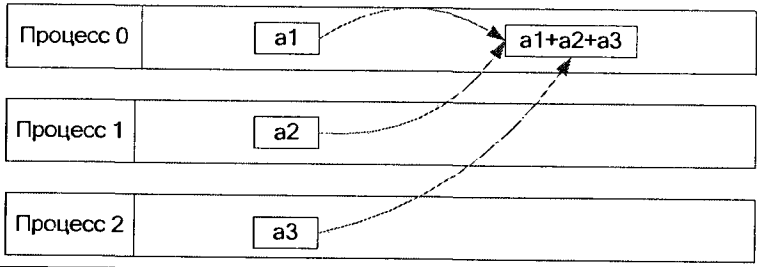
`count` – число элементов данных в буфере;

`datatype` – тип элемента данных в буфере;

`op` – бинарная операция;

`root` – номер процесса в коммуникаторе `comm`, на котором формируется результат операции;

`comm` – коммуникатор, по которому выполняется редукция.



В качестве бинарной операции может быть взята одна из предопределенных редуктивных операций MPI (или определенная пользователем операция). При этом операция должна быть ассоциативной, но может не быть коммутативной:

`MPI_MAX`	максимум
`MPI_MIN`	минимум
`MPI_SUM`	сумма
`MPI_PROD`	произведение
`MPI_LAND`	логическое «И»
`MPI_BAND`	побитовое «И»
`MPI_LOR`	логическое «ИЛИ»
`MPI_BOR`	побитовое «ИЛИ»
`MPI_LXOR`	логическое «исключающее ИЛИ»
`MPI_BXOR`	побитовое «исключающее ИЛИ»
`MPI_MAXLOC`	максимальное значение и его позиция
`MPI_MINLOC`	минимальное значение и его позиция

Операция, указанная при вызове `MPI_Reduce`, применяется ко всем элементам данных, расположенных в буферах операндов. Тип и число этих элементов должны быть одинаковыми на всех процессах, выполняющих редукцию. При этом операция применяется поэлементно ко всем компонентам операндов, и поэтому результат содержит столько же компонентов, сколько и операнды. Обязательным требованием является совместимость операции к типам операндов (табл. 2). Например, некорректным является применение побитовых операций к операндам одного из типов с плавающей запятой.

Проиллюстрируем использование редуктивной операции на примере программы численного интегрирования. Ранее был рассмотрен вариант реализации параллельного алгоритма с использованием функций точечного обмена сообщениями (листинг 8). Напомним как выглядела основная функция этой программы:

```
1:  double quad(MPI_Comm comm,double a,double b,
        int n)
2:  {
```

```
3:    int r, p, i;
4:    double h, sum;
5:    MPI_Status st;
6:    MPI_Comm_rank(comm, &r);
7:    MPI_Comm_size(comm, &p);
8:    sum = 0.0;
9:    h = (b - a) / n;
10:   for(i = r; i < n; i += p)
11:     sum += f(a + (i + 0.5) * h);
12:   sum *= h;
13:   if(r != 0)
14:     MPI_Send(&sum, 1, MPI_DOUBLE, 0, 0,
          comm);
15:   if(r == 0) {
16:     double s;
17:     for(i = 1; i < p; i ++) {
18:       MPI_Recv(&s, 1, MPI_DOUBLE, i, 0,
            comm, &st);
19:       sum += s;
20:     }
21:     return sum;
22:   }
23: }
```

Каждый процесс вычисляет отведенную ему часть интегральной суммы (строки 10–12), а затем все процессы, кроме нулевого, отправляют ему вычисленное значение (строки 13–14). Нулевой процесс принимает сообщения и суммирует полученные значения (строки 15–20). Операции в строках 13–21 можно заменить одним вызовом функции MPI_Reduce. В результате функция quad приобретет следующий вид:

```
1:  double quad(MPI_Comm comm,double a,double b,
      int n)
2:  {
3:    int r, p, i;
4:    double h, sum, result;
5:    MPI_Status st;
6:    MPI_Comm_rank(comm, &r);
```


Таблица 2. Совместимость типов и предопределенных операций в MPI

тип \ операция	MPI_MAX MPI_MIN	MPI_SUM MPI_PROD	MPI_LAND MPI_LOR MPI_LXOR	MPI_BAND MPI_BOR MPI_BXOR	MPI_MAXLOC MPI_MINLOC
MPI_INT, MPI_LONG MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG	+	+	+	+	-
MPI_INTEGER	+	+	+	+	-
MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_REAL	+	+	+	+	-
MPI_COMPLEX	-	+	-	-	-
MPI_BYTE	-	-	-	+	-
MPI_LOGICAL	-	-	+	-	-
MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT, MPI_SHORT_INT, MPI_LONG_DOUBLE_INT	-	-	-	-	+

```
7:    MPI_Comm_size(comm, &p);
8:    sum = 0.0;
9:    h = (b - a) / n;
10:   for(i = r; i < n; i += p)
11:     sum += f(a + (i + 0.5) * h);
12:   sum *= h;
13:   MPI_Reduce(&sum,&result,1,MPI_DOUBLE,
        MPI_SUM,0,comm);
14:  return result;
15: }
```

В строке 13 производится вызов функции `MPI_Reduce`, где в качестве адреса буфера операнда параметра подставляется адрес переменной `sum`, а в качестве адреса буфера результата – адрес переменной `result`. Параметры `1` и `MPI_DOUBLE` задают тип операндов (одно число плавающего типа). Константа `MPI_SUM` соответствует операции суммирования. После выполнения `MPI_Reduce`, значения переменной `sum` на разных процессах, входящих в коммуникатор `comm`, суммируются и результат сохраняется в переменной `result`. В результате проведенной замены объем кода сократился с 22 до 14 строк, программа стала более простой для восприятия.

1.12. Система типов в MPI

Процесс передачи сообщения в MPI состоит из следующих шагов (рис. 11):

1) данные считываются из области памяти, указанной на процессе отправителе, и из них формируется сообщение в соответствии с указанным типом;
2) сообщение передается процессу-получателю;
3) принятое сообщение распаковывается на процессе-получателе в соответствии с указанным типом и записывается в память по указанному адресу.

При отправке и приеме сообщения всегда указывается не только буфер, содержащий сообщение, но и тип данных, расположенных в этом буфере. Эта информация необходима для того,

чтобы правильно сформировать или распаковать сообщение. Библиотека MPI обладает достаточно развитой системой типов (табл. 3), которые делятся на три категории:

1) базовые типы (соответствуют скалярным типам языка);
2) производные типы (позволяют конструировать сложные типы данных на основе существующих типов данных);
3) нетипизированные сообщения (соответствуют произвольному сообщению).

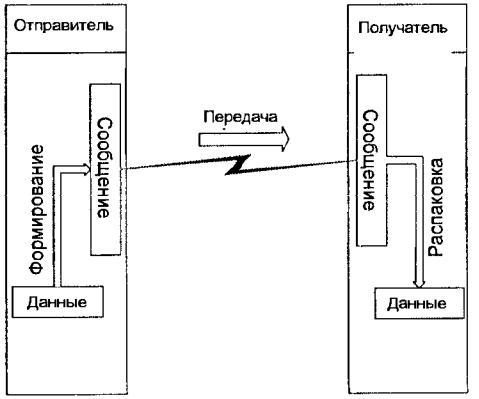


Рис. 11. Передача сообщения в MPI

Таблица 3. Категории типов в MPI

Базовый тип (язык Си)	Производный тип	Нетипизированное сообщение
MPI_CHAR	MPI_TYPE_CONTIGUOUS	MPI_PACKED
MPI_SHORT	MPI_TYPE_VECTOR	MPI_BYTE
MPI_INT	MPI_TYPE_HVECTOR	
MPI_LONG	MPI_TYPE_INDEXED	
MPI_UNSIGNED_CHAR	MPI_TYPE_HINDEXED	
MPI_UNSIGNED_SHORT	MPI_TYPE_STRUCT	
MPI_UNSIGNED		
MPI_UNSIGNED_LONG		
MPI_FLOAT		
MPI_DOUBLE		
MPI_LONG_DOUBLE		

1.12.1. Производные типы MPI

Базовые типы применяются при обмене сообщениями, представляющими собой последовательность элементов одного из скалярных типов. Каждому базовому типу MPI соответствует один скалярный тип используемого языка программирования (табл. 4).

Базовые типы позволяют осуществлять обмены массивами скалярных данных: в функциях передачи сообщений указывается тип и число элементов в сообщении. Для обмена более сложными данными требуются производные типы. В MPI предусмотрены следующие производные типы:

1. Массив — `MPI_TYPE_CONTIGUOUS`.
2. Вектор с регулярным шагом, кратным размеру элемента, — `MPI_TYPE_VECTOR`.
3. Вектор с шагом, измеряемым в байтах, — `MPI_TYPE_HVECTOR`.
4. Набор однотипных блоков разной длины с шагом, кратным размеру элемента, — `MPI_TYPE_INDEXED`.
5. Набор однотипных блоков разной длины с шагом, измеряемым в байтах, — `MPI_TYPE_HINDEXED`.
6. Структурный тип (совокупность элементов различных блоков, произвольно расположенных в памяти) — `MPI_TYPE_STRUCT`.

Таблица 4. Соответствие типов MPI и языка Си

Базовый тип MPI	Скалярный тип языка Си
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

На рис. 12 представлена диаграмма производных типов, упорядоченных по включению. Включение типа A в тип B обозначается стрелкой, ведущей из B в A, и означает, что диапазон данных, описываемых типом B, шире диапазона данных, описываемых типом A. В соответствии с этой диаграммой, наибольшими выразительными возможностями обладает структурный тип, а наименьший диапазон значений у типа массива `MPI_TYPE_CONTIGUOUS`.

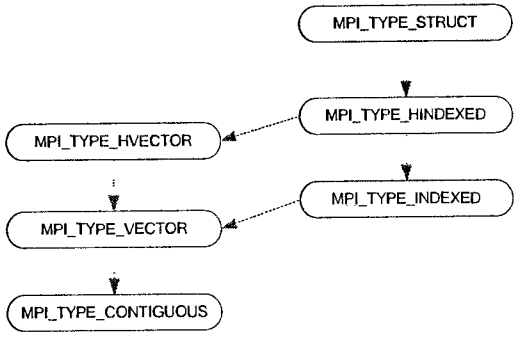


Рис. 12. Диаграмма производных типов MPI

Производные типы создаются с помощью так называемых конструкторов типов. В результате работы конструктора типа создается новый тип. Рассмотрим в качестве примера конструктор типа массив:

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

У данного конструктора типа три параметра: количество элементов массива (`count`), тип элементов (`oldtype`) и выходной параметр `newtype`, по адресу которого записывается созданный тип.

Перед использованием в качестве аргумента в функциях передачи данных созданный тип должен быть зарегистрирован в системе с помощью функции `MPI_Type_commit`. После того как все операции с данным типом выполнены, следует освободить выделенные ресурсы с помощью функции `MPI_Type_`

free. Обе функции принимают на вход единственный аргумент — обрабатываемый тип.

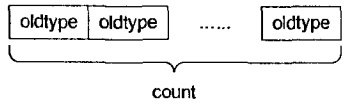
Важно отметить следующие особенности регистрации и освобождения типа:

- если тип используется для построения другого производного типа, но не используется в пересылках непосредственно, то регистрацию производить не нужно;
- освобождение типа не затрагивает производные типы, созданные на его основе.

---

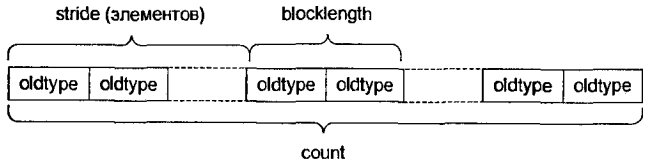
1. `int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Создает новый тип newtype, соответствующий последовательности подряд расположенных объектов типа oldtype, содержащей count элементов:



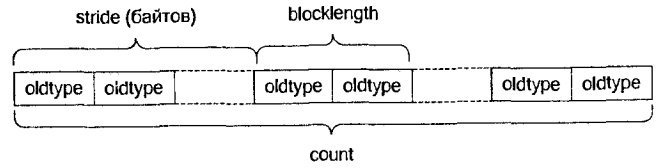
2. `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Создает новый тип newtype, соответствующий последовательности блоков, содержащей count элементов. Каждый элемент представляет собой блок, состоящий из blocklength элементов типа oldtype. Расстояние между первыми элементами блоков составляет stride элементов типа oldtype:



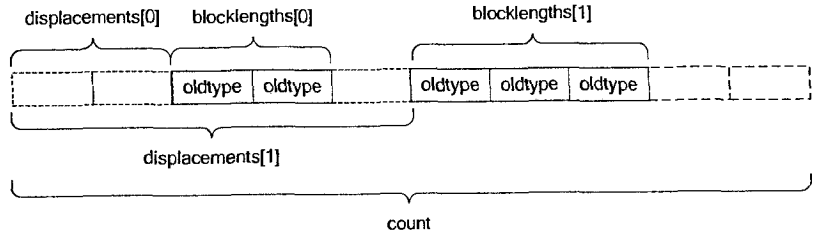
3. `int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Создает новый тип newtype, соответствующий последовательности блоков, содержащей count элементов. Каждый элемент представляет собой блок, состоящий из blocklength элементов типа oldtype. Расстояние между первыми элементами блоков составляет stride байтов:



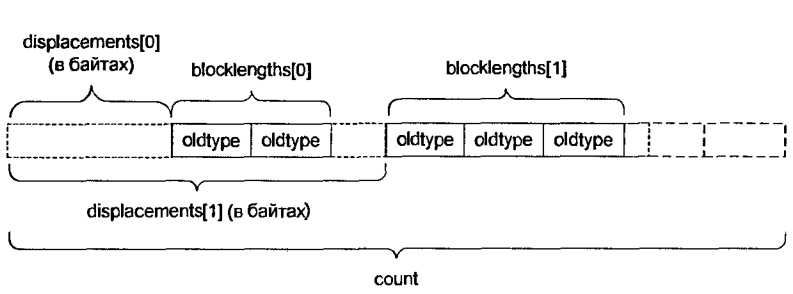
4. `int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Создает новый тип newtype, соответствующий последовательности блоков, содержащей count элементов. Каждый блок состоит из элементов типа oldtype. Число элементов в блоке *i* определяется значением array_of_blocklengths[i]. Смещение от начала буфера, измеряемое в элементах типа oldtype, задается значением array_of_displacements[i].



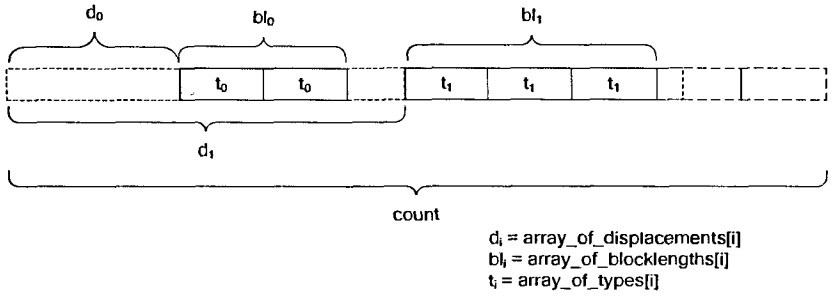
5. `int MPI_Type_hindexed(int count, int *array_of_blocklengths, int *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype *newtype)`

Создает новый тип newtype, соответствующий последовательности блоков, содержащей count элементов. Каждый блок состоит из элементов типа oldtype. Число элементов в блоке *i* определяется значением array_of_blocklengths[i]. Смещение от начала буфера, измеряемое в байтах, задается значением array_of_displacements[i].



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

Создает новый тип `newtype`, соответствующий последовательности блоков, содержащей `count` элементов. При этом блок `i` смещен от начала буфера на `array_of_displacements[i]` байтов и содержит количество элементов типа `array_ot_types[i]`, равное `array_of_blocklengths[i]`:



Рассмотрим применение производных типов для пересылки матрицы с транспонированием (рис. 13). Заметим, что матрица может быть представлена как совокупность строк или как совокупность столбцов. Транспонирование эквивалентно отображению строки с номером *i* в столбец с номером *i*. Эту операцию можно совместить с пересылкой данных, если на процессе-отправителе задать тип матрицы как совокупность строк, а на процессе-получателе — как совокупность столбцов.

Перейдем теперь к рассмотрению реализации описанного алгоритма на MPI (листинг 20). Процесс 0 заполняет матрицу `A` с помощью функции `fill_matrix` и выводит ее содержимое на экран (строки 8–10). После чего на нем формируется и регистрируется тип для отправки матрицы (строки 11–12). Этот тип соответствует массиву элементов типа `double` с числом элементов `N×N` (где `N` — размерность матрицы), расположенных в памяти подряд. Далее матрица пересылается процессу 1 с помощью вызова `MPI_Send` в строке 13.

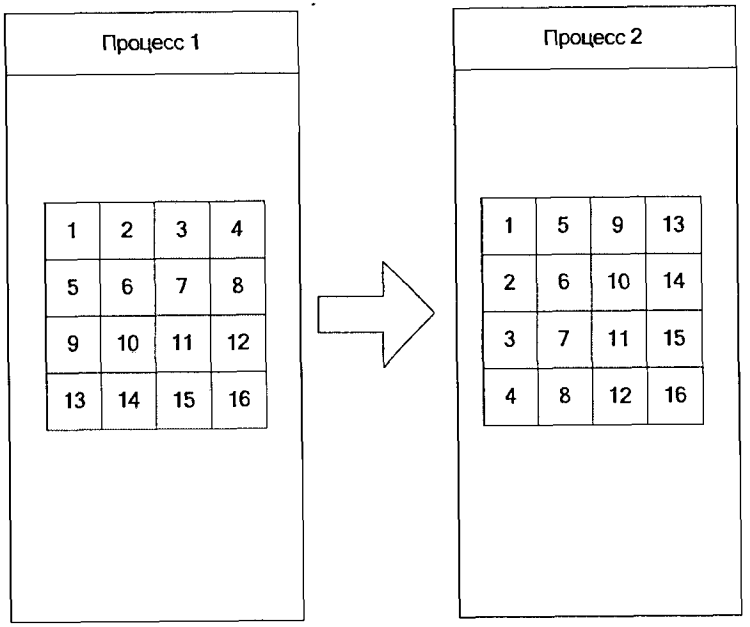


Рис. 13. Пересылка матрицы с транспонированием

Процесс 1 формирует тип, соответствующий представлению матрицы как совокупности столбцов. Это делается в два этапа. На первом этапе формируется тип «вектор», соответствующий одному столбцу (строка 16). Величина шага между элементами вектора составляет `N`, так как массив в языке Си располагается в памяти по строкам и последовательные элементы столбца отстоят друг от друга на `N` элементов. Начальные элементы столбцов рас-

положены подряд в первой строке. Расстояние между ними составляет в точности размер в байтах одного элемента типа double. Матрицу можно описать векторным типом, элементами которого являются столбцы, расположенные в памяти с шагом sizeof(double) байтов. Для создания такого типа применяется конструктор MPI_Type_hvector в строке 17.

```
1:  main(int argc, char* argv[]) {
2:    int r;
3:    MPI_Status st;
4:    MPI_Datatype typ, typ1;
5:    MPI_Init(&argc, &argv);
6:    MPI_Comm_rank(MPI_COMM_WORLD, &r);
7:    if(r == 0) {
8:      fill_matrix();
9:      printf("\n Source:\n");
10:     print_matrix();
11:     MPI_Type_contiguous(N * N, MPI_INT,
          &typ);
12:     MPI_Type_commit(typ);
13:     MPI_Send(&(A[0][0]),1,typ,1,0,
          MPI_COMM_WORLD);
14:     MPI_Type_free(&typ);
15:   } else if(r == 1){
16:     MPI_Type_vector(N, 1, N, MPI_INT, &typ);
17:     MPI_Type_hvector(N,1,sizeof(int),typ,
          &typ1);
18:     MPI_Type_commit(&typ1);
19:     MPI_Recv(&(A[0][0]),1,typ1,0,0,
         MPI_COMM_WORLD,&st);
20:     printf("\n Transposed:\n");
21:    print_matrix();
22:    MPI_Type_free(&typ);
23:    MPI_Type_free(&typ1);
24:   }
25:   MPI_Finalize();
26: }
```

Листинг 20. Пересылка матриц с транспонированием — реализация на MPI

Описание вспомогательных функций fill_matrix и print_matrix не приводится, так как эти функции очень просты и не представляют интереса в ракурсе рассматриваемой тематики. После того как созданные типы были использованы в пересылках, память освобождается с помощью вызова MPI_Type_free (строки 14, 22, 23).

В результате работы для матрицы размерности 3×3 на экран будет выведена следующая информация:

```
 Source:
0 1 2
3 4 5
6 7 8
 Transposed:
0 3 6
1 4 7
2 5 8
```

1.12.2. Правила соответствия типов при обменах

Любой тип MPI описывает структуру данных в памяти, которую в общем виде можно представить как совокупность пар *(type, disp)*, где *type* – некоторый базовый тип, а *disp* – его смещение от начала буфера. Таким образом, каждому MPI-типу ставится в соответствие последовательность вида $\{(type_1, disp_1),\ldots,(type_n, disp_n)\}$, называемая картой типа. Последовательность $\{type_1,\ldots, type_n\}$, состоящая только из базовых типов, называется сигнатурой типа.

Типы в MPI-программе обладают достаточно сложной семантикой, и технология их корректного употребления содержит много нюансов. Следующие три основных правила соответствия типов позволяют избежать большинства ошибок, связанных с обработкой типов в MPI-программах.

Правило 1. Тип данных в буфере для отправки должен соответствовать MPI-типу сообщения, указанного в качестве параметра при вызове функции: по адресу, смещенному на $disp_i$ от начала буфера, должны располагаться данные типа $type_i$.

Правило 2. MPI-тип принятого сообщения должен соответствовать MPI-типу, указанному в функции приема. В случае точечных обменов это означает, что сигнатура типа пришедшего

сообщения должна быть начальной подпоследовательностью сигнатуры типа, указанного в качестве аргумента функции приема. В случае коллективных обменов сигнатура типа принятого сообщения должна совпадать с сигнатурой типа, указанного в функции. При этом карты типов на стороне отправителя и на стороне получателя могут не совпадать.

Правило 3. Размер и выравнивание адресов в буфере для приема должны соответствовать карте MPI-типа, указанного в функции приема сообщения: запись элемента типа $type_i$ по адресу, смещенному на $disp_i$ байтов от начала буфера для приема сообщения, не должна приводить к переполнению буфера или нарушению правил выравнивания для типов базового языка.

Из этих правил существует единственное исключение — передача данных типа MPI_PACKED. Сообщение, посланное как MPI_PACKED, может быть принято как сообщение типа, сигнатура которого соответствует данным, упакованным внутри сообщения. И наоборот – любое сообщение можно получить, используя тип MPI_PACKED. Более подробно этот случай рассматривается в следующем разделе. Рассмотрим случаи некорректного употребления MPI-типов:

---

1. Нарушено правило 1 – MPI-тип соответствует последовательности двух элементов целого типа, а буфер содержит два элемента типа с плавающей точкой:

```
double a[2];
MPI_Send(a, 2, MPI_INT, ...)
```

2. Нарушено правило 2 – тип посланного сообщения имеет сигнатуру (MPI_INT, MPI_INT), а в функции приема указан тип с сигнатурой (MPI_INT):

```
MPI_Send(a, 2, MPI_INT,...)
MPI_Recv(b, 1, MPI_INT,...)
```

3. Правило 2 не нарушено – сигнатура типа посылаемого сообщения (MPI_INT, MPI_INT) является начальной подпоследовательностью сигнатуры типа, указанного в функции приема (MPI_INT, MPI_INT, MPI_INT):

```
MPI_Send(a, 2, MPI_INT,...)
MPI_Recv(b, 3, MPI_INT,...)
```


4. Нарушено правило.3 — размер буфера, указанного в функции приема, меньше, чем требуется для хранения данных, соответствующих указанному MPI-типу:

```
int a;
MPI_Recv(a, 2, MPI_INT, ...)
```

Упаковка и распаковка сообщений

В типичной ситуации функции формировании сообщения для передачи и его распаковки при приеме берет на себя библиотека MPI. В то же время предусмотрены функции MPI_Pack и MPI_Unpack, предоставляющие пользователю возможность формировать и распаковывать сообщения самостоятельно. Применение данных функций оправдано, если требуется отделить операцию формирования от операции отправки сообщения. Такая тактика может быть полезна с точки зрения повышения производительности программ за счет уменьшения частоты обменов и снижения накладных расходов на передачу данных.

Функция MPI_Pack предназначена для запаковки сообщений и имеет следующий заголовок:

```
int MPI_Pack(void* inbuf, int incount,
    MPI_Datatype datatype, void *outbuf,
    int outsize, int *position, MPI_Comm comm)
```

Буфер, содержащий данные, предназначенные для запаковки, расположен по адресу inbuf. Число элементов и их тип передаются через параметры incount и datatype. Параметры outbuf, outsize и position задают адрес начала, размер буфера в байтах и позицию в нем соответственно. Последним параметром указывается коммуникатор comm, который будет использован для передачи сообщения. В результате выполнения этой операции данные расположенные в inbuf запаковываются в буфер outbuf, начиная с позиции, номер которой передается через указатель position. После завершения работы функции значение по адресу position содержит номер первого свободного байта буфера outbuf.

Функция распаковки данных MPI_Unpack имеет следующий прототип:

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

Эта функция распаковывает данные, расположенные в буфере inbuf размера insize, начиная с позиции, значение которой хранится по адресу position. Распаковываются outcount элементов типа datatype и помещаются в буфер outbuf. В результате выполнения функции позиция изменяется: после завершения вызова функции по адресу position хранится смещение первого байта, следующего за распакованными данными, от начала буфера inbuf.

Пример применения функций упаковки и распаковки данных приведен на листинге 21. В строках 11–12 определяется размер буфера и выделяется соответствующий объем памяти. Далее в процессе с номером 0 в цикле производится запаковка данных (строка 16) — последовательно запаковывается N последовательных целых чисел. В строке 19 производится отправка сообщения. Указывается тип MPI_PACKED и размер в байтах, который совпадает с номером последней свободной позиции в буфере (переменная pos).

Другой процесс с номером 1 (предполагается, что в выполнении программы участвуют два процесса) принимает данные в строке 22. При приеме указывает тип MPI_PACKED и в качестве размера указывается размер буфера buf. После этого в цикле производится распаковка данных и их вывод на печать (строки 24–25).

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #include <mpi.h>
4:  #define N 3
5:  main(int argc, char* argv[])
6:  {
7:    int r, i, sz, pos = 0, a;
8:    void* buf;
9:    MPI_Init(&argc, &argv);
10:   MPI_Comm_rank(MPI_COMM_WORLD, &r);
11:   MPI_Pack_size(N,MPI_INT,MPI_COMM_WORLD,
        &sz);
```



```
12:   buf = (void*) malloc(sz);
13:   if(r == 0){
14:     a = 1;
15:     for(i = 0; i < N; i ++) {
16:       MPI_Pack(&a,1,MPI_INT,buf,sz,&pos,
            MPI_COMM_WORLD);
17:       a ++;
18:     }
19:     MPI_Send(buf,pos,MPI_PACKED,1,0,
          MPI_COMM_WORLD);
20:   } else {
21:     MPI_Status st;
22:     MPI_Recv(buf,sz,MPI_PACKED,0,0,
          MPI_COMM_WORLD, &st);
23:     for(i = 0; i < N; i ++) {
24:       MPI_Unpack(buf,sz,&pos,&a,1,
            MPI_INT,MPI_COMM_WORLD);
25:       printf("%d ", a);
26:     }
27:   }
28:   MPI_Finalize();
29: }
```

Листинг 21. Запаковка и пересылка данных

Следует отметить, что при приеме данных, посланных с использованием типа MPI_PACKED, можно указывать не только тип MPI_PACKED, но и тип, соответствующий запакованным данным. В примере, представленном на листинге 22, данные посылаются тем же способом, что и в рассмотренном примере, а принимаются с применением типа MPI_INT (строка 23).

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #include <mpi.h>
4:  #define N 3
5:  main(int argc, char* argv[])
6:  {
7:    int r, i;
```

```
8:    MPI_Init(&argc, &argv);
9:    MPI_Comm_rank(MPI_COMM_WORLD, &r);
10:   if(r == 0){
11:     int sz, pos = 0, a = 1;
12:     void* buf;
13:     MPI_Pack_size(N,MPI_INT,MPI_COMM_WORLD,
          &sz);
14:     buf = (void*) malloc(sz);
15:     for(i = 0; i < N; i ++) {
16:       MPI_Pack(&a,1,MPI_INT,buf,sz,&pos,
            MPI_COMM_WORLD);
17:       a ++;
18:     }
19:     MPI_Send(buf,pos,MPI_PACKED,1,0,
          MPI_COMM_WORLD);
20:   } else {
21:     MPI_Status st;
22:     int A[N];
23:     MPI_Recv(A,N,MPI_INT,0,0,MPI_COMM_WORLD,
          &st);
24:     for(i = 0; i < N; i ++)
25:       printf("%d ", A[i]);
26:   }
27:   MPI_Finalize();
28: }
```

Листинг 22. Прием запакованных данных с применением базового типа MPI

2. Многопоточное программирование

2.1. Процессы и потоки в многозадачной операционной системе

В этом разделе приводятся основные сведения об организации потоков и процессов в операционной системе, существенные для дальнейшего изложения. Следует отметить, что приводимое описание является кратким и упрощенным, так как не является основным предметом учебника. Подробное изложение материала по вопросам программной архитектуры и принципам функционирования операционных систем можно найти в монографии [2].

2.1.1. Процессы

Одной из основных функций операционной системы (ОС) является организация выполнения задач. Для этого в ОС предусмотрено такое понятие, как *процесс*. Можно дать много различных определений этому понятию. Достаточно общее определение звучит так: процесс – это экземпляр выполняемой программы.

Другими словами, любая программа выполняется в рамках некоторого процесса ОС. С процессом связаны некоторые ресурсы, используемые программой во время выполнения, важнейшим из которых является *адресное пространство задачи*. В нем размещаются код программы и все переменные.

Большинство современных ОС поддерживают многозадачность: одновременно могут выполняться несколько процессов. Если в системе установлено однопроцессорное устройство то одновременность выполнения достигается с помощью механизма *квантования времени*: каждый процесс выполняется в течение некоторого, как правило, достаточно короткого промежутка времени, после чего ресурсы процессора передаются другому процессу (рис. 14). Такой механизм создает иллюзию одновременной работы, но ускорения вычислений не происходит, так как используется только один процессор.

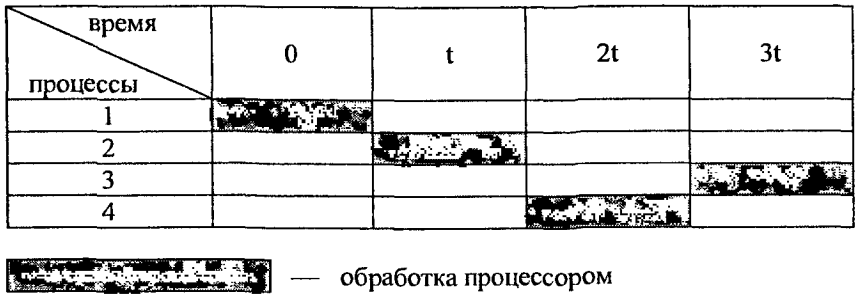


Рис. 14. Планирование процессов в системе с одним процессором

Ситуация становится иной, если в системе установлено несколько процессоров, либо процессор содержит несколько вычислительных ядер и может поддерживать одновременное выполнение нескольких потоков инструкций. В этом случае между работающими процессами распределяется ресурсы нескольких процессоров, и инструкции различных процессов получают возможность выполняться в один и тот же момент времени (рис. 15).

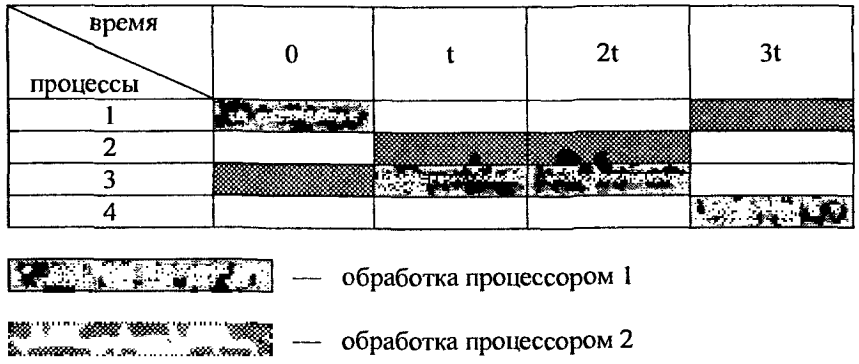


Рис. 15. Планирование процессов на многопроцессорной системе

Процессы применяются для параллельной обработки данных как на многопроцессорных системах с общей памятью, так и на системах с распределенной памятью. Для обеспечения взаимодействия процессов, выполняющихся на одном компьютере, применяются механизмы *межпроцессного взаимодействия* [2].

Если процессы выполняются на разных процессах, то наиболее естественным способом взаимодействия становится прием и отправка сообщений. Этой парадигме, в частности, соответствует основное средство программирования систем с распределенной памятью — среда Message Passing Interface (MPI) [17].

Параллельные приложения для систем с общей памятью на основе процессов, как правило, разрабатываются с помощью MPI. Такой подход удобен, так как программа получается переносимой и может выполняться на системах как с общей, так и с распределенной памятью. При этом эффективность выполнения MPI-программы на системе с общей памятью может быть достаточно высокой. Это объясняется тем, что реализация MPI для систем с общей памятью, как правило, оптимизирована с учетом особенностей архитектуры: операции сетевого взаимодействия в реализации операций обмена заменяются копированием данных.

Следует отметить, что более естественным, чем обмен сообщениями, для систем с общей памятью является взаимодействие с помощью записи и чтения данных из памяти. В случае процессов применяется взаимодействие через разделяемые сегменты памяти. При этом часть адресного пространства двух процессов отображается на одну и ту же физическую область в памяти. Процессы осуществляют взаимодействие с помощью чтения и записи данных в этой области. Такой метод, часто используемый при разработке системного и прикладного ПО, не получил широкого распространения в практике параллельного программирования.

Новые перспективы для разработки параллельных программ для систем с общей памятью обозначились с появлением *потоков*.

2.1.2. Потоки в операционной системе

Для многих задач характерно наличие частей, которые могут функционировать параллельно. В некоторых случаях наличие возможности параллельной обработки данных в рамках одного процесса не только возможно, но и целесообразно. Например, обработка непересекающихся секций массива может проводиться параллельно. Решение этой задачи с помощью нескольких процессов, разделяющих общий сегмент данных, является достаточно тяжеловесным и требует существенных затрат на реализацию.

Подобные рассуждения привели к созданию механизма *потоков*. Сам термин является переводом английского слова *thread*. Иногда его переводят как *нить* или оставляют без перевода, употребляя термин *тред*.

Поток представляет собой самостоятельную цепочку последовательно выполняемых операторов программы, соответствующих некоторой подзадаче. В адресном пространстве одного процесса может выполняться несколько потоков. При этом все глобальные данные являются общими для различных потоков, в то время как локальные данные индивидуальны.

В случае программ на языке Си к глобальным относятся все данные, расположенные в статической памяти (статические, внешние переменные), и в области динамической памяти, выделенные с помощью функций `malloc`, `calloc` и других функций управления динамической памятью. К локальным данным относятся автоматические переменные, т.е. переменные, объявленные внутри функций или составных блоков. Пример программы, в котором присутствуют как глобальные, так и локальные данные, приведен на листинге 23.

```
1:  int a;
2:  main()
3:  {
4:    double b;
5:    static int k;
6:    char* p;
7:     p = (char*)malloc(10);
8:  }
```

В данном примере переменные `a` и `k`, а также область памяти, выделенная в строке 7, являются глобальными данными. Переменные `b` и `p` представляют собой пример локальных данных.

Листинг 23. Локальные и глобальные данные в программе

Таким образом, главным отличием потоков от процессов является то, что каждый процесс имеет собственное адресное пространство, а потоки создаются в адресном пространстве существующего процесса. Соответственно, в адресном пространстве одного процесса может выполняться одновременно несколько потоков.

2.1.3. Потоки и разработка параллельных программ

Для систем с общей памятью потоки представляются более удачной программной платформой, чем процессы с точки зрения разработки параллельных программ. Однако создание и управление потоками с помощью системных вызовов, так же как и аналогичные функции для процессов, являются средством достаточно низкого уровня. Поэтому для разработки параллельных программ было бы желательно иметь более развитый инструментарий. До недавнего времени таких средств, получивших всеобщее признание, не существовало. Поэтому, многопоточное программирование стало одним из основных средств разработки параллельных приложений для систем с общей памятью.

В последнее время также получил распространение пакет OpenMP, который относится к инструментам высокого уровня, предназначенным для разработки параллельных программ на системах с общей памятью. Этот пакет также будет подробно рассмотрен в одном из следующих разделов.

Важно отметить, что выигрыш в производительности при параллельной обработке данных с помощью потоков возможен только в том случае, если операционная система поддерживает независимое планирование потоков, т. е. при создании двух потоков в рамках одного процесса для получения ускорения необходимо, чтобы оба потока выполнялись разными процессорными устройствами, что невозможно, если весь процесс рассматривается планировщиком как единое целое. Первоначально такая реализация доминировала: потоки реализовывались на уровне приложения. При этом польза от них все равно была невелика в случае, например, операций асинхронного взаимодействия с устройствами ввода-вывода. Однако в рамках такой реализации параллельная обработка данных не ускорялась.

Ситуация была исправлена, и в современных ОС предоставляется специальная поддержка для планирования потоков: в ядро ОС вводится понятие потока как объекта планирования. При создании потока приложение обращается к ядру с помощью механизма системных вызовов и запрашивает создание потока. Ядро создает поток и помещает его в список для планирования. В результате параллельно могут обрабатываться как потоки из разных процессов, так и потоки одного процесса.

Описанная схема называется *один-в-один*, т. е. одному потоку приложения ставится в соответствие один поток ядра. Схема один-в-один достаточно широко распространена. Иногда применяют более сложную схему *двухуровнего планирования*, при которой несколько потоков приложения отображаются на один поток ядра. Эта схема обладает большей гибкостью, но и требует больших накладных расходов, связанных с наличием нескольких уровней планирования.

2.1.4. Интерфейс POSIX Threads

Существуют различные реализации прикладного интерфейса для управления потоками в разных средах. Усилиями международного сообщества разработчиков программного обеспечения была созданная единая спецификация, которая получила статус стандарта и вошла как часть в стандарт прикладного интерфейса операционной системы IEEE Std 1003.1–2001. Часто этот интерфейс управления потоками называют POSIX Threads или сокращенно pthreads. Подробное описание можно найти на странице консорциума Open Group, посвященной открытому стандарту для Unix-систем [7]. Далее в главе рассматривается минимальный набор функций, необходимых для создания многопоточного приложения.

2.2. Создание и завершение потока в интерфейсе POSIX Threads

Сразу после запуска процесс состоит в точности из одного потока. Если программа не производит никаких специальных действий по запуску других потоков, то этот единственный поток будет выполняться до завершения процесса. Инициировать выполнение дополнительных потоков в рамках того же процесса можно с помощью вызова специальной функции `pthread_create`:

```
int pthread_create(pthread_t * iaddr,
pthread_attr_t * attr,void *(*start_routine)
(void*), void *arg);
```


`iaddr` – идентификатор создаваемого потока (выходной параметр);

`attr` – атрибуты создаваемого потока;

`start_routine` – адрес функции, которая будет выполняться в создаваемым потоком;

`arg` – указатель на аргументы функции.

Создает поток с атрибутами `attr`. Если в качестве `attr` передается NULL, то применяются атрибуты по умолчанию. Выполнение потока начинается с функции `start_routine`. Последний аргумент `arg` – параметр, передаваемый в функцию. Функция возвращает 0 в случае успешного завершения или код ошибки в противном случае.

В результате выполнения функции создается новый поток, который начинает свое выполнение с точки входа в функцию `start_routine`, адрес которой также передается в качестве аргумента при вызове. Функция, с которой созданный поток начинает свое выполнение, должна принимать в точности один аргумента типа `void*` и должна возвращать значение также типа `void*`. Фактическое значение передаваемого в функцию аргумента указывается при вызове `pthread_create`. Идентификатор созданного потока записывается по адресу `idaddr`, передаваемому в качестве первого аргумента.

Так же как и другие функции библиотеки pthreads, функция `pthread_create` возвращает значение целого типа, которое равняется нулю в случае успешного завершения или содержит код ошибки в противном случае.

Рассмотрим в качестве примера такую функцию, которая распечатывает строку, переданную ей в качестве аргумента (листинг 24). Функция возвращает нулевой указатель.

```
1:  void* hello(void* arg)
2:  {
3:    printf("Hi from thread %s \n",
        (char*)arg);
4:    return NULL;
5:  }
```

Листинг 24. Функция печати приветствия

Подождать момент завершения потока и получить возвращаемое значение можно с помощью функции pthread_join:

```
int pthread_join(pthread_t thread, void **value_
ptr)
```

thread – идентификатор потока;

value_ptr – указатель на значение, возвращаемое функцией потока; функция возвращает 0 в случае успешного завершение и код ошибки в противном случае.

Необходимо всегда вызывать эту функцию. В противном случае может произойти так, что основной поток и вместе с ним весь процесс завершит выполнение. При этом остальные потоки будут принудительно завершены. Кроме того, если созданный поток не был обработан с помощью pthread_join, то ресурсы, выделенные для него системой, считаются занятыми даже после его завершения. Поэтому пренебрежение данной функцией может привести к переполнению системных ресурсов.

В качестве примера использования приведенных функций рассмотрим простейшую программу, которая запускает по потоку на каждый аргумент командной строки (листинг 25). Поток распечатывает переданный ему аргумент командной строки.

В начале программы необходимо подключить заголовочный файл pthread.h, содержащий необходимые определения для работы с потоками, в строках 4–8 определяется рассмотренная ранее функция hello. В строке 16 выделяется память для массива идентификаторов потоков, длина которого равна числу аргументов командной строки. Запуск потоков производится в цикле по аргументам командной строки в строках 15–21. Также в цикле (строки 22–28) основной поток ожидает завершение запущенных потоков. Отметим необходимость проверки кода завершения функций управления потоками: такая проверка позволяет обнаруживать ошибочные ситуации и своевременно их обрабатывать. В дальнейших примерах эта проверка часто не будет приводиться из соображений экономии места и упрощения восприятия кода.


```
1:  #include <pthread.h>
2:  #include <string.h>
3:  #include <stdio.h>
4:  void* hello(void* arg)
5:  {
6:    printf("Hi from thread %s \n",
        (char*)arg);
7:    return NULL;
8:  }
9:  main(int argc, char* argv[])
10: {
11:   int i, rc, *status;
12:   pthread_t *threads;
13:   if(argc > 1) {
14:     threads = (pthread_t*)malloc((argc-1)*
          sizeof(pthread_t));
15:     for(i = 1; i < argc; i ++) {
16:       rc=pthread_create(&(threads[i-1]),
            NULL,hello,argv[i]);
17:       if(rc != 0) {
18:         fprintf(stderr, "Error creating
              thread: code %d\n", rc);
19:         exit(-1);
20:       }
21:     }
22:     for(i = 1; i < argc; i ++) {
23:       rc = pthread_join(threads[i - 1],
            NULL);
24:       if(rc != 0) {
25:         fprintf(stderr, "Error joining
              thread:
              code %d\n" ,rc);
26:         exit(-1);
27:       }
28:     }
29:   }
30: }
```

Листинг 25. Простейшая многопоточная программа

Рассмотрим подробно процесс компиляции и запуска данной программы. Для сборки необходимо подключить библиотеку потоков. Обычно в системе Unix для этого следует указывать флаг компиляции `-lpthread`. Если файл с кодом программы назывался `hw.c`, то для получения выполняемого файла `hw` надо выполнить следующую команду:

```
cc -o hw hw.c -lpthread
```

Выполняемый файл запускается в обычном порядке из командной строки. В результате на терминал будет выведено:

```
$./hw sun moon
Hi from thread sun
Hi from thread moon
```

Функция потока может возвращать значение типа указатель. Значение возвращается либо при помощи оператора `return`, либо через аргумент функции `pthread_exit`:

```
void pthread_exit(void *value_ptr)
```

`value_ptr` – указатель на область памяти, содержащую возвращаемое значение.

Функция `pthread_exit` позволяет завершать поток не только из главной функции потока, но и из вызываемых функций. Так как все локальные данные потока уничтожаются при его завершении, то возвращаемый указатель не может указывать на локальные данные функции, а должен содержать адрес глобальных данных.

2.3. Многопоточная программа численного интегрирования

2.3.1. Описание программы

Небольшой набор функций, рассмотренный в предыдущем разделе, оказывается достаточным для разработки простейшего параллельного приложения, выполняющего вычисления определенного интеграла уже знакомым нам методом прямоугольников (разд. 1.4).

Приложение работает по следующей схеме: основной поток запускает несколько рабочих потоков, каждый из которых вычисляет часть суммы прямоугольников, составляющих приближение. Полученные результаты сохраняются в массиве, элементы которого суммируются на основном потоке (рис. 16).

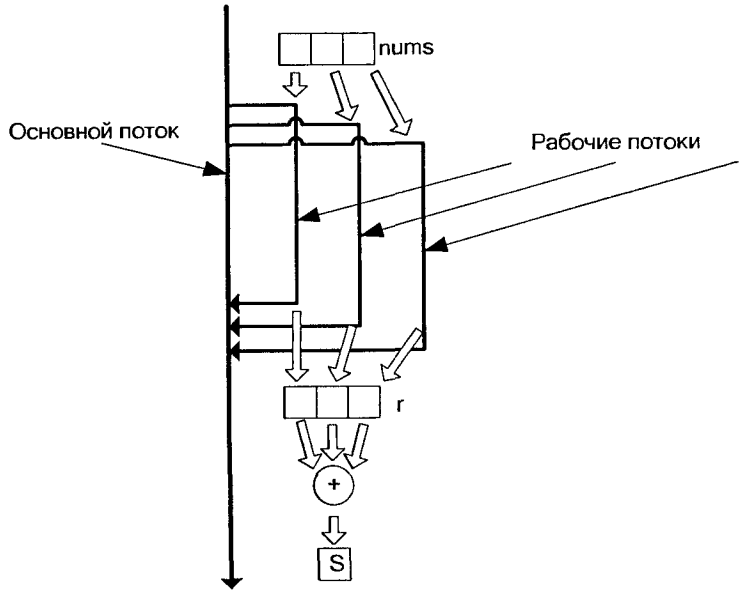


Рис. 16. Схема работы многопоточной программы численного интегрирования

В первых строках программы подключаются необходимые заголовочные файлы, определяется ряд переменных и подынтегральная функция $f(x)=\frac{4}{1+x^2}$ (листинг 26).

```
1:  #include <pthread.h>
2:  #include <string.h>
3:  #include <stdio.h>
4:
```

```
5:  double a = 0.0, b = 1.0, h, *r;
6:  int *nums, numt, n;
7:
8:  double f(double x)
9:  {
10:    return 4 / (1 + x * x);
11:  }
```

Листинг 26. Определение подынтегральной функции

Для получения ускорения необходимо распределить отрезки интегрирования между потоками примерно поровну. Применим ту же схему, что и в случае с MPI-программой, рассмотренной ранее: поток с номером *i* получает для обработки отрезки с номерами *i*, *i+p*, *i+2p*,..., где *p* – общее число потоков. Номер, сохраняемый в переменной `my` передается через аргумент-указатель функции `worker`, выполняемой рабочим потоком (листинг 27).

С помощью цикла в строках 18–19 вычисляется часть суммы площадей прямоугольников, которая сохраняется в элементе `r[my]` массива `r`. Строго говоря, вычисляется не площадь, а только сумма «высот» прямоугольников, которая будет умножена на длину отрезка интегрирования `h` на последнем этапе. Такой подход позволяет сэкономить большую часть операций умножения.

```
12: void* worker(void* addr)
13: {
14:  int my, i;
15:  double s, p;
16:  my = *(int*)addr;
17:   s = 0.0;
18:   for(p = a + my * h; p < b; p += numt * h)
19:     s += 0.5 * (f(p) + f(p + h));
20:   r[my] = s;
21:   return NULL;
22: }
```

Листинг 27. Функция рабочего потока

Главная задача основного потока заключается в том, чтобы запустить рабочие потоки, передав им в качестве аргумента номера по порядку. Решение, которое кажется наиболее очевидным, состоит в том, чтобы передавать потоку адрес итератора цикла, в котором производится запуск потока:

```
for(i = 0; i < numt; i ++)
    pthread_create(threads + i, NULL, worker,
(void*)&i);
```

Это решение не является корректным, так как переменная цикла изменяется в промежутках между запусками потоков. При этом возможна такая ситуация, что при запуске поток с номером *N* прочитает по переданному адресу измененное значение счетчика цикла, равное *N*+1. Поэтому, для каждого потока необходим адрес области памяти, предназначенной для хранения номера только этого потока. Для этого используется массив `numt`, *i*-й элемент которого равен *i*, и его адрес передается потоку в качестве аргумента при создании.

Рассмотрим подробно программу, выполняемую основным потоком (листинг 28). В начале определяется количество рабочих потоков и интервалов разбиения отрезка интегрирования из аргументов командной строки (строки 28–29). В строках 30–32 выделяется память для массива идентификаторов рабочих потоков `threads`, массива их номеров `numt` и массива результатов работы `r`. Далее (строка 33) вычисляется длина стороны одного прямоугольника как отношение длины отрезка интегрирования к числу отрезков разбиения.

Создание потоков выполняется в цикле с помощью функции `pthread_create` (строки 34–41). Перед вызовом заполняется соответствующий элемент массива номеров (строка 35), после чего производится сам вызов (строка 36). В функцию передаются следующие аргументы: адрес *i*-го элемента массива идентификаторов потоков `threads + i`, нулевой указатель в качестве атрибута создания потока, адрес функции потока `worker` и адрес *i*-го элемента массива номеров `numt+i`, содержащего номер создаваемого потока. Код возврата `rc` используется для анализа успешности создания потока (строки 37–40). Проверять код возврата вообще считается хорошим стилем про-

граммирования. В рассматриваемом случае библиотеки POSIX Threads такой подход позволяет избежать многих нетривиальных ошибок в программе.

Цикл в строках 42–47 служит для ожидания завершения рабочих потоков. После выхода из этого цикла все рабочие потоки завершены, и массив r заполнен. Его элементы суммируются и умножаются на длину основания прямоугольника h. Приближенное значение интеграла вычислено и может быть выведено на печать (строка 52).

```
23: main(int arc, char* argv[])
24: {
25:   double S;
26:   pthread_t *threads;
27:   int i, rc;
28:   numt = atoi(argv[1]);
29:   n = atoi(argv[2]);
30:   threads = (pthread_t*)malloc(numt*
        sizeof(pthread_t));
31:   nums = (int*)malloc(numt * sizeof(int));
32:   r = (double*)malloc(numt *
        sizeof(double));
33:   h = (b - a) / n;
34:   for(i = 0; i < numt; i ++) {
35:     nums[i] = i;
36:     rc = pthread_create(threads+
          i,NULL,worker, nums + i);
37:     if(rc != 0) {
38:      fprintf(stderr,"pthread_create:
          error code %d\n", rc);
39:      exit(-1);
40:     }
41:   }
42:   for(i = 0; i < numt; i ++) {
43:     rc = pthread_join(threads[i], NULL);
44:     if(rc != 0) {
45:       fprintf(stderr, "pthread_join: error
            code %d\n", rc);
46:       exit(-1);
47:     }
```



```
48:   }
49:   S = 0;
50:   for(i = 0; i < numt; i ++)
51:     S += r[i];
52:   printf("pi = %lf\n", S * h);
53: }
```

Листинг 28. Основной поток

2.3.2. Вычислительный эксперимент

Для эксперимента был использован суперкомпьютер HP SuperDome фирмы Hewlett-Packard. На данном компьютере установлено 64 процессора HP PA8500 и 64 GB общей памяти.

Вычислительный эксперимент проводился с целью получить характеристики эффективности распараллеливания для многопоточной программы численного интегрирования. Для этого приложение запускалось с различным значением количества создаваемых потоков и измерялось общее время его работы. Количество интервалов разбиения было взято равным 10^8. Результаты вычислительного эксперимента (рис. 17) показывают, что время работы программы существенно снижается при увеличении числа задействованных потоков, что позволяет говорить о высокой эффективности применения потоков в данном примере.

2.4. Синхронизация

Параллельные потоки многопоточного приложения работают независимо. При этом относительные скорости работы потоков зависят от очень многих факторов, влияние которых и практически невозможно заранее предсказать. Если потоки не используют общие данные, то различие в скоростях их работы не приводит к каким-либо проблемам. Ситуация становится принципиально иной, если потоки обрабатывают общие данные.

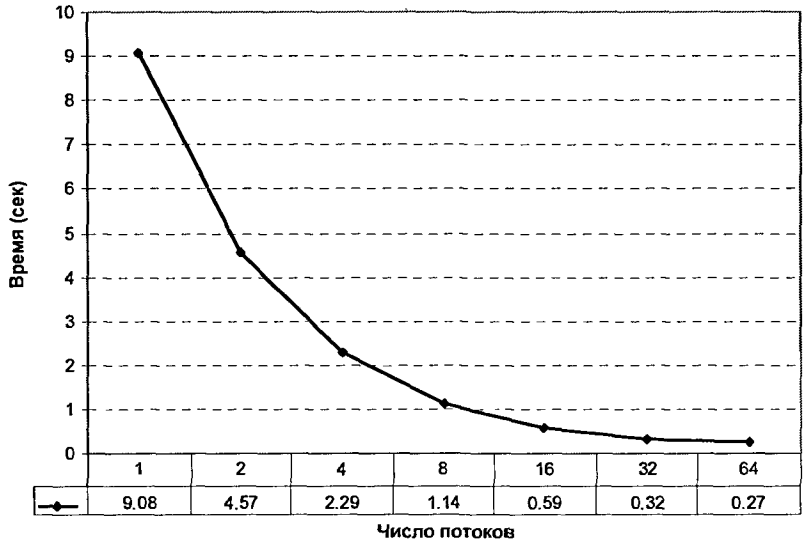


Рис. 17. Зависимость времени работы программы численного интегрирования от числа потоков для суперкомпьютера HP Superdome

В качестве иллюстрации рассмотрим альтернативное решение для проблемы, рассмотренной в предыдущем разделе. Она состояла в том, чтобы каждому рабочему потоку присвоить уникальный номер в диапазоне от 0 до $N-1$, где N — общее число рабочих потоков. В рассмотренном варианте эта задача решалась с помощью присваивания соответствующего номера элементу массива `nums`. Каждый поток при этом работал со своим элементом массива `nums`, и синхронизация не требовалась.

Другое возможное решение основано на общей переменной `counter`, играющей роль счетчика запущенных рабочих потоков. Каждый рабочий поток считывает значение счетчика и увеличивает его значение на единицу (листинг 29). От предыдущего варианта (листинг 27) данный отличается тем, что вместо считывания номера потока из области памяти, на которую указывает аргумент `addr`, вычисление номера производится описанным выше способом (строки 5–6). Перед запуском первого рабочего потока переменной `counter` должно быть присвоено значение 0.


```
1:  void* worker(void* addr)
2:  {
3:    int my, i;
4:    double s, p;
5:    my = counter;
6:    counter ++;
7:    s = 0.0;
8:    for(p = a + my * h; p < b; p += numt * h)
9:      s += 0.5 * (f(p) + f(p + h));
10:   r[my] = s;
11:   return NULL;
12: }
```

Листинг 29. Функция рабочего потока — некорректный (без необходимой синхронизации) вариант с общей переменной-счетчиком

Приведенный вариант может показаться корректным и работоспособным. На самом же деле он содержит ошибку. Рассмотрим возможные варианты выполнения двух потоков (рис. 18). Вариант,

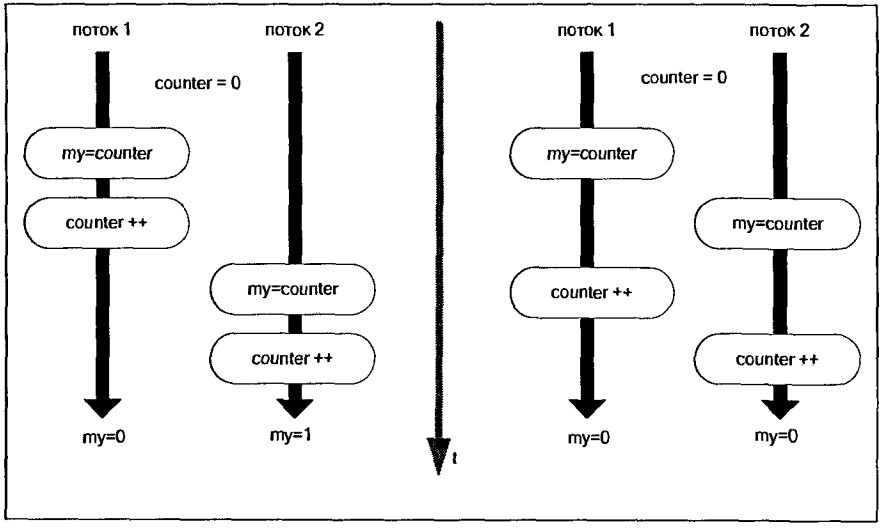


Рис. 18. Различные варианты относительного выполнения двух потоков

который имел в виду разработчик данной реализации, представлен слева. В этом случае первый поток сохраняет значение счетчика в локальной переменной my, а затем увеличивает его значение. После этого второй поток выполняет ту же последовательность действий. Если до начала выполнения перечисленных действий переменная counter имела значение 0, то в результате их выполнения на потоке 1 переменная my имеет значение 0, а на потоке 2 — значение 1.

В варианте справа поток 2 сохраняет значение общей переменной counter в локальной переменной my между операциями присваивания и увеличения счетчика на потоке 1. В результате значение переменной my на обоих потоках становится равным 0. Это означает, что разные потоки получили одинаковые номера, что не является желаемым результатом.

Для предотвращения подобного рода коллизий при обращении к памяти необходим механизм, позволяющий запрещать обращение к переменной counter в тот момент, когда один из потоков выполняет с ней операторы my = counter и counter++. Другими словами, требуется, чтобы последовательность операторов в строках 5 и 6 выполнялась только одним потоком в данным момент времени.

Механизм такой защиты носит название *критических секций*. Под критической секцией понимается последовательность операторов программы, которая в данный момент может выполняться только одним из потоков. Реализация должна предоставлять гарантии, что если один поток находится в процессе выполнения критической секции, то другие потоки не могут начать ее выполнение. Если же несколько потоков одновременно пытаются начать выполнение критической секции, то только один из них получит такую возможность.

Для обеспечения механизма критических секций в библиотеке POSIX Threads предусмотрен механизм мьютексов (mutex) — специальных общих переменных для синхронизации, имеющих тип pthread_mutex_t. Над мьютексами определены три основные операции — захватить мьютекс, освободить мьютекс и проверить мьютекс:

Функция захвата мьютекса:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

mutex - мьютекс;

Функция возвращает 0 в случае успешного завершения или код ошибки в противном случае.

Функция освобождения мьютекса:

```
Int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

mutex - мьютекс;

Освобождает захваченный потоком мьютекс. Возвращает 0 в случае успешного завершения или код ошибки в противном случае.

Операции над мьютексами подчиняются определенным правилам, позволяющим применять их для организации критических секций:

- если несколько потоков пытаются захватить мьютекс одновременно, то только один из потоков сможет это сделать;
- освободить мьютекс может только поток (хозяин мьютекса), выполнивший его захват;
- если мьютекс захвачен одним из потоков, то при попытке захвата этого мьютекса другой поток будет заблокирован до момента освобождения мьютекса.

С помощью мьютексов критическая секция организуется следующим образом: на входе в критическую секцию осуществляется захват мьютекса, а на выходе – освобождение. Рассмотрим в качестве примера функцию worker рабочего потока (листинг 30). Последовательность операторов в строках 6, 7 выделена в критическую секцию: в строке 5 производится захват мьютекса mut, а в строке 8 — его освобождение.

```
1:  void* worker(void* p)
2:  {
3:    int my, i;
4:    double s;
5:    pthread_mutex_lock(&mut);
6:    my = counter;
7:    counter ++;
8:    pthread_mutex_unlock(&mut);
9:    s = 0.0;
```

```
10:    for(i = my; i < n; i += numt)
11:    s += f(i * h + 0.5 * h);
12:    r[my] = s;
13:    return NULL;
14: }
```

Листинг 30. Функция рабочего потока — корректный вариант (добавлена синхронизация)

Введение общего счетчика позволило избавиться от массива `nums`, который использовался для хранения номеров потоков в варианте без синхронизации. Аналогично можно обойтись без массива `r`, который предназначен для хранения результатов работы рабочего потока. Вместо присваивания вычисленного значения массива элементу `r[my]` в новом варианте (листинг 31) выполняется увеличение общей переменной `S`. Доступ к общей переменной также необходимо сделать критической секцией для предотвращения некорректного изменения этой переменной. Для это потребуется еще один мьютекс – `mut1`. Можно было бы воспользоваться мьютексом `mut`, но это привело бы к излишней синхронизации – нет необходимости запрещать одновременно изменять счетчик `counter` и переменную `S`.

```
1:  void* worker(void* p)
2:  {
3:    int my, i;
4:    double s;
5:    pthread_mutex_lock(&mut);
6:    my = counter;
7:    counter ++;
8:    pthread_mutex_unlock(&mut);
9:    s = 0.0;
10:   for(i = my; i < n; i += numt)
11:     s += f(i * h + 0.5 * h);
12:   pthread_mutex_lock(&mut1);
13:   S += s;
14:   pthread_mutex_unlock(&mut1);
15:   return NULL;
16: }
```

Листинг 31. Функция рабочего потока — корректный вариант с двумя критическими секциями

Для того чтобы мьютекс можно было использовать в программе, необходимо его сначала объявить как общую переменную типа `pthread_mutex_t`, затем инициализировать с помощью функции `pthread_mutex_init`, а после использования освободить занятые системные ресурсы с помощью функции `pthread_mutex_destroy`.

Функция инициализации мьютекса:

```
int pthread_mutex_init(pthread_mutex_t *restrict
mutex, const pthread_mutexattr_t *restrict attr);
```

`mutex` – мьютекс;
`attr` – атрибут мьютекс (NULL – атрибут по умолчанию).

Функция возвращает 0 в случае успешного завершения и код ошибки в противном случае.

Функция освобождения ресурсов:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Функция возвращает 0 в случае успешного завершения и код ошибки в противном случае.

В результате код функции `main` изменится (листинг 32) с учетом необходимости инициализации мьютексов (строки 9, 10) и освобождения ресурсов (строки 25, 26).

```
1:  main(int arc, char* argv[])
2:  {
3:    pthread_t *threads;
4:    int i, rc;
5:    numt = atoi(argv[1]);
6:    n = atoi(argv[2]);
7:    threads = (pthread_t*)malloc(numt*
        sizeof(pthread_t));
8:    h = (b - a) / n;
9:    pthread_mutex_init(&mut, NULL);
10:   pthread_mutex_init(&mut1, NULL);
11:   for(i = 0; i < numt; i ++) {
12:     rc = pthread_create(threads+i,NULL,
          worker,NULL);
13:     if(rc != 0) {
14:       fprintf(stderr, "pthread_create (%d):
```

```
                error code %d\n", i, rc);
15:         exit(-1);
16:       }
17:     }
18:     for(i = 0; i < numt; i ++) {
19:       rc = pthread_join(threads[i], NULL);
20:       if(rc != 0) {
21:         fprintf(stderr, "pthread_join:
                    error code %d\n", rc);
22:         exit(-1);
23:       }
24:     }
25:     pthread_mutex_destroy(&mut);
26:     pthread_mutex_destroy(&mut1);
27:     printf("pi = %lf\n", S * h);
28: }
```

Листинг 32. Функция основного потока: добавлена инициализация мьютексов и освобождение системных ресурсов

Мьютексы являются наиболее простым, но в вместе с тем достаточно мощным механизмом синхронизации. Стандарт POSIX Threads предусматривает также ряд других средств синхронизации. В первую очередь это — так называемые семафоры и условные переменные. Изложение этого материала выходит за рамки настоящего пособия. Подробную информацию о них можно найти в стандарте [7].

3. Среда программирования OpenMP

Основным средством программирования систем с распределенной памятью является Message Passing Interface (MPI). Длительное время для разработки программ на системах с общей памятью применялась либо та же библиотека MPI, либо библиотеки для работы с потоками, в частности POSIX Threads. Программирование на MPI имеет тот недостаток, что применение механизма передачи сообщений не позволяет в полной мере воспользоваться возможностями, предоставляемыми общей памятью. Следовательно, система работает неэффективно. Применение потоков хотя и позволяет полностью задействовать ресурсы системы, но является средством достаточно низкого уровня, характерного, скорее, для разработки системных приложений, чем для параллельного программирования.

В силу перечисленных причин задача разработки адекватных высокоуровневых средств программирования систем с общей памятью была и на сегодняшний день остается актуальной. К таким средствам, прежде всего, относятся языки программирования. Наиболее привлекательным представляется подход, основанный на применении интеллектуальных трансляторов, которые генерируют высокоэффективный код для конкретных платформ с общей памятью по последовательной программе на одном из традиционных языков программирования. Такой подход позволил бы автоматически распараллелить большое число существующих последовательных программ, тем самым сэкономив большие трудозатраты, связанные с «ручным» переписыванием этих программ. Попытки создания таких трансляторов неоднократно предпринимались. К сожалению, несмотря на частичный успех, достигнутый в данном направлении, проблема генерации кода удовлетворительного качества для многопроцессорных систем с общей памятью так и остается нерешенной.

Основным препятствием на пути решения задачи автоматического распараллеливания является сложность выявления частей последовательной программы, которые могли бы выполняться

одновременно. Выявление таких частей представляет определенные трудности даже для самого разработчика параллельной программы. Эти трудности многократно возрастают, когда такая задача решается транслятором в автоматическом режиме. Вследствие этого получил распространение подход, основанный на расширении традиционных языков программирования конструкциями, облегчающими автоматическое выявление параллельных операторов и функций в программе. Таких расширений было предложено достаточно много. Необходимость унификации средств разработки параллельных программ привела к созданию в 1997 году единого стандарта, получившего название OpenMP. В настоящее время стандарт развивается и его текущая версия 2.5 [16] доступна на сайте OpenMP [8].

За период, прошедший с момента публикации первой версии стандарта, было разработано много трансляторов, поддерживающих данное расширение. В частности, крупнейшие производители компьютерной техники и программного обеспечения предоставляют такую поддержку в своих продуктах. Подводя итог, можно сказать, что OpenMP является одним из основных высокоуровневых средств программирования систем с общей памятью, доступных на данный момент.

3.1. Общая организация среды OpenMP и модель выполнения

3.1.1. Основные компоненты OpenMP

С точки зрения прикладного программиста в состав среды OpenMP входят три основных компонента: набор директив транслятору, набор функций библиотеки и переменных окружения.

Директивы служат основным средством выражения параллелизма в OpenMP. Директивы представляют собой специальным образом оформленные комментарии (в случае Fortran) или директивы компилятору — «прагмы» (в случае C/C++). Такой подход удобен тем, что эти директивы игнорируются обычным транслятором, не поддерживающим OpenMP. В результате программу можно компилировать и выполнять в последовательном варианте, что существенно облегчает процесс поиска ошибок. При этом

следует принимать во внимание, что в параллельном варианте могут появиться ошибки, которые стали результатом некорректно проведенного распараллеливания.

Функции библиотеки позволяют управлять различными характеристиками в процессе выполнения OpenMP-программы, например числом потоков и т.п. С помощью переменных среды пользователь имеет возможность управлять настройками среды выполнения, например устанавливать характерные параметры распараллеливания циклов. Использование переменных среды позволяет влиять на поведение параллельной программы без перекомпиляции.

3.1.2. Модель выполнения OpenMP-программы

Модель выполнения задает эталонное поведение программы, определяя, каким образом она должна выполняться и какой наблюдаемый эффект будет при этом произведен. Тем самым модель задает рамки, в пределах который конкретная реализация может выбрать наиболее эффективный для данной платформы способ выполнения программы.

В OpenMP принята многопоточная модель (рис. 19). В начале программа всегда состоит из одного потока, называемого главным (master-thread), и выполняется в обычном последовательном режиме. Если в некоторый момент в программе выполняется директива распараллеливания, то к главному потоку добавляется еще несколько, образуя группу параллельно выполняющихся потоков. Потоки выполняют некоторый фрагмент программы, после чего параллельный участок заканчивается, и выполнение вновь продолжает один поток. Таким образом, процесс выполнения OpenMP-программы состоит в чередовании последовательных и параллельных участков.

Параллельные участки могут быть вложенными. Если OpenMP функционирует в режиме с разрешенным вложенным параллелизмом, то каждый из выполняющихся потоков может создавать новые потоки при выполнении нового параллельного участка (рис. 20). Параллельный участок всегда завершается барьерной синхронизацией всех выполняющих его потоков.

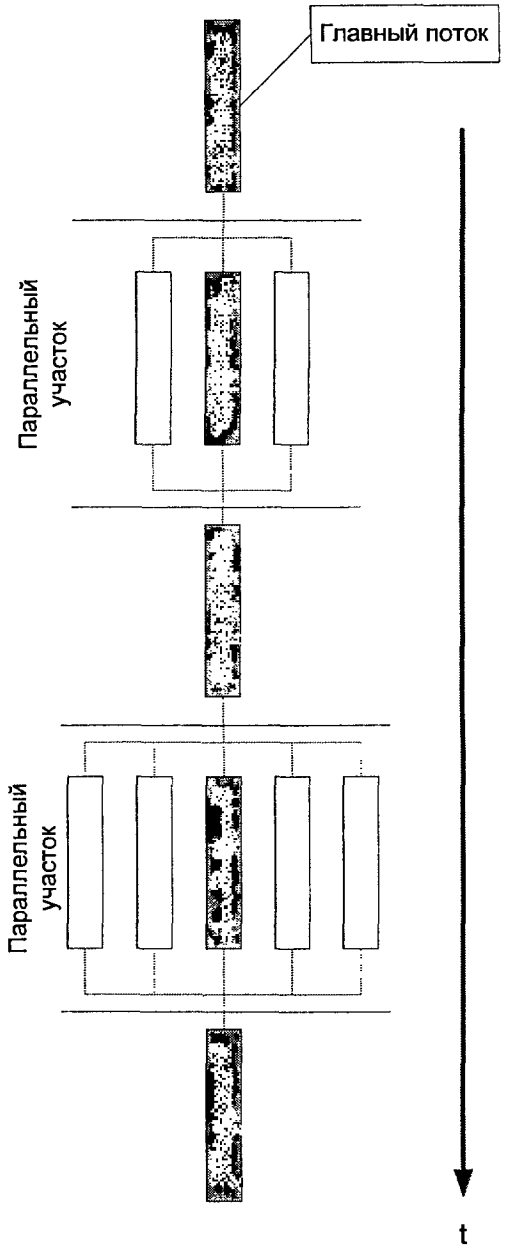


Рис. 19. Модель выполнения OpenMP-программы

3.1.3. Модель памяти OpenMP-программы

Переменные в OpenMP-программе делятся на *общие* (shared) и *индивидуальные* (private). Индивидуальные переменные соответствуют некоторому потоку и могут считываться или записываться только им. Общие переменные доступны для чтения и записи нескольким потокам одной группы. Следует отметить, что если чтение и запись или повторная запись общей переменной производятся без синхронизации, то результирующее значение переменной считается неопределенным.

В отличие от POSIX Threads в OpenMP отсутствует строгая корреляция между классом памяти переменной (статическая, автоматическая и т.п.) и тем, как она рассматривается различными потоками: автоматическая переменная может рассматриваться как общая на некотором параллельном участке, а статическая переменная может быть «приватизирована» каждым потоком.

3.2. Hello World на OpenMP

Рассмотрим простейшую программу на OpenMP (листинг 33). Программа начинается с подключения необходимых библиотек. Для того чтобы получить возможность использовать функции и макросы OpenMP, необходимо подключить заголовочный файл `omp.h` (строка 2). В данной программе содержится всего одна директива OpenMP, расположенная в строке 4. Это – директива `parallel`, обозначающая параллельный участок.

```
1:  #include <stdio.h>
2:  #include <omp.h>
3:  main(){
4:  #pragma omp parallel
5:    printf("Hello World!\n");
6:  }
```

Листинг 33. Самая простая программа на OpenMP

Рассмотрим подробнее синтаксис директивы OpenMP (рис. 21). В языках C/C++ для выражения конструкций OpenMP применяется механизм директив компилятору, начинающихся с ключевого слова `#pragma`. Чтобы директивы OpenMP можно было отличить от

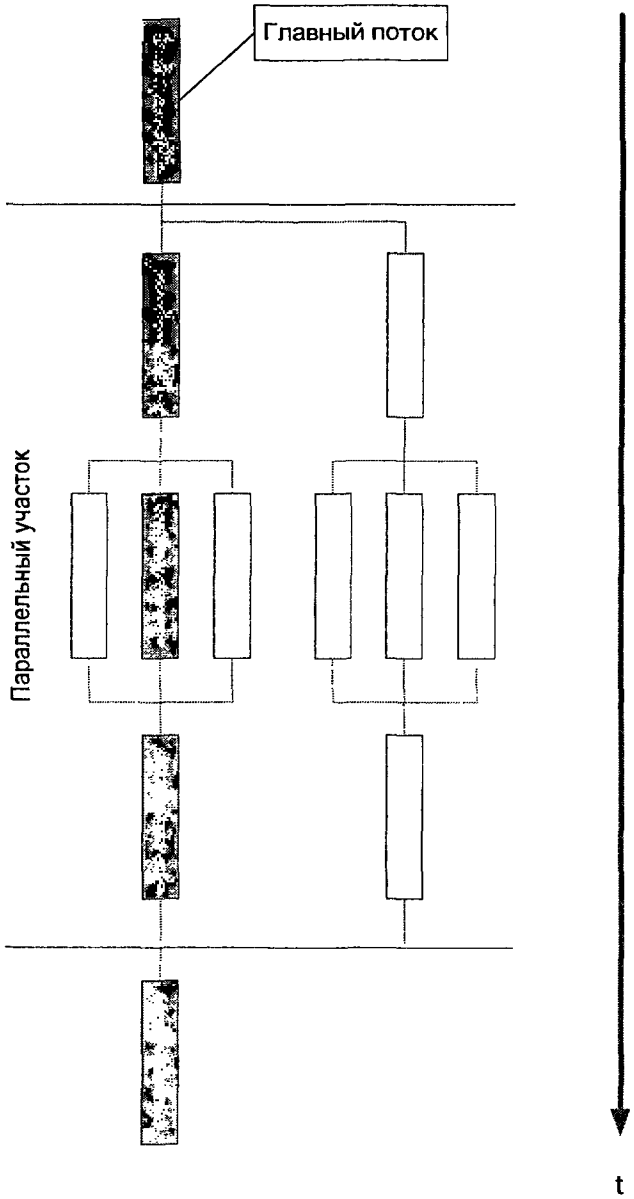


Рис. 20. Выполнение OpenMP-программы со вложенным параллелизмом

прочих директив, после слова `pragma` следует ключевое слово `omp`. Далее следует имя директивы и возможные опции, если они предусмотрены.

Директива `parallel`, как и большинство других директив OpenMP, применяется непосредственно к оператору, следующему за ней. В частности, такой оператор может быть составным, т.е. представлять последовательность операторов, заключенную в фигурные скобки. Такой подход позволяет применять директиву к произвольным последовательным участкам кода программы. Оператор, к которому применяется директива, должен иметь одну точку входа и одну точку выхода из него. Такой оператор принято называть *структурным блоком.*

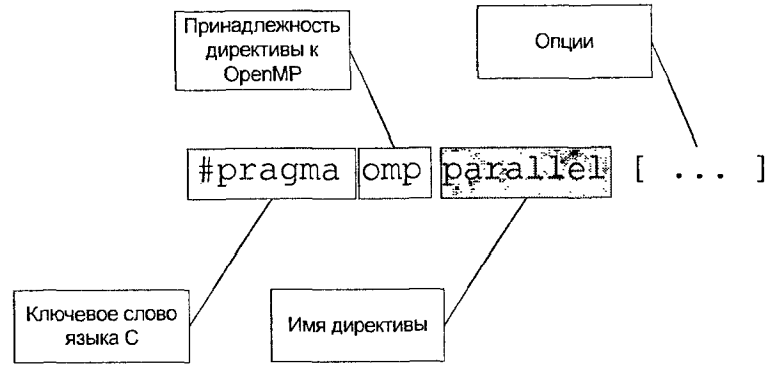


Рис. 21. Синтаксис директивы OpenMP для языков C/C++

В рассматриваемом примере директива применяется к оператору, состоящему из единственного выражения — вызова функции `printf`. Действие директивы `parallel` состоит в создании параллельного участка, в результате чего оператор, к которому применяется это директива, будет выполнен несколькими созданными потоками.

Произведем сборку и запуск этого простого приложения. Для компиляции необходим транслятор языка С, поддерживающий OpenMP. К числу таких трансляторов относится компилятор фирмы «Интел» [6]. Сборка OpenMP программы с помощью транслятора `icc` версии 9.0 выполняется следующим образом:

```
$ icc -openmp -o hi hi.c
```

В результате компиляции будет создан выполняемый файл hi. Флаг openmp включает поддержку OpenMP, без него все директивы OpenMP игнорируются. Собранная программа запускается из командной строки:

```
$ ./hi
```

На экран будет выведено:

```
Hello World!
Hello World!
```

Выведенная информация позволяет заключить, что было создано два потока. Число создаваемых потоков по умолчанию определяется настройками системы. Это число можно изменять с помощью переменных среды как до запуска программы, так и в процессе работы программы.

Переменная среды OMP_NUM_THREADS определяет число потоков, используемых приложением. Если значение этой переменной не определено, то количество создаваемых потоков определяется системным настройками. Присвоим этой переменной значение 5 с помощью следующей команды оболочки bash (для другой оболочки команда может иметь другой синтаксис):

```
$ export OMP_NUM_THREADS=5
```

Запуск программы после этого действия приводит к созданию 5 потоков. В результате на терминале будет напечатано:

```
$./hi
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Таким образом, использование переменной среды OMP_NUM_THREADS позволяет менять количество запускаемых потоков без перекомпиляции приложения.

Альтернативным способом управления числом создаваемых потоков является опция num_threads директивы parallel. Эта опция имеет единственный аргумент, который задает число

потоков, создаваемых для выполнения соответствующего параллельного участка.

В качестве иллюстрации применения данной опции рассмотрим пример (листинг 34). Число запускаемых потоков считывается из аргумента командной строки и сохраняется в переменной n (строка 4). В строке 5 переменная n подставляется в качестве аргумента в опцию num_threads.

```
1:  main(int argc, char* argv[])
2:  {
3:    int n;
4:    n = atoi(argv[1]);
5:  #pragma omp parallel num_threads(n)
6:    printf("Hello World\n");
7:  }
```

Листинг 34. Пример изменения числа выполняемых потоков с помощью опции num_threads

Число потоков, которое целесообразно задавать для выполнения параллельного участка в программе, может определяться исходя из различных соображений. В частности, полезным является учет информации о числе процессоров, доступных для выполнения параллельной программы. Это число возвращается информационной функцией omp_get_num_procs. Две другие информационные функции (omp_get_num_threads и omp_get_thread_num) возвращают число созданных потоков и номер потока, выполняющего вызов. Потоки в группе, создаваемой для выполнения параллельного участка, нумеруются последовательными целыми числами, начиная с нуля.

Получение количества процессоров, доступных для выполнения OpenMP-приложения в момент вызова:

```
int omp_get_num_procs(void)
```

Получение количества потоков в группе:

```
int omp_get_num_threads(void)
```

Получение номера потока в группе:

```
int omp_get_thread_num(void)
```

Следующий пример (листинг 35) демонстрирует использование информационных функций. В строке 4 определяется число доступных для выполнения процессоров. Директива в строке 6 задает параллельный участок, который будет выполняться группой потоков, число которых совпадает с числом доступных процессоров. Функция печати (строка 7) распечатывает текст приветствия, номер каждого потока и число потоков в группе.

```
1:  main()
2:  {
3:    int n;
4:    n = omp_get_num_procs();
5:    printf("%d processors available\n", n);
6:  #pragma omp parallel num_threads(n)
7:    printf("Hello. I'm thread %d from %d.\n",
       omp_get_thread_num(),
       omp_get_num_threads());
8:  }
```

Листинг 35. Пример изменения числа выполняемых потоков с помощью опции num_threads

На двухпроцессорной системе данная программа выведет на печать следующий текст:

```
2 processors available
Hello. I'm thread 0 from 2.
Hello. I'm thread 1 from 2.
```

3.3. Опции для переменных в OpenMP-программе

В предыдущем разделе рассмотрены базовые возможности OpenMP, с помощью которых можно инициировать параллельный участок в программе. Теперь мы рассмотрим средства задания дисциплины работы с данными в OpenMP-программе.

Переменные в OpenMP могут быть либо общими для группы потоков, либо индивидуальными для каждого потока. Принадлежность переменной к тому или иному типу определяется либо с помощью явного указания, либо правилами по умолчанию.

Правила по умолчанию применяются к переменным, которые не являются аргументами опций каких-либо директив. Согласно этим правилам:

- любая переменная, объявленная вне блока параллельного выполнения, будет общей для потоков, выполняющих этот блок;
- общими также являются статические переменные;
- любая автоматическая переменная, объявленная внутри блока параллельного выполнения, будет индивидуальной для потоков, выполняющих этот блок;
- локальные переменные и формальные параметры функций, вызываемых внутри блока параллельного выполнения, также будут индивидуальными.

Проиллюстрируем эти правила на следующем примере:

```
1:  extern int A;
2:  void f(int c)
3:  {
4:    static double z;
5:    int x;
6:  }
7:
8:  main(){
9:    double y;
10:   #pragma omp parallel
11:   {
12:     int a;
13:     f(5);
14:   }
15: }
```

По отношению к параллельному участку следующие переменные являются общими:

A, z, y

остальные переменные:

a, c, x

являются индивидуальными.

Листинг 36. Правила по умолчанию для переменных в OpenMP

Правила по умолчанию можно изменить с помощью специальных опций директивы parallel. Аргументом опции является список идентификаторов переменных, разделенных запятой. В список аргументов могут входить только переменные, принадлежащие области видимости, включающей параллельный участок. Другим словами, они должны быть определены до на-

чала параллельного участка. Рассмотрим наиболее употребительные опции.

Опция `private` определяет список переменных, которые будут индивидуальными для потоков, выполняющих параллельный участок. При этом не задается, каким именно образом производится инициализация значения переменных на потоках. Также неопределенным будет значение переменной на главном потоке после завершения параллельного участка.

Опция `firstprivate` задает способ инициализации индивидуальных переменных: переменные, перечисленные в списке аргументов этой области, получают значение, равное значению переменной на главном потоке в момент входа в параллельный участок. Таким образом, опция `firstprivate` предоставляет всю функциональность опции `private`, добавляя к ней способ инициализации переменных.

Опция `reduction` определяет значение переменных, входящих в список ее аргументов, на главном потоке после завершения параллельного участка как результат выполнения редуктивной операции. На каждом из потоков, выполняющих параллельный участок, переменная получает значение, соответствующее редуктивной операции:

Опция редукции значений переменных на потоках:

```
reduction(operator:list)
```

`list` – список идентификаторов
`operator` – одна из следующих редуктивных операций:

Операция	Значение для инициализации
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

После завершения параллельного участка значение переменной на главном потоке меняется в результате применения к нему и значениям переменных на всех потоках указанной операции. Порядок применения операции определяется реализацией.

Более формально это можно выразить следующим образом. Пусть некоторая переменная a входит в список аргументов опции `reduction` с операцией **op**. Пусть параллельный участок выполнялся n потоками и до него переменная имела значение v. Если в конце выполнения параллельного участка локальные копии переменной a имели значения v_1, ..., v_n, то после параллельного участка переменная a на главном потоке получит значение, равное (v **op** v_1 **op** v_2 **op** ... **op** v_n).

Следующий пример (листинг 37) иллюстрирует применение опции `reduction`. Программа, представленная на листинге, вычисляет произведение целых чисел, переданных через аргументы командной строки. В строках 6, 7 производится разбор аргументов командной строки: сомножители сохраняются в переменных a и b.

Директива `parallel` в строке 9 имеет три опции: `firstprivate`, `reduction` и `num_threads`. Опция `firstprivate(a)` означает, что переменная a будет индивидуальной и инициализируется на всех потоках значением, взятым с главного. Опция `reduction(+ : t)` означает, что после выхода из параллельного участка переменная на главном потоке t будет увеличена на величину суммы значений этой переменной, подсчитанных на каждом из потоков. В данном случае это означает, что переменная t получит значение, равное сумме значения ее локальных копий, так как до входа в участок параллельного выполнения переменная имела значение 0. Последняя опция `num_threads(b)` задает число потоков, равное второму сомножителю b. В теле параллельного участка присутствует единственный оператор присваивания `t = a`, в результате которого индивидуальные переменные t получают одинаковые значения, равные первому сомножителю. Общее число потоков, таким образом, равняется величине второго сомножителя, а на каждом из них переменная t имеет значение, равное первому сомножителю. В результате редукции, после выхода из параллельного участка

переменная t получит значение, равное произведению a и b. В строке 14 это значение выводится на печать.

```
1:  #include <stdio.h>
2:  #include <omp.h>
3:  main(int argc, char* argv[])
4:  {
5:    int a, b, t;
6:    a = atoi(argv[1]);
7:    b = atoi(argv[2]);
8:    t = 0;
9:    #pragma omp parallel firstprivate(a)
10:     reduction(+ : t) num_threads(b)
11:   {
12:     t = a;
13:   }
14:   printf("a x b = %d\n", t);
15: }
```

Листинг 37. Применение опции `reduction`

3.4. Синхронизация в OpenMP

Как и в любой среде многопоточного программирования, в OpenMP важную роль играет синхронизация. Синхронизация необходима, если различные потоки работают с общими данными. Если чтение и запись или повторная запись общей переменной производятся без синхронизации, то результирующее значение переменной считается неопределенным.

Самым простым способом защитить данные от возможных проблем, связанных с одновременным доступом, является директива `atomic`. Эта директива применяется к оператору-выражению одного из следующих типов:

```
x op= expr;
x ++;
++ x;
x --;
-- x;
```


где x — переменная, `expr` — выражение, не ссылающееся на x, `op` — бинарная операция. Данная директива обеспечивает атомарность соответствующей операции: в момент выполнения этой операции одним из потоков другие потоки не имеют доступа к переменной x.

Другим базовым механизмом синхронизации является механизм *критических секций*. Критическая секция в коде программы выделяется с помощью директивы `critical`, которая имеет следующий синтаксис:

```
#pragma omp critical [name]
```

Опция `name` является необязательной; если она отсутствует, то считается, что директива имеет зарезервированное не специфицированное имя. Таким образом, каждой критической секции ставится в соответствие некоторое имя. Синхронизация обеспечивается следующим образом: критические секции с одинаковыми именами не могут выполняться одновременно.

В некоторых случаях требуется ограничить набор потоков, выполняющих некоторый фрагмент кода. Это достигается при помощи директивы `master`, имеющей синтаксис:

```
#pragma omp master
```

Оператор, к которому применяется данная директива, выполняется только главным потоком.

В OpenMP предусмотрен также ряд других директив синхронизации, о которых более подробная информация содержится в [16].

3.5. Распределение работы между параллельными потоками

Директива `parallel` позволяет инициировать параллельное выполнение участка программы группой потоков. Используя информацию о номере потока, которая доступна через функцию `omp_get_thread_num`, и механизмы синхронизации, можно разрабатывать достаточно сложные параллельные программы по технологии, аналогичной применяемой в POSIX Threads. При этом процесс разработки остается достаточно сложным и низкоуровневым.

В отличие от библиотек для многопоточного программирования, OpenMP предоставляет механизмы автоматизации распределения работы по потокам, которые далее подробно рассматриваются в этом разделе.

3.5.1. Последовательный участок внутри параллельного — директива `single`

Самой простой директивой распределения работы в OpenMP является директива `single`. Эта директива выделяет оператор, выполняемый только одним потоком из группы (не обязательно главным). Таким образом, внутри параллельного участка появляется фрагмент кода, выполняемый в последовательном режиме.

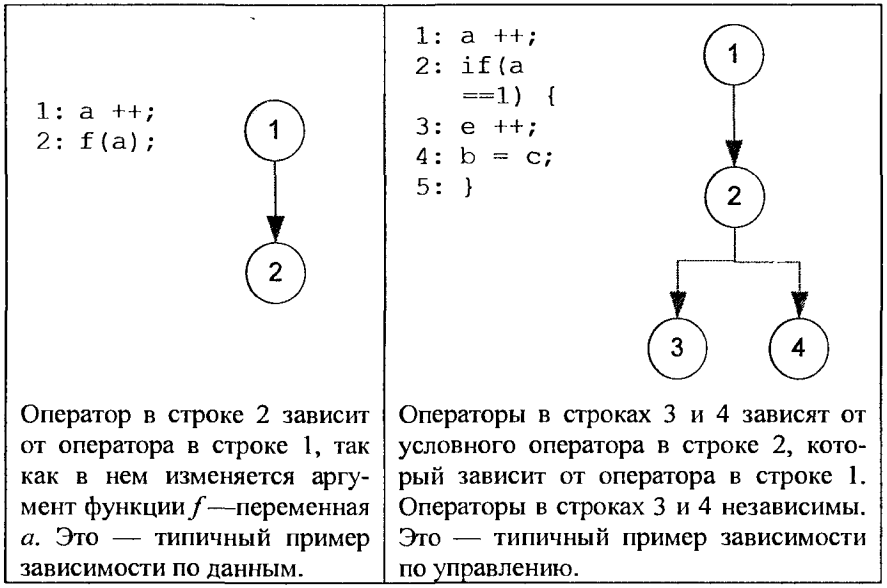
3.5.2. Информационные зависимости в программе

Одной из основных причин, которая не позволяет провести эффективное распараллеливание, является наличие информационных зависимостей между операторами программы. Информационные зависимости имеют место в случае, когда один из операторов использует результаты работы другого оператора. Например, считывает значения переменной, измененной другим оператором.

Будем говорить, что два оператора имеют *зависимость по данным*, если 1) оба обращаются к общему участку памяти, и 2) хотя бы одно из этих обращений является записью. Помимо зависимости по данным, существует также *зависимость по управлению* — когда один из операторов является оператором, влияющим на поток управления, а выполнение второго зависит от выполнения первого.

Программе можно сопоставить ориентированный граф, если поставить в соответствие операторам его вершины, а дугами обозначить информационные зависимости. Примеры фрагментов кода программ и соответствующие им зависимости приводятся на рис. 22.

Информационные зависимости необходимо учитывать при распараллеливании. Очевидно следующее утверждение: если между операторами нет информационной зависимости, то их можно выполнять одновременно. В противном случае может быть нарушена семантика программы.



Оператор в строке 2 зависит от оператора в строке 1, так как в нем изменяется аргумент функции f — переменная a. Это — типичный пример зависимости по данным.

Операторы в строках 3 и 4 зависят от условного оператора в строке 2, который зависит от оператора в строке 1. Операторы в строках 3 и 4 независимы. Это — типичный пример зависимости по управлению.

Рис. 22. Примеры информационных зависимостей

Информационные зависимости приобретают более сложный вид, если программа содержит вызовы функций, циклические участки, условные и безусловные переходы. Понятие зависимости и графа информационных зависимостей можно ввести достаточно строго и в этом случае [13]. Мы ограничимся лишь описанием на идейном уровне.

Важным частным случаем информационной зависимости является зависимость между итерациями цикла. Две итерации цикла зависят друг от друга, если 1) они обращаются к общим данным и 2) хотя бы одно из этих обращений является записью. Пример цикла с зависимостями между итерациями приведен на рис. 23. Для пяти итераций построен граф информационных зависимостей: итерация `j` зависит от итераций `j - 1` и `j - 2`.

Итерации цикла, между которыми существует информационная зависимость, необходимо выполнять в порядке, определяемом этой зависимостью. В частности, зависимые итерации не могут выполняться одновременно.

```
for(j = 0; j < n;
j ++)
    a[j] = a[j -
1] + a[j - 2];
```

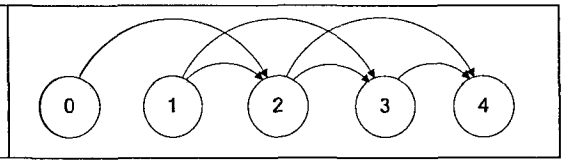


Рис. 23. Цикл с информационными зависимостями между итерациями

Стандарт OpenMP не требует от компилятора проверки зависимости по данным. Поэтому в большинстве существующих реализаций такая проверка отсутствует. Это означает, что ответственность за то, что при распределении работы между параллельными потоками зависимость по данным не нарушается, лежит на разработчике.

3.5.3. Разбиение на независимые блоки — директива sections

В ряде случаев в программе удается выделить достаточно крупные фрагменты, которые можно выполнять параллельно. Директива `sections` позволяет организовать такое выполнение.

Директива применяется к составному оператору, т.е. к последовательности операторов, заключенных в фигурные скобки. Тело составного оператора должно быть представлено в виде последовательности операторов (секций), каждому из которых, за исключением, быть может, первого, предшествует директива `section`, имеющая следующий синтаксис:

```
#pragma omp sections
{
#pragma omp section
        statement1
#pragma omp section
        statement2
...
}
```

Структурный блок, к которому применяется директива `sections`, должен выполняться как часть параллельного участка.

При этом каждая из секций этого блока выполняется только один раз одним из потоков, входящих в группу.

При применении данной директивы необходимо убедиться, что разные секции не содержат операторов, между которыми существует информационная зависимость. Если этого не сделать, в результате распараллеливания программа может стать некорректной.

Выполнение директивы `sections` должно происходить внутри параллельного участка. Предусмотрена сокращенная директива `parallel sections`, позволяющая комбинировать инициализацию параллельного участка и распределение работы по секциям.

3.5.4. Распараллеливание циклов

Большая часть времени работы приложений приходится на циклические участки. Поэтому ускорению работы таких участков с помощью распараллеливания уделяется большое внимание.

Значительная часть работ по автоматическому распараллеливанию программ посвящена именно циклам [13]. Несмотря на значительный прогресс, достигнутый в области автоматического распараллеливания циклов, эффективность параллельного кода, генерируемого современными промышленными компиляторами, невысока. Это связано с тем, что задача автоматического выявления и анализа зависимостей по данным между итерациями цикла в общем случае является чрезвычайно сложной. Кроме того, распараллеливанию целесообразно подвергать только циклы, выполнение которых вносит существенный вклад в работу программы. Автоматическое выявление таких циклов также является трудновыполнимой задачей.

Для преодоления этой проблемы в OpenMP принят следующий подход: разработчик параллельного приложения самостоятельно выбирает циклические участки, которые надо распараллеливать. При необходимости имеющиеся циклы модифицируются с целью устранения информационных зависимостей. Выбранные циклы помечаются специальной директивой, в которой указываются некоторые параметры распределения работы. В процессе трансляции компилятор OpenMP генерирует код, который распределяет итерации цикла по потокам. При этом предполагается, что разработчик убедился в отсутствии информационных зависимостей между итерациями.

Для обозначения распараллеливаемых циклов применяется директива `for`. В программе эта директива предшествует распараллеливаемому циклу типа `for`. При этом цикл должен удовлетворять дополнительным условиям, выполнение которых требуется для того, чтобы компилятор смог эффективно произвести анализ и сформировать качественный параллельный код:

```
#pragma omp for [опции ... ]
for (init-expr; var rel b; incr-expr)
  тело_цикла
```

При этом:

`initexpr` – либо `var = lb`,
либо `integer-type var = lb`

`increxpr` – одно из следующих выражений:

```
++var
var++
--var
var--
var += incr
var -= incr
var = var + incr
var = incr + var
var = var - incr
```

`var` – переменная знакового целого типа

`rel` – одна из следующих операций сравнения:

```
<
<=
>
>=
```

Важно отметить, что переменная-итератор цикла автоматически становится индивидуальной для всех потоков. Стандарт OpenMP не допускает изменение ее значения в теле цикла.

Директива `for` должна выполняться внутри параллельного участка. Предусмотрена также директива `parallel for`, предоставляющая возможность совмещения функциональности директив `parallel` и `for` в одной директиве. В результате выполнения этой директивы создается параллельный участок выполнения и итерации цикла распределяются между выполняющими его потоками.

В качестве примера применения директивы `for` рассмотрим программу вычисления поэлементной суммы двух векторов. Сначала рассмотрим последовательную функцию вычисления суммы двух векторов (листинг 38). Функция принимает следующие входные параметры:

`n` – размерность вектора;

`a, b` – суммируемые вектора;

`c` – результат.

Функция в цикле (строки 4–5) вычисляет поэлементную сумму двух векторов.

```
1:  void vsum(int n, double* a, double* b,
      double* c)
2:  {
3:  int i;
4:  for(i = 0; i < n; i ++)
5:    c[i] = a[i] + b[i];
6:  }
```

Листинг 38. Вычисление поэлементной суммы двух векторов — последовательный вариант

Итерации цикла в строках 4–5 не имеют информационных зависимостей, так как на каждой следующей итерации обрабатываются новые элементы массивов `a, b, c`. Поэтому к нему можно применить директиву `for` (листинг 39).

```
1:  void vsump(int n, double* a, double* b,
      double* c)
2:  {
3:    int i;
4:    #pragma omp for
5:    for(i = 0; i < n; i ++)
6:      c[i] = a[i] + b[i];
7:  }
```

Листинг 39. Вычисление поэлементной суммы двух векторов — параллельный вариант

Рассмотрим теперь функцию main, которая вызывает обе функции, измеряет и сравнивает времена их работы (листинг 40). Размерность складываемых векторов и число итераций цикла берется из первого и второго аргументов командной строки (строки 5, 6). В строках 7–9 производится выделение памяти под вектора. Далее в строках 13–14 заданное число раз выполняется последовательный вариант функции вычисления суммы векторов, в строках 22–23 — параллельный. Измеренные времена выводятся на печать.

Параллельный вариант функции суммирования векторов вызывается в цикле, который работает внутри параллельного участка в строках 19–24.

```
1:  main(int argc, char* argv[])
2:  {
3:    int n, iters, t;
4:    double* a, *b, *c;
5:    n = atoi(argv[1]);
6:    iters = atoi(argv[2]);
7:    a = (double*)malloc(n * sizeof(double));
8:    b = (double*)malloc(n * sizeof(double));
9:    c = (double*)malloc(n * sizeof(double));
10:   t = time(NULL);
11:   {
12:     int i;
13:     for(i = 0; i < iters; i ++)
14:       vsum(n, a, b, c);
15:   }
16:   t = time(NULL) - t;
17:   printf("Sequential: %d iterations consumed
        %d seconds\n", iters, t);
18:   t = time(NULL);
19:   #pragma omp parallel firstprivate(n)
20:   {
21:   int i;
22:   for(i = 0; i < iters; i ++)
23:     vsump(n, a, b, c);
24:   }
25:   t = time(NULL) - t;
26:   printf("Parallel: %d iterations consumed
```



```
        %d seconds\n", iters, t);
27: }
```

Листинг 40. Вычисление поэлементной суммы двух векторов — функция main

3.5.5. Устранение зависимости по данным с помощью редукции

Функция суммирования векторов удобна для распараллеливания, так как итерации цикла не содержат зависимостей по данным. Ситуация обстоит по-другому в случае вычисления скалярного произведения векторов (листинг 41). Итерации цикла for (строки 6–7) содержат зависимости по данным: каждая итерация считывает и изменяет значения общей переменной s. В этом частном случае зависимость можно устранить: итоговое значение s не зависит от порядка, в котором выполняются итерации цикла. Этот факт создает предпосылки для распараллеливания.

```
1:  double dotprod(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    for(i = 0; i < n; i ++)
7:      s += a[i] * b[i];
8:    return s;
9:  }
```

Листинг 41. Последовательный вариант функции для скалярного произведения векторов

Применение к циклу директивы for не является корректным, так как доступ к общей переменной s производится разными потоками без синхронизации. Вариант с синхронизацией представлен на листинге 42. К оператору сложного присваивания в строке 10 применена директива atomic. Это означает, что при выполнении данного оператора различными потоками не происходит конфликтов из-за обращения к общей переменной s.

```
1:  double dotprods(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    #pragma omp parallel for
7:    for(i = 0; i < n; i ++)
8:      #pragma omp atomic
9:      s += a[i] * b[i];
10:   return s;
11: }
```

Листинг 42. Параллельный вариант функции вычисления скалярного произведения векторов — вариант с синхронизацией

Применение синхронизации в рассматриваемом примере позволяет сохранить корректность, но при этом существенно страдает эффективность. Действительно, различные потоки вынуждены выполнять итерации цикла по очереди. Вследствие этого реального параллельного выполнения не происходит и вместо ускорения наблюдается замедление работы программы.

Альтернативный вариант основан на следующем простом наблюдении: каждый поток может вычислить часть суммы независимо, а потом следует просто просуммировать значения на разных потоках. Операции суммирования элементов массива и другие редукции достаточно часто встречаются в практике разработки программ. Поэтому в OpenMP предусмотрена специальная опция `reduction` для поддержки таких операций. Эта опция предусмотрена для директив `parallel`, `for` и `sections` и имеет два аргумента — *редуктивную операцию* и *редуктивную переменную* (см. стр. 131). Напомним, что редуктивная переменная вычисляется независимо на разных потоках, после чего значения, полученные на разных потоках, комбинируются с помощью редуктивной операции.

Если применить опцию `reduction`, функция вычисления скалярного произведения примет вид, представленный на листинге 43. В этом примере используется комбинированная директива `parallel for` с опцией `reduction(+:s)`. В результате каждый поток вычислит часть скалярного произведения. Окончательное значение будет найдено как сумма значений, вычисленных на всех потоках.


```
1:  double dotprodp(int n, double* a, double* b)
2:  {
3:    int i;
4:    double s;
5:    s = 0;
6:    #pragma omp parallel for reduction(+:s)
7:    for(i = 0; i < n; i ++)
8:      s += a[i] * b[i];
9:    return s;
10: }
```

Листинг 43. Параллельный вариант функции вычисления скалярного произведения векторов с применением опции `reduction`

Функции, представленные на листингах 41, 42, 43, вычисляют одно и то же значение, но работают с различной производительностью. Параллельный вариант с синхронизацией работает неприемлемо долго: время работы превышает время последовательного варианта на два порядка. Сравним эффективность их работы при различных размерностях векторов последовательного варианта и параллельного варианта с редукцией (табл. 5). Вычислительный эксперимент проводился на двухпроцессорной системе следующей конфигурации: процессор: 2 x Intel® Itanium-2® 1.6 ГГц, оперативная память: 2ГБ.

При средних и высоких значениях размерности векторов параллельный вариант лучше последовательного в среднем в два раза, что соответствует ожидаемым показателям производительности для двухпроцессорной системы. Для относительно коротких векторов время работы параллельного варианта превосходит время работы последовательного. Это связано с тем, что накладные расходы на организацию параллельного выполнения цикла превосходят эффект от распараллеливания. Данное наблюдение показывает, что для параллельного выполнения следует выбирать циклы с большим количеством итераций.

Таблица 5. Время работы последовательного и параллельного (с редукцией) вариантов функции вычисления скалярного произведения при различной длине векторов

Время работы варианта, сек	Число итераций				
	10000000	1000000	100000	10000	1000
	Длина вектора				
	1000	10000	100000	1000000	10000000
последовательного	13	13	12	12	13
параллельного	36	9	7	7	6

3.5.6. Стратегия распределения итераций цикла по потокам

Итерации цикла, к которому применена директива `for`, могут по-разному распределяться по потокам. Повлиять на стратегию распределения, применяемую по умолчанию, позволяет опция `schedule` директивы `for`. Данная опция предусматривает два аргумента. Первый определяет способ распределения итераций. Второй необязательный аргумент задает число итераций в порции, которая служит единицей распределения нагрузки. Предусмотрены четыре различных варианта соответствующие различным комбинациям аргументам `schedule` (табл. 6). Если опция не указана, то применяется стратегия, установленная по умолчанию.

В качестве примера применения опции `schedule` можно привести знакомый нам цикл вычисления скалярного произведения векторов, для которого зададим статическое распределение порциями по 8 итераций:

```
#pragma omp parallel for reduction(+:s) \
                         schedule(static,8)
  for(i = 0; i < n; i ++)
      s += a[i] * b[i];
```

В результате итерации цикла будут распределены циклически между потоками блоками по 8 итераций. В этом разделе мы рассмотрели только базовые конструкции OpenMP и привели примеры их применения. Более полная и подробная информация по этому пакету содержится в [6, 8].

Таблица 6. Опции, определяющие стратегию распределения итераций цикла по потокам

Опция	Описание
`schedule(static, chunk_size)`	Итерации разделяются на порции размера `chunk_size`, которые статическим образом распределяются по потокам в группе циклическим образом. Последняя порция итераций может содержать менее чем `chunk_size` итераций Если значение `chunk_size` не задано, то все множество итераций делится на приблизительно одинаковые порции так, что каждый поток получает не более одной порции
`schedule(dynamic, chunk_size)`	Итерации распределяются по потокам по запросу: поток, закончивший выполнение, получает очередную порцию. Этот процесс продолжает до тех пор, пока все итерации цикла не будут выполнены Каждая порция, за исключением, быть может, последней, содержит `chunk_size` итераций. По умолчанию `chunk_size` считается равным 1
`schedule(guided, chunk_size)`	Итерации распределяются по потокам по запросу: поток, закончивший выполнение, получает очередную порцию. Этот процесс продолжает до тех пор, пока все итерации цикла не будут выполнены Число итераций в распределяемой порции задается как отношение количества нераспределенных итераций к количеству потоков, если это количество не меньше чем `chunk_size`. В противном случае, если порция не является последней, ее размер полагается равным `chunk_size`. По умолчанию `chunk_size` считается равным 1
`schedule(runtime)`	Стратегия балансировки определяется во время выполнения программы. Для этого используется значение переменной окружения `OMP_SCHEDULE` или настройка по умолчанию если значение этой переменной не определено

Заключение

Основная цель, которую авторы книги ставили перед собой, состояла в том, чтобы развить у читателя практические навыки разработки параллельных приложений. Этим объясняется ограничение списка рассматриваемых тем только наиболее употребительными средствами разработки параллельных программ. Приложения к учебному пособию содержат справочную информацию, необходимую для того, чтобы самостоятельно разрабатывать параллельные приложения. Дополнительные источники информации перечислены в списке литературы.

Следует отметить, что за период существования параллельных вычислительных систем было создано немало библиотек и языков, облегчающих создание параллельных программ. Эти разработки внесли существенный вклад в развитие методов параллельных вычислений и представляют значительный научный интерес. В то же время, подавляющее большинство из них не получили широкого признания в качестве средств разработки параллельных программ. Поэтому мы сознательно не уделили внимание таким инструментам. Тем читателям, которых интересуют подобные разработки, можно рекомендовать в качестве отправной точки монографию [3].

За рамками настоящего пособия остались также философские аспекты параллельного программирования и математические модели процесса вычислений. Информацию по этому вопросу можно найти в монографиях [1, 12].

ЛИТЕРАТУРА

1. *Корнеев В.В.* Вычислительные системы. Гелиос АРВ, 2004 г.
2. *Шнитман В.З.* Современные высокопроизводительные компьютеры. http://www.citforum.ru, 1996 г.
3. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002 г.
4. *Барский А.Б.* Параллельные информационные технологии. М. БИНОМ, 2007 г.
5. *M.S. Otto, S. Huss-Liderman, D. Walker, J. Dongarra.* MPI: The Complete Reference. MIT Press, 1996.
6. MPI-Forum. http://www.mpi-forum.org.
7. *Д. Каханер, К. Моулер, С. Нэш.* Численные методы и программное обеспечение. М.: Мир, 1997.
8. *Г.Э. Эндрюс.* Основы многопоточного, параллельного и распределенного программирования. Издательский дом «Вильямс», 2003.
9. *А.А. Самарский, А.В. Гулин.* Численные методы. М.: Наука, 1989.
10. *A. Grama, A. Gupta, G. Karypis, V. Kumar.* Introduction to Parallel Computing, Addison Wesley, 2003.
11. *Ю. Вахалия.* UNIX изнутри. – СПб.:Питер, 2003.
12. Открытый стандарт для Unix-систем. http://www.unix.org/
13. OpenMP Application Program Interface. Version 2.5, 2005.
14. Официальный сайт OpenMP: www.openmp.org.
15. Сайт фирмы Интел: www.intel.com.
16. *K. Kennedy, John R. Allen.* Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., 2001.

Приложение 1
Справочная информация по MPI

Источник: http://www.mpi-forum.org. Права на исходный текст принадлежат © 1993, 1994, 1995 University of Tennessee, Knoxville, Tennessee.

Источник: http://www-unix.mcs.anl.gov/mpi/www/ Права на исходный текст принадлежат © 1993, 1994, 1995

Argonne National Laboratory Group

W. Gropp: (630) 252-4318; FAX: (630) 252-5986; e-mail: gropp@mcs.anl.gov

E. Lusk: (630) 252-7852; FAX: (630) 252-5986; e-mail: lusk@mcs.anl.gov

Mathematics and Computer Science Division Argonne National Laboratory, Argonne IL 60439

Mississippi State Group

N. Doss: (601) 325-2565; FAX: (601) 325-7692; e-mail: doss@erc.msstate.edu

A. Skjellum:(601) 325-8435; FAX: (601) 325-8997; e-mail: tony@erc.msstate.edu

Mississippi State University, Computer Science Department & NSF Engineering Research Center for Computational Field Simulation P.O. Box 6176, Mississippi State MS 39762

Справочная информация, представленная в дальнейших разделах приложения, прежде всего, описывает функциональность наиболее распространенной реализации MPI — MPICH (http://www-unix.mcs.anl.gov/mpi/mpich1/). Следует отметить, что при этом основная часть материала универсальна и применима к любой реализации стандарта MPI-1.

Коды ошибок

Все MPI-функции (кроме `MPI_Wtime` и `MPI_Wtick`) возвращают код ошибки. Перед возвратом кода ошибки вызывается текущий обработчик ошибок MPI. По умолчанию этот обработчик ошибок прерывает выполнение MPI-программы. Обработчик ошибок может быть изменен с помощью `MPI_Errhandler_ set`; в этом случае предопределенный обработчик ошибок `MPI_ERROR_ RETURN` можно использовать для определения причины появления ошибки. Если вы будете использовать свой обработчик ошибок, то следует помнить, что MPI не гарантирует возможности дальнейшего выполнения программы после обнаружения ошибки.

MPI_SUCCESS — нет ошибок; MPI-функция завершена успешно.

MPI_ERR_COMM — неверный коммуникатор. Частая причина — использование `NULL` коммуникатора в вызове функции.

MPI_ERR_COUNT — неверный аргумент `count`. Он должен быть неотрицательным; нулевое значение также допустимо.

MPI_ERR_TYPE — неверный тип данных у аргумента.

MPI_ERR_TAG — неверный тэг. Тэги должны быть неотрицательными; в функциях `MPI_Recv`, `MPI_Irecv`, `MPI_ Sendrecv` и т.п. можно использовать и `MPI_ANY_TAG`.

MPI_ERR_RANK — неверный ранг источника или приемника сообщений. Ранг может быть задан в диапазоне от 0 до числа процессов в коммуникаторе минус 1; в функциях `MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv` и т.п. можно использовать и `MPI_ANY_SOURCE`.

MPI_ERR_INTERN — ошибка означает, что некоторым компонентам MPICH не выделяется память. Внутренняя фатальная ошибка. При ее появлении необходимо отправить отчет (a bug report) по адресу: `mpi-bugs@mcs. anl.gov`.

MPI_ERR_REQUEST — неверный дескриптор — или 0, или, в случае функций `MPI_Start` и `MPI_Startall`, недопустимый (закрытый) дескриптор.

MPI_ERR_IN_STATUS — этот код ошибки возвращается только функциями `MPI_ Testall`, `MPI_Testany`, `MPI_ Testsome`, `MPI_Waitall`, `MPI_Waitany` и `MPI_Waitsome`.

Поле MPI_ERROR в статусе завершения содержит код ошибки или MPI_SUCCESS (завершено) либо MPI_ERR_PENDING, в случае если запрос не завершен.

MPI_ERR_PENDING — задержанная операция (не ошибка). Этот код означает, что операция не завершена или обнаружена случайная ошибка.

MPI_ERR_TRUNCATE — сообщение урезано в приемнике. Размер буфера мал для принимаемого сообщения. При возникновении этой ошибки в реализации MPICH-приложение может продолжить выполнение.

MPI_ERR_BUFFER — неверный указатель буфера. Обычно null buffer там, где это недопустимо.

MPI_ERR_ROOT — неверный корневой процесс. Он должен быть задан как ранг процесса в коммуникаторе. Ранг может быть задан в диапазоне от 0 до $p - 1$, где p — число процессов в коммуникаторе минус.

MPI_ERR_ARG — неверные аргументы функции. Возвращается, если некоторые ошибочные аргументы не могут быть классифицированы более точно.

MPI_ERR_UNKNOWN — неизвестная (нераспознаваемая) ошибка. При ее появлении необходимо отправить отчет (a bug report) по адресу: mpi-bugs@mcs.anl.gov.

MPI_ERR_OP — неверный дескриптор операции. MPI операция должна быть или предопределеной (например MPI_SUM), или созданной при помощи MPI_Op_create.

MPI_ERR_GROUP — в функцию передана нулевая группа.

Функции точечных обменов

MPI_Send — отправка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Send( void *buf, int count,
              MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Примечание. Эта функция может (но не обязана) блокировать процесс-отправитель до момента вызова парной операции приема.

MPI_Recv — прием сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Recv( void *buf, int count,
              MPI_Datatype datatype, int
              source, int tag, MPI_Comm comm,
              MPI_Status *status )
```

Выходные параметры:

buf — указатель на начало буфера, в который будут сохранены полученные данные (choice);

status — указатель на структуру MPI_Status.

Входные параметры:

count — максимальное число элементов в пересылаемом буфере;

datatype — MPI-тип данных каждого пересылаемого элемента буфера;

source — номер процесса, который отправляет данные;

tag — тэг сообщения;

comm — коммуникатор.

Примечание. Аргумент count определяет максимальную длину сообщения; текущее количество полученных элементов можно определить при помощи MPI_Get_count.

MPI_Get_count — возвращение числа полученных элементов указанного типа.

Синтаксис:

```
#include "mpi.h"
int MPI_Get_count(MPI_Status *status,
        MPI_Datatype datatype, int *count )
```

Входные параметры:

status — статус операции приема сообщения;

datatype — MPI-тип данных принимаемых элементов.

Выходной параметр:

count — число полученных элементов.

MPI_Bsend — буферизированная пересылка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Bsend(void *buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Примечание. Эта функция позволяет пользователю быть уверенным в том, что сообщение будет буферизовано (буфер *должен* быть выделен с помощью функции MPI_Buffer_attach).

MPI_Ssend — синхронная пересылка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Ssend( void *buf, int count,
        MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Примечание. Эта функция блокирует процесс-отправитель до момента вызова парной операции приема.

MPI_Rsend — пересылка по готовности.

Синтаксис:

```
#include "mpi.h"
int MPI_Rsend( void *buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

MPI_Buffer_attach — создание пользовательского буфера для пересылки.

Синтаксис:

```
#include "mpi.h"
int MPI_Buffer_attach( void *buffer, int size )
```

Входные параметры:

buffer — начальный адрес буфера;

size — размер буфера в байтах.

Примечание. Размер буфера должен быть равен сумме размеров всех сообщений выполняемых функций Bsend, плюс служебная информация. Для вычисления размера буфера можно использовать MPI_Pack_size. Например, в коде:

```
    MPI_Buffer_attach( buffer, size );
    MPI_Bsend( ..., count=20, datatype=type1,
... );
...
    MPI_Bsend( ..., count=40, datatype=type2,
... );
```

величина size в MPI_Buffer_attach должна быть не меньше, чем:

```
    MPI_Pack_size( 20, type1, comm, &s1 );
    MPI_Pack_size( 40, type2, comm, &s2 );
size = s1 + s2 + 2 * MPI_BSEND_OVERHEAD;
```

MPI_BSEND_OVERHEAD — размер служебной информации для каждого передаваемого сообщения.

MPI_Buffer_detach — освобождает выделенный буфер.

Синтаксис:

```
#include "mpi.h"
int   MPI_Buffer_detach(void   *bufferptr,   int
*size )
```

Входные параметры:

buffer — начальный адрес буфера;

size — размер буфера в байтах.

MPI_Isend — неблокирующая пересылка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Isend( void *buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm, MPI_Request
        *request )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

request — указатель на дескриптор операции.

MPI_Ibsend — неблокирующая буферизированная пересылка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Ibsend( void *buf, int count,
```



```
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm, MPI_Request
        *request )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

request — указатель на дескриптор операции.

MPI_Issend — синхронная неблокирующая пересылка сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Issend( void *buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm, MPI_Request
        *request )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

request — указатель на дескриптор операции.

MPI_Irsend — неблокирующая пересылка по готовности.

Синтаксис:

```
#include "mpi.h"
int MPI_Irsend( void *buf, int count,
        MPI_Datatype datatype, int dest,
```

```
        int tag, MPI_Comm comm, MPI_Request
        *request )
```

Входные параметры:

buf — указатель на начало буфера передаваемых данных;

count — число элементов передаваемых данных;

datatype — MPI-тип передаваемых данных;

dest — номер процесса, которому передаются даны;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

request — указатель на дескриптор операции.

MPI_Irecv — неблокирующий прием сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Irecv( void *buf, int count,
        MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm, MPI_Request
        *request )
```

Входные параметры:

buf — указатель на начало буфера, в который будут сохранены полученные данные;

count — число элементов в приемном буфере;

datatype — MPI-тип данных каждого пересылаемого элемента буфера;

source — номер процесса, который отправляет данные;

tag — тэг сообщения;

comm — коммуникатор, в пределах которого передаются данные;

Выходной параметр:

request — указатель на дескриптор операции.

MPI_Wait — ожидание завершения операций передачи или приема сообщений.

Синтаксис:

```
#include "mpi.h"
int MPI_Wait (MPI_Request *request, MPI_Status
        *status)
```


Входной параметр: .

request — указатель на дескриптор операции.

Выходной параметр:

status — статус объекта. Может быть MPI_STATUS_IGNORE.

MPI_Test — проверка завершения операций передачи или приема сообщений.

Синтаксис:

```
#include "mpi.h"
int MPI_Test (MPI_Request *request, int *flag,
        MPI_Status *status)
```

Входной параметр:

request — указатель на дескриптор операции.

Выходные параметры:

flag — логическое значение «ИСТИНА» если операция завершена;

status — статус завершения операции.

MPI_Request_free — освобождает дескриптор операции.

Синтаксис:

```
#include "mpi.h"
int MPI_Request_free( MPI_Request *request )
```

Входной параметр:

request — указатель на дескриптор операции.

Примечание. Функция используется для закрытия дескриптора, созданного MPI_Recv_init , MPI_Send_init или аналогичными функциями. Дескрипторы, созданные при помощи отложенных операций (MPI_Isend, MPI_Irecv), как правило, освобождаются при выполнении функции wait или test (например, MPI_Wait). В общем случае, запрещается закрывать активный дескриптор и использовать закрытый дескриптор в качестве аргументов функций wait или test.

MPI_Waitany — ожидает завершения любой операции обмена из указанных в списке/

Синтаксис:

```
#include "mpi.h"
```

```
int MPI_Waitany(int count, MPI_Request ar-
        ray_of_requests[],
        int *index, MPI_Status *status )
```

Входные параметры:

count — количество дескрипторов в списке;

array_of_requests — массив дескрипторов.

Выходные параметры:

index — порядковый номер (индекс в массиве) дескриптора завершенной операции. Принимает значение от 0 до count-1;

status — статус объекта. Может быть MPI_STATUS_IGNORE.

Примечание. В случае если все дескрипторы определены как MPI_REQUEST_ NULL, функция вернет значение index, равное MPI_ UNDEFINED, и пустой status.

MPI_Testany — проверяет завершение любой операции обмена из определенных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Testany(int count, MPI_Request array_
        of_requests[], int *index, int *flag,
        MPI_Status *status )
```

Входные параметры:

count — количество дескрипторов в списке;

array_of_requests — массив дескрипторов.

Выходные параметры:

index — порядковый номер (индекс в массиве) дескриптора завершенной операции, или MPI_UNDEFINED, если нет завершенной операции;

flag — «ИСТИНА» если хотя бы одна из операций завершена;

status — статус объекта.

MPI_Waitall — ожидает завершения всех операций отправки или приема, определенных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Waitall(int count, MPI_Request array_
```



```
        requests[], MPI_Status array_of_
        statuses[] )
```

Входные параметры:

count — количество дескрипторов в списке;

array_of_requests — массив дескрипторов.

Выходной параметр:

array_of_statuses — массив статусов объектов.

MPI_Testall — проверяет завершение всех инициированных операций обмена, определенных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Testall(int count, MPI_Request array_
        of_requests[], int *flag, MPI_Status
        array_of_ statuses[] )
```

Входные параметры:

count — количество дескрипторов в списке;

array_of_requests — массив дескрипторов.

Выходные параметры:

flag — если все обмены завершены;

array_of_statuses — массив статусов объектов. Может быть MPI_STATUSES_IGNORE.

Примечание. flag принимает значение «ИСТИНА» только если завершены все обмены. В противном случае флаг равен «ЛОЖЬ».

MPI_Waitsome — ожидает завершения некоторых инициированных операций обмена, определенных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Waitsome( int incount, MPI_Request ar-
        ray_of_requests[], int *outcount, int
        array_of_ indices[],MPI_Status array_
        of_statuses[])
```

Входные параметры:

incount — длина массива дескрипторов;

array_of_requests — массив дескрипторов.

Выходные параметры:

outcount — число завершенных операций;

array_of_indices — массив индексов завершенных операций;

array_of_statuses — массив статусов завершенных операций. Может быть MPI_STATUSES_IGNORE.

Примечание. Массив индексов завершенных операций содержит числа в диапазоне от 0 до incount - 1 для C. NULL дескрипторы игнорируются; если все дескрипторы равны NULL, то функция возвращает значение outcount, равное MPI_UNDEFINED.

MPI_Testsome — проверяет завершение некоторых инициированных операций обмена, определенных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount, int array_of_indices[], MPI_Status array_of_statuses[] )
```

Входные параметры:

incount — длина массива дескрипторов;

array_of_requests — массив дескрипторов.

Выходные параметры:

outcount — число завершенных операций;

array_of_indices — массив индексов завершенных операций;

array_of_statuses — массив статусов завершенных операций.

MPI_Iprobe — неблокирующая проверка наличия сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Iprobe( int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status )
```

Входные параметры:

source — ранг отправителя сообщения, или MPI_ANY_SOURCE;

tag — тэг или MPI_ANY_TAG;

comm — коммуникатор.

Выходной параметр:

flag — «ИСТИНА», если сообщение принято;

status — статус завершенной операции.

MPI_Probe — блокирующая проверка наличия сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status *status )
```

Входные параметры:

source — ранг отправителя сообщения, или MPI_ANY_SOURCE;

tag — тэг или MPI_ANY_TAG;

comm — коммуникатор.

Выходной параметр:

status — статус завершения

MPI_Cancel — отмена операции обмена.

Синтаксис:

```
#include "mpi.h"
int MPI_Cancel( MPI_Request *request )
```

Входной параметр:

request — указатель на дескриптор операции.

Примечание. Функция может применяться для отмены только операций приема сообщений (не применима для операций отправки сообщений).

MPI_Sendrecv — одновременный прием и передача сообщений.

Синтаксис:

```
#include "mpi.h"
int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status )
```

Входные параметры:

sendbuf — указатель на начало буфера передаваемых данных;

sendcount — число элементов передаваемых данных;

sendtype — тип элементов в буфере отправки;

dest — номер процесса, которому передаются данные;

sendtag — тэг отправляемого сообщения;

recvcount — число элементов принимаемых данных;

recvtype — тип элементов в буфере приема;

source — номер процесса, который отправляет данные;

recvtag — тэг принимаемого сообщения;

comm — коммуникатор, в пределах которого передаются данные.

Выходные параметры:

recvbuf — указатель на начало буфера, в который будут сохранены полученные данные;

status — статус завершения (для операции приема сообщения).

MPI_Sendrecv_replace — одновременный прием и передача сообщений с одним буфером.

Синтаксис:

```
#include "mpi.h"
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 )
```

Входные параметры:

count — число элементов в буфере приема/передачи;

datatype — тип элементов в буфере приема/передачи;

dest — номер процесса, которому передаются данные;

sendtag — тэг передачи;

source — номер процесса, который отправляет данные;

recvtag — тэг приема;

comm — коммуникатор, в пределах которого передаются данные.

Выходные параметры:

buf — начальный адрес буфера приема/передачи;

status — статус завершения (для операции приема сообщения).

Работа с типами данных

MPI_Type_contiguous — создание нового типа данных, составленного из count идущих подряд блоков исходного.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_contiguous( int count,MPI_Datatype
        old_type, MPI_Datatype *newtype)
```

Входные параметры:

count — число повторений;

oldtype — исходный тип данных.

Выходной параметр:

newtype — новый тип данных.

MPI_Type_vector — создание нового (векторного) типа данных из count блоков исходного. При этом между блоками имеются промежутки длиной stride элементов исходного типа.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_vector( int count, int blocklen,
        int stride, MPI_Datatype old_type,
        MPI_Datatype *newtype )
```

Входные параметры:

count — число блоков;

blocklength — число элементов в каждом блоке;

stride — число элементов между соседними блоками (смещение);

oldtype — исходный тип данных.

Выходной параметр:

newtype — новый тип данных.

MPI_Type_hvector — создание нового (векторного) типа данных из count блоков исходного. При этом длина промежутков составляет stride байтов.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_hvector( int count, int blocklen,
        MPI_Aint stride, MPI_Datatype old_type,
        MPI_Datatype *newtype )
```

Входные параметры:

count — число блоков;

blocklength — число элементов в каждом блоке;

stride — число байт между соседними блоками (смещение);

old_type — исходный тип данных.

Выходной параметр:

newtype — новый тип данных.

MPI_Type_indexed — создание нового (индексного) типа данных из блоков разной длины исходного типа. При этом промежутки между блоками могут быть различными.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_indexed( int count, int block-
        lens[], int indices[], MPI_Datatype
        old_type, MPI_Datatype *newtype )
```

Входные параметры:

count — число блоков;

blocklens — число элементов в каждом блоке (массив);

indices — смещение каждого блока от начала типа в элементах (массив);

old_type — исходный тип данных.

Выходной параметр:

newtype — новый тип данных.

MPI_Type_hindexed — создание нового (индексного) типа данных из блоков разной длины исходного типа. При этом между блоками могут иметься разные промежутки.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_hindexed( int count, int block-
        lens[], MPI_Aint indices[], MPI_Data-
        type old_type, MPI_Datatype *newtype )
```

Входные параметры:

count — число блоков;

blocklens — число элементов в каждом блоке (массив);

indices — смещение каждого блока от начала типа в байтах (массив);

old_type — исходный тип данных.

Выходной параметр:

newtype — новый тип данных.

MPI_Type_struct — определение нового (структурного) типа данных.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_struct( int count, int blocklens[],
        MPI_Aint indices[], MPI_Datatype old_
        types[], MPI_Datatype *newtype )
```

Входные параметры:

count — число блоков;

blocklens — число элементов в каждом блоков (массив);

indices — смещение каждого блока от начала типа в байтах (массив);

old_types — исходный тип данных каждого блока (массив);

Выходной параметр:

newtype — новый тип данных.

MPI_Address — получение адреса области памяти.

Синтаксис:

```
#include "mpi.h"
int MPI_Address( void *location, MPI_Aint
        *address)
```

Входные параметры:

location — область памяти.

Выходной параметр:

address — адрес области памяти.

MPI_Type_commit — регистрация нового (производного) типа данных.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_commit ( MPI_Datatype *datatype )
```

Входной параметр:

datatype — производный тип данных.

MPI_Type_free — освобождение производного типа данных.

Синтаксис:

```
#include "mpi.h"
int MPI_Type_free ( MPI_Datatype *datatype )
```

Входной параметр:

`datatype` — освобождаемый тип данных.

Примечание. Нельзя освобождать предопределенный тип данных.

MPI_Pack — упаковка данных для передачи.

Синтаксис:

```
#include "mpi.h"
int MPI_Pack ( void *inbuf, int incount, MPI_Da-
        tatype datatype, void *outbuf, int out-
        count, int *position, MPI_Comm comm )
```

Входные параметры:

`inbuf` — входной буфер с элементами для упаковки;

`incount` — количество упаковываемых элементов в буфере;

`datatype` — MPI тип данных упаковываемых элементов;

`outcount` — размер выходного буфера в байтах;

`position` — начальное смещение указателя в буфере, в байтах от начала буфера;

`comm` — коммуникатор для упакованного сообщения.

Выходной параметр

`outbuf` — выходной буфер для упакованных элементов.

MPI_Unpack — распаковка принятого сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Unpack ( void *inbuf, int insize, int
        *position, void *outbuf, int outcount,
        MPI_Datatype datatype, MPI_Comm comm )
```

Входные параметры:

`inbuf` — входной буфер с элементами для распаковки;

`insize` — размер входного буфера в байтах;

`position` — начальное смещение указателя в буфере, в байтах от начала буфера;

`outcount` — количество распаковываемых элементов в буфере;

`datatype` — MPI тип данных распаковываемых элементов;

`comm` — коммуникатор для упакованного сообщения.

Выходной параметр:

`outbuf` — выходной буфер для распакованных элементов.

MPI_Pack_size — определение необходимо размера буфера для упаковки сообщения.

Синтаксис:

```
#include "mpi.h"
int MPI_Pack_size ( int incount, MPI_Datatype
        datatype, MPI_Comm comm, int *size )
```

Входные параметры:

`incount` — количество упаковываемых элементов;

`datatype` — MPI тип упаковываемых элементов;

`comm` — коммуникатор для упакованного сообщения.

Выходной параметр:

`size` — верхняя граница размера упакованного сообщения в байтах.

Примечание. Величина `size` является верхней границей размера упакованного сообщения как для `MPI_Pack`, так и для `MPI_Unpack`.

Коллективные взаимодействия

Общие замечания. Операции коллективного взаимодействия в MPI всегда реализуют обмен между процессами одной группы заданного коммуникатора, передаваемого в качестве параметра при вызове функции коллективного взаимодействия, который должен быть выполнен всеми процессами группы.

MPI_Barrier — блокирует все процессы до тех пор, пока они не выполнят эту функцию.

Синтаксис:

```
#include "mpi.h"
int MPI_Barrier ( MPI_Comm comm )
```

Входные параметры:

`comm` — коммуникатор.

Примечание. Блокируется вызвавший функцию процесс до тех пор, пока все процессы коммуникатора не выполнят вызов этой функции.

MPI_Bcast — отправка сообщения от корневого процесса всем остальным процессам группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Bcast ( void *buffer, int count,
        MPI_Datatype datatype, int root,
        MPI_Comm comm )
```

Входные/выходные параметры:

`buffer` — начальный адрес буфера;

`count` — число элементов в буфере;

`datatype` — тип передаваемых данных;

`root` — ранг корневого процесса;

`comm` — коммуникатор, в пределах которого передаются данные.

MPI_Gather — собирает данные от группы процессов.

Синтаксис:

```
#include "mpi.h"
int MPI_Gather ( void *sendbuf, int sendcnt,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm )
```

Входные параметры:

`sendbuf` — начальный адрес буфера отправки;

`sendcount` — число элементов передаваемых данных;

`sendtype` — тип данных передаваемых элементов;

`recvcount` — число элементов собираемых от одного процесса;

`recvtype` — тип данных буфера-приемника;

`root` — ранг процесса-приемника;

`comm` — коммуникатор, в пределах которого передаются данные;

Выходной параметр:

`recvbuf` — адрес приемного буфера.

MPI_Gatherv — собирает разные по объему данные от всех процессов в группе, размещая их в указанное место буфера.

Синтаксис:

```
#include "mpi.h"
int MPI_Gatherv ( void *sendbuf, int sendcnt,
        MPI_Datatype sendtype, void *recvbuf,
        int *recvcnts, int *displs, MPI_Data-
        type recvtype, int root, MPI_Comm
        comm)
```

Входные параметры:

`sendbuf` — начальный адрес буфера отправки;

`sendcount` — число элементов передаваемых данных;

`sendtype` — тип данных передаваемых элементов;

`recvcounts` — массив (его длина равна размеру группы), содержащий число принимаемых от каждого процесса;

`displs` — массив (его длина равна размеру группы). Элемент `i` определяет смещение в выходном буфере, начиная с которого будут размещаться данные, получаемые от процесса `i`;

`recvtype` — тип данных буфера приемника;

`root` — ранг принимающего процесса;

`comm` — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

`recvbuf` — адрес приемного буфера.

MPI_Scatter — пересылает одинаковые по объему блоки данных от корневого процесса ко всем остальным в пределах группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Scatter ( void *sendbuf, int sendcnt,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcnt, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

Входные параметры:

`sendbuf` — начальный адрес буфера отправки;

`sendcount` — число элементов данных передаваемых каждому процессу;

sendtype — тип данных передаваемых элементов;
recvcount — число элементов в буфере приемнике;
recvtype — тип данных буфера приемника;
root — ранг корневого процесса (отправителя);
comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:
recvbuf — адрес буфера для приема.

MPI_Scatterv — пересылает разные по объему блоки данных от корневого процесса ко всем остальным в пределах группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Scatterv ( void *sendbuf, int
        *sendcnts, int *displs, MPI_Datatype
        sendtype, void *recvbuf, int recvcnt,
        MPI_Datatype recvtype, int root,
        MPI_Comm comm )
```

Входные параметры:
sendbuf — начальный адрес буфера отправки;
sendcounts — массив (его длина равна размеру группы), содержащий число элементов передаваемых каждому процессу;
displs — массив (его длина равна размеру группы). Элемент i определяет смещение в буфере отправки, начиная с которого будут передаваться данные процессу i;
sendtype — тип данных передаваемых элементов;
recvcount — число элементов в буфере приемнике;
recvtype — тип данных буфера приемника;
root — ранг корневого процесса;
comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:
recvbuf — адрес приемного буфера.

MPI_Allgather — Собирает одинаковые по объему данные от всех процессов и распределяет их по всем процессам группы.

Синтаксис:

```
#include "mpi.h"
```



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

Входные параметры:
sendbuf — начальный адрес буфера отправки;
sendcount — число элементов передаваемых данных;
sendtype — тип данных в буфере отправки;
recvcount — число элементов, принимаемых от любого процесса;
recvtype — тип данных приемного буфера;
comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:
recvbuf — начальный адрес буфера для приема.

MPI_Allgatherv — собирает разные по объему данные от всех процессов и отправляет их всем процессам группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Allgatherv ( void *sendbuf, int send-
        count, MPI_Datatype sendtype, void
        *recvbuf, int *recvcounts, int
        *displs, MPI_Datatype recvtype,
        MPI_Comm comm )
```

Входные параметры:
sendbuf — начальный адрес буфера отправки;
sendcount — число элементов передаваемых данных;
sendtype — тип данных в буфере отправки;
recvcounts — массив (его длина равна размеру группы), содержащий число элементов принимаемых от каждого процесса;
displs — массив (его длина равна размеру группы). Элемент i определяет смещение в выходном буфере, начиная с которого будут размещаться данные, получаемые от процесса i;
recvtype — тип данных приемного буфера;
comm — коммуникатор, в пределах которого передаются данные.

Выходной параметр:
recvbuf — начальный адрес буфера приема.

MPI_Alltoall — пересылает данные одного объема от всех процессов ко всем.

Синтаксис:

```
#include "mpi.h"
int MPI_Alltoall( void *sendbuf, int sendcount,
        MPI_Datatype sendtype, void *recvbuf,
        int recvcnt, MPI_Datatype recvtype,
        MPI_Comm comm )
```

Входные параметры:

`sendbuf` — начальный адрес буфера отправки;

`sendcount` — число элементов данных, передаваемых каждому процессу;

`sendtype` — тип данных в буфере отправки;

`recvcount` — число элементов данных, принимаемых от каждого процесса;

`recvtype` — тип данных приемного буфера;

`comm` — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

`recvbuf` — начальный адрес буфера приема.

MPI_Alltoallv — пересылает данные различного объема от всех процессов ко всем.

Синтаксис:

```
#include "mpi.h"
int MPI_Alltoallv ( void *sendbuf, int
        *sendcnts, int *sdispls, MPI_Datatype
        sendtype, void *recvbuf, int
        *recvcnts, int *rdispls, MPI_Datatype
        recvtype, MPI_Comm comm)
```

Входные параметры:

`sendbuf` — начальный адрес буфера отправки;

`sendcounts` — массив (его длина равна размеру коммуникатора), содержащий максимальное число передаваемых каждому процессу элементов;

`sdispls` — массив (его длина равна размеру коммуникатора). Элемент `j` определяет смещение (относительно буфера отправки) начиная с которого будут отправлены данные процессу `j`;

`sendtype` — тип данных в буфере отправки;

`recvcnts` — массив (его длина равна размеру коммуникатора), содержащий максимальное число принимаемых от каждого процесса элементов;

`rdispls` — массив (его длина равна размеру коммуникатора). Элемент `i` определяет смещение в выходном буфере, начиная с которого будут размещаться данные, получаемые от процесса `i`;

`recvtype` — тип данных приемного буфера;

`comm` — коммуникатор, в пределах которого передаются данные.

Выходной параметр:

`recvbuf` — начальный адрес буфера приема.

MPI_Reduce — вычисляет и размещает на корневом процессе результат применения бинарной операции к операндам, расположенным на всех процессах группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Reduce (void *sendbuf, void *recvbuf,
        int count, MPI_Datatype datatype,
        MPI_Op op, int root, MPI_Comm comm)
```

Входные параметры:

`sendbuf` — адрес буфера отправки;

`count` — число элементов передаваемых данных;

`datatype` — тип данных буфера отправки;

`op` — операция редукции;

`root` — ранг корневого процесса;

`comm` — коммуникатор, в пределах которого выполняется операция.

Выходной параметр:

`recvbuf` — адрес буфера результата.

Примечание. Эта функция использует простой древовидный алгоритм.

Коллективные операции (`MPI_Op`) не возвращают код ошибки. В результате, если обнаружена ошибка, все процессы могут или вызвать `MPI_Abort` или пропустить эту ситуацию. Таким

образом, если вы измените обработчик ошибки MPI_ERRORS_ARE_FATAL на что-то другое, например, на MPI_ERRORS_RETURN, то ошибка не будет выведена.

MPI_Op_create — создание пользовательских операций.

Синтаксис:

```
#include "mpi.h"
int MPI_Op_create(MPI_User_function *function,
        int commute, MPI_Op *op)
```

Входные параметры:

function — пользовательская функция;

commute — «ИСТИНА» если операция коммутативна; «ЛОЖЬ» в противном случае.

Выходной параметр:

op — дескриптор операции.

MPI_Op_free — освобождает дескриптор пользовательской операции.

Синтаксис:

```
#include "mpi.h"
int MPI_Op_free( MPI_Op *op )
```

Входной параметр:

op — дескриптор операции.

Примечание. op получит значение MPI_OP_NULL после завершения.

MPI_Allreduce — вычисляет и размещает на всех процессах результат применения бинарной операции к операндам, расположенным на всех процессах группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Allreduce ( void *sendbuf, void
        *recvbuf, int count, MPI_Datatype
        datatype, MPI_Op op, MPI_Comm comm )
```

Входные параметры:

sendbuf — начальный адрес буфера отправки;

count — число элементов передаваемых данных;

datatype — тип данных буфера отправки;

op — дескриптор операции;

comm — коммуникатор.

Выходной параметр:

recvbuf — начальный адрес буфера результата.

MPI_Reduce_scatter — вычисляет результат применения бинарной операции к операндам, расположенным на всех процессах группы. Полученный результат рассылается всем процессам группы заданными порциями.

Синтаксис:

```
#include "mpi.h"
int MPI_Reduce_scatter (void *sendbuf, void
        *recvbuf, int *recvcnts, MPI_Datatype
        datatype, MPI_Op op, MPI_Comm comm)
```

Входные параметры:

sendbuf — начальный адрес буфера отправки.

recvcnts — массив, определяющий число элементов, пересылаемых каждому процессу. Массив должен быть одинаковым у всех вызывающих процессов;

datatype — тип данных входного буфера;

op — дескриптор операции;

comm — коммуникатор.

Выходной параметр:

recvbuf — начальный адрес буфера результата.

MPI_Scan — вычисляет частичные суммы на группе процессов

Синтаксис:

```
#include "mpi.h"
int MPI_Scan (void *sendbuf, void *recvbuf, int
        count, MPI_Datatype datatype,MPI_Op
        op, MPI_Comm comm)
```

Входные параметры:

sendbuf — начальный адрес буфера отправки;

count — число элементов во входном буфере;

datatype — тип данных входного буфера;

op — дескриптор операции;

comm -- коммуникатор.

Выходной параметр:
recvbuf — начальный адрес буфера результата.

Операции с группами и коммуникаторами

MPI_Group_size — возвращает число процессов в группе.
Синтаксис:

```
#include "mpi.h"
int MPI_Group_size ( MPI_Group group, int *size )
```

Входной параметр:
group — дескриптор группы.
Выходной параметр:
size — число процессов в группе.

MPI_Group_rank — возвращает ранг процесса в группе.
Синтаксис:

```
#include "mpi.h"
int MPI_Group_rank ( MPI_Group group, int *rank )
```

Входной параметр:
group — дескриптор группы.
Выходной параметр:
rank — ранг вызывающего процесса или MPI_UNDEFINED если он не член группы.

MPI_Group_translate_ranks — установление соответствия между рангами одинаковых процессов в различных группах.
Синтаксис:

```
#include "mpi.h"
int MPI_Group_translate_ranks (MPI_Group
        group_a, int n, int *ranks_a,
        MPI_Group group_b, int *ranks_b)
```

Входные параметры:
group1 — дескриптор 1-ой группы;
n — число элементов массивов ranks1 и ranks2;
ranks1 — массив номеров процессов в первой группе;
group2 — дескриптор 2-ой группы;

Выходной параметр:
ranks2 — массив для сохранения номеров процессов во второй группе или MPI_UNDEFINED если процесса из первой группы нет во второй.

MPI_Group_compare — сравнение двух групп процессов.
Синтаксис:

```
#include "mpi.h"
int MPI_Group_compare (MPI_Group group1,
        MPI_Group group2, int *result)
```

Входные параметры:
group1 — дескриптор 1-ой группы;
group2 — дескриптор 2-ой группы.
Выходной параметр:
result — MPI_IDENT, если наборы процессов и их порядок в группах совпадают, MPI_SIMILAR, если только наборы процессов совпадают, и MPI_UNEQUAL в остальных случаях.

MPI_Comm_group — определяет дескриптор группы в заданном коммуникаторе.
Синтаксис:

```
#include "mpi.h"
int MPI_Comm_group (MPI_Comm comm, MPI_Group
        *group)
```

Входной параметр:
comm — коммуникатор.
Выходной параметр:
group — дескриптор группы в коммуникаторе.
Примечание: Нельзя использовать MPI_COMM_NULL в качестве аргумента MPI_Comm_group.

MPI_Group_union — создает новую группу из процессов нескольких групп.
Синтаксис:

```
#include "mpi.h"
int MPI_Group_union (MPI_Group group1,
        MPI_Group group2, MPI_Group *group_out)
```

Входные параметры:

group1 — дескриптор 1-ой группы;

group2 — дескриптор 2-ой группы;

Выходной параметр:

newgroup — дескриптор объединенной группы.

MPI_Group_intersection — создает новую группу из процессов, входящих одновременно в заданные (пересечение групп).

Синтаксис:

```
#include "mpi.h"
int MPI_Group_intersection (MPI_Group group1,
        MPI_Group group2, MPI_Group *group_out)
```

Входные параметры:

group1 — дескриптор 1-ой группы;

group2 — дескриптор 2-ой группы.

Выходной параметр:

newgroup — дескриптор новой группы.

MPI_Group_difference — создает новую группу из процессов, не входящих одновременно в заданные.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_difference (MPI_Group group1,
        MPI_Group group2, MPI_Group *group_out)
```

Входные параметры:

group1 — дескриптор 1-ой группы;

group2 — дескриптор 2-ой группы.

Выходной параметр:

newgroup — дескриптор новой группы.

MPI_Group_incl — создает новую группу из части процессов старой группы, указанных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_incl (MPI_Group group, int n, int
        *ranks, MPI_Group *group_out)
```

Входные параметры:

group — дескриптор группы;

n — число элементов в массиве ranks (и размер новой группы);

ranks — массив рангов процессов из group которые войдут в newgroup.

Выходной параметр:

newgroup — дескриптор новой группы, процессы получат ранги в порядке ranks.

Примечание. Эта функция не проверяет массив ranks на повторы.

MPI_Group_excl — создает новую группу из части процессов старой группы, не указанных в списке.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_excl ( MPI_Group group, int n,
        int *ranks, MPI_Group *newgroup )
```

Входные параметры:

group — дескриптор группы;

n — число элементов в массиве ranks;

ranks — массив рангов процессов из group которые не войдут в newgroup.

Выходной параметр:

newgroup — дескриптор новой группы, процессы получат ранги в порядке group.

Примечание. Каждый исключаемый процесс (элемент массива ranks) должен быть членом группы group. Элементы масива ranks не должны повторяться.

MPI_Group_range_incl — создает новую группу из части процессов старой группы, ранги которых лежат в заданном диапазоне.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_range_incl (MPI_Group group, int
        n, int ranges[][3], MPI_Group
        *newgroup)
```

Входные параметры:

group — дескриптор группы;

n — число триплетов в массиве ranges;

ranges — одномерный массив триплетов (меньший ранг, больший ранг, шаг) образующий условия отбора процессов из group, включаемых в newgroup.

Выходной параметр:

newgroup — дескриптор новой группы, процессы получат ранги в порядке ranges.

Примечание. Функция не проверяет условия массив ranges на правильность для group.

MPI_Group_range_excl — создает новую группу из части процессов старой группы, ранги которых не лежат в заданном диапазоне.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_range_excl ( MPI_Group group, int
        n, int ranges[][3], MPI_Group
        *newgroup )
```

Входные параметры:

group — дескриптор группы;

n — число триплетов в массиве ranges.

ranges — одномерный массив триплетов (меньший ранг, больший ранг, шаг) образующий условия отбора процессов из group, не включаемых в newgroup.

Выходной параметр:

newgroup — дескриптор новой группы, процессы получат ранги в порядке group.

Примечание. Ранг процесса, исключаемого из group, должен быть допустимым для этой группы.

MPI_Group_free — освобождает дескриптор группы.

Синтаксис:

```
#include "mpi.h"
int MPI_Group_free (MPI_Group *group)
```

Входной параметр:

group — дескриптор группы.

Примечание. По завершению операции дескриптор group устанавливается в MPI_GROUP_NULL.

MPI_Comm_size — определяет число процессов, ассоциированных с коммуникатором.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

Входной параметр:

comm — коммуникатор.

Выходной параметр:

size — число процессов в коммуникаторе comm.

MPI_Comm_rank — определяет ранг вызывающего процесса в коммуникаторе.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Входные параметры:

comm — коммуникатор.

Выходной параметр:

rank — ранг вызывающего процесса в коммуникаторе comm.

MPI_Comm_compare — сравнение двух коммуникаторов.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_compare ( MPI_Comm  comm1,
        MPI_Comm  comm2, int *result )
```

Входные параметры:

comm1 — коммуникатор 1;

comm2 — коммуникатор 2.

Выходной параметр:

Результатом будет MPI_IDENT если у коммуникаторов совпадают контекст и группы, MPI_COMGRUENT, если контексты отличаются, а группы совпадают, MPI_SIMILAR, если при различных контекстах наборы (но не порядок) процессов в группах совпадают и MPI_UNEQUAL в остальных случаях.

Примечание. Нельзя использовать MPI_COMM_NULL как аргумент MPI_Comm_compare.

MPI_Comm_create — создает новый коммуникатор.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_create ( MPI_Comm comm, MPI_Group
        group, MPI_Comm *comm_out )
```

Входные параметры:

comm — коммуникатор;

group — группа, которая является подмножеством группы коммуникатора comm.

Выходной параметр:

comm_out — новый коммуникатор.

MPI_Comm_dup — дублирует существующий коммуникатор.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_dup ( MPI_Comm comm, MPI_Comm
        *comm_out )
```

Входной параметр:

comm — коммуникатор.

Выходной параметр:

newcomm — новый коммуникатор, включающий ту же группы что и comm но с новым контектсом.

MPI_Comm_split — разбиение коммуникатора на несколько новых.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_split (MPI_Comm comm, int color,
        int key, MPI_Comm *comm_out)
```

Входные параметры:

comm — исходный коммуникатор;

color — индикатор разбиения. Процессы с одинаковым значением color попадут в один коммуникатор;

key — порядок присвоения рангов процессам в создаваемых коммуникаторах.

Выходной параметр:

newcomm — новый коммуникатор.

Примечание. Параметр color должен быть неотрицательным или MPI_UNDEFINED.

MPI_Comm_free — помечает коммуникатор как предназначенный для освобождения.

Синтаксис:

```
#include "mpi.h"
int MPI_Comm_free (MPI_Comm *comm)
```

Входной параметр:

comm — освобождаемый коммуникатор.

Приложение 2
ОСНОВНЫЕ ФУНКЦИИ МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ

pthread_create
Создание потока

Синтаксис:

```
#include <pthread.h>
int pthread_create(pthread_t *restrict
   thread,const pthread_attr_t *restrict
   attr, void *(*start_routine)(void*), void
   *restrict arg);
```

Функция pthread_create() инициирует новый поток, с атрибутами attr, внутри процесса. Если в качестве attr передается NULL, то применяются атрибуты по умолчанию. Если атрибуты attr в дальнейшем будут изменены, то это никак не отразится на потоке. В случае успешного завершения, pthread_create() сохранит идентификатор ID созданного потока по адресу thread.

Поток создается запуском функции start_routine с единственным аргументом arg. Если происходит завершение start_routine, эффект будет таким же, как если бы был выполнен неявный вызов функции завершения pthread_exit() используя возвращаемое значение от start_routine как статус завершения. Поведение процесса, в котором main() был первоначально запущен, отличается от этого. Когда происходит возврат из main(), эффект будет таким же, как если бы был выполнен неявный вызов функции завершения exit() используя возвращаемое значение от main() как статус завершения.

Набор сигналов нового потока должен быть установлен следующим образом:

• маска сигналов должна наследоваться у родительского потока;

• набор необработанных (pending) сигналов ожидания пуст.

Окружение выполнения операций с плавающей точкой наследуется от родительского потока.

Если функция pthread_create() будет завершена с ошибкой, то новый поток не будет создан и его идентификатор (содержимое по адресу thread) не будет определен.

Если определено значение _POSIX_THREAD_CPUTIME, то при выделении новому потоку CPU начальное значение счетчика времени будет установлено в ноль.

Выходные параметры:

• возвращает значение целого типа, которое равняется нулю в случае успешного завершения и содержит код ошибки в противном случае.

Ошибки:

• Функция pthread_create() завершается с ошибкой в одном из следующих случаев:

[EAGAIN] — системе не хватает необходимых для создания потока ресурсов, или уже достигнуто ограничение на число создаваемых в процессе потоков заданное в {PTHREAD_THREADS_MAX}.

[EPERM] — вызывающее приложение не имеет необходимых прав для установки или изменения политики планирования.

• функция pthread_create() может завершиться с ошибкой в одном из следующих случаев:

[EINVAL] — ошибочные атрибуты attr.

• функция pthread_create() не должна возвращать код ошибки [EINTR].

pthread_exit

Завершение потока

Синтаксис:

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

Функция *pthread_exit()* завершает поток, который ее вызвал и делает значение *value_ptr* доступным функции pthread_join, вызываемой для этого потока. Обработчики завершения потока (cancellation cleanup handlers), идентификаторы которых сохранены в стеке, должны закрываться в порядке обратном их помещению в стек. После того как все обработчики будут завершены, если поток имеет данные, ассоциированные с ключами, соответствующие деструкторы могут быть вызваны в произвольном порядке. Завершение потока автоматически не освобождает любые принадлежащие ему ресурсы, включая мьютексы и дескрипторы файлов, если они только не были предварительно закрыты специальными процедурами.

Неявный вызов функции pthread_exit() происходит, если поток, отличный от того в котором был запущен main(), завершает выполнение функции, использованной для его создания. Возвращаемое значение должно быть сохранено как статус завершения потока.

Поведение функции pthread_exit() не определено, если вызов происходит из обработчика завершения потока или деструкторов данных, ассоциированных с ключами.

После завершения потока его локальные переменные уничтожаются. Это означает, что в качестве аргумента value_ptr функции pthread_exit() нельзя использовать ссылки на локальные переменные завершенного потока.

Процесс будет завершен с кодом завершения 0 после завершения последнего, связанного с ним потока. Поведение аналогично ситуации, когда вызывается exit() с нулевым аргументом.

Выходные параметры:

• функция pthread_exit() не имеет возвращаемых параметров, так как не возвращает управления в вызываемую функцию.

Ошибки не определены.

pthread_join

Ожидание завершения потока

Синтаксис:

```
#include <pthread.h>
int pthread_join(pthread_t thread, void
   **value_ptr);
```

Функция pthread_join() приостанавливает выполнение вызывающего потока до тех пор, пока поток с идентификатором thread не будет завершен, при условии, что такой поток *thread* уже не был завершен до вызова функции.

При возврате из успешно завершенной функции pthread_join() с аргументом value_ptr, не равным NUUL, значение, передаваемое функции pthread_exit() после завершения потока thread, будет доступно по адресу value_ptr. Функция pthread_join() завершается успешно, после того как поток thread был завершен. Результат одновременного вызова функции pthread_join() с одинаковым значением идентификатора потока thread несколькими потоками будет неопределенным.

Стандарт не определяет, учитывается ли завершенный, но не отсоединенный поток при сравнении числа запущенных потоков с числом PTHREAD_THREADS_MAX.

Выходные параметры:

В случае успешного завершения, функция pthread_join() возвращает ноль, в противном случае — код ошибки.

Ошибки:

• функция pthread_join() завершается с ошибками в следующих ситуациях:

[ESRCH] — не существует потока с идентификатором ID;

• функция pthread_join() может быть завершена с ошибкой в следующих ситуациях:

[EDEADLK] — обнаружен дедлок или поток пытается завершить сам себя;

[EINVAL] — идентификатор потока *thread* не указывает на поток, который является корректным аргументом функции pthread_join;

• функция pthread_join() не возвращает код ошибки — [EINTR].

Пример

Создание и удаление потока:

```
typedef struct {
    int *ar;
    long n;
} subarray;
void *
incer(void *arg)
{
    long i;
    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}
int main(void)
{
    int         ar[1000000];
    pthread_t   th1, th2;
    subarray    sb1, sb2;
    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);
    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);
    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```

pthread_mutex_destroy
pthread_mutex_init

Уничтожение и инициализация мьютекса

Синтаксис:

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t
   *mutex);
```



```
int pthread_mutex_init(pthread_mutex_t
   *restrict mutex, const
   pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex =
   PTHREAD_MUTEX_INITIALIZER;
```

Функция pthread_mutex_destroy() уничтожает мьютекс (mutex), в результате чего он становится неинициализированным. Попытка обращения к неинициализированному мьютексу приведет к неопределенной ситуации. уничтоженный мьютекс может быть повторно инициализирован с помощью функции pthread_mutex_init().

Разрешается уничтожать только инициализированный незаблокированный мьютекс. Попытка удалить заблокированный мьютекс приведет к неопределенной ситуации.

Функция pthread_mutex_init() инициализирует мьютекс (mutex) и присваивает ему атрибуты (attr). Если атрибуты мьютекса не указаны или равны NULL, используются атрибуты по умолчанию. После успешной инициализации мьютекс становится инициализированным и незаблокированным.

Для синхронизации потоков можно использовать только значение, хранящееся в mutex. Использование копий *mutex* в вызовах функций pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock(), и pthread_mutex_destroy() приведет к неопределенной ситуации. Повторная инициализация мьютекса также приведет к неопределенной ситуации. Для инициализации статических мьютексов может быть использован макрос PTHREAD_MUTEX_INITIALIZER, если атрибуты мьютекса это допускают. Это равносильно динамической инициализации при помощи функции pthread_mutex_init() с параметром attr, равным NULL, с точностью до проверки ошибок.

Выходные параметры:

• в случае успешного выполнения, функции pthread_mutex_destroy() и pthread_mutex_init() возвращают ноль, в противном случае возвращается код ошибки;

• проверка ошибок [EBUSY] и [EINVAL] производится в начале выполнения функций и возвращает код ошибки до изменения состояния мьютекса.

Ошибки:

• функция pthread_mutex_destroy() может завершиться с ошибкой в следующих случаях:

[EBUSY] — выполнена попытка уничтожить заблокированный или связанный с другим потоком;

[EINVAL] — задано неверное значение mutex;

• функция pthread_mutex_init() завершается с ошибкой в следующих случаях:

[EAGAIN] — системе не хватает необходимых ресурсов (не память) для инициализации другого мьютекса;

[ENOMEM] — недостаток памяти для инициализации мьютекса;

[EPERM] — вызывающий поток не обладает нужным уровнем прав для выполнения операции.

pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock

Блокировка и освобождение мьютекса

Синтаксис:

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t
   *mutex);
int pthread_mutex_trylock(pthread_mutex_t
   *mutex);
int pthread_mutex_unlock(pthread_mutex_t
   *mutex);
```

Функция *pthread_mutex_lock()* блокирует объект mutex. Если мьютекс заблокирован, вызывающий процесс блокируется до его освобождения. Эта операция возвратит заблокирует mutex установит вызывающий поток «хозяином» мьютекса.

Если тип мьютекса задан как PTHREAD_MUTEX_NORMAL, то проверка взаимной блокировки (deadlock) не выполняется, поэтому попытка повторного захвата мьютекса приведет к блокировке. Если поток пытается разблокировать свободный или заблокированный не им мьютекс, то поведение не определено.

Если тип мьютекса задан как PTHREAD_MUTEX_ERRORCHECK, то выполняется проверка ошибок. Если поток пытается повторно заблокировать мьютекс, то будет возвращен код ошибки. Если поток пытается разблокировать свободный или заблокированный не им мьютекс, то также возвращается код ошибки.

Если тип мьютекса задан как PTHREAD_MUTEX_RECURSIVE, используется поддержка счетчика блокировок. Когда поток в первый раз захватит мьютекс, счетчик блокировок будет равен 1. Каждый последующий захват мьютекса увеличивает, а каждое освобождение уменьшает счетчик на 1. Мьютекс будет доступен другим потокам, если значение счетчика станет равным 0. Если поток пытается разблокировать свободный или заблокированный не им мьютекс, будет возвращен код ошибки.

Если тип мьютекса задан как PTHREAD_MUTEX_DEFAULT, попытка рекурсивно заблокировать мьютекс приведет к неопределенной ситуации. Попытка разблокировать незаблокированный вызывающим процессом мьютекс приведет к неопределенной ситуации. Попытка разблокировать незаблокированный мьютекс приведет к неопределенной ситуации.

Функция pthread_mutex_trylock() эквивалентна pthread_mutex_lock(), кроме случаев, когда объект mutex заблокирован (любым потоком, включая вызывающий). При этом будет выполнен немедленный возврат из функции. Если тип мьютекса задан как PTHREAD_MUTEX_RECURSIVE и мьютекс уже принадлежит вызывающему потоку, то счетчик свободных блокировок мьютекса будет увеличен на 1 и функция pthread_mutex_trylock() завершится без ошибок (код ошибки 0).

Функция pthread_mutex_unlock() разблокирует объект mutex. Способ которым разблокируется мьютекс зависит от типа его атрибутов. Если есть потоки, ожидающие разблокировки объекта mutex при вызове pthread_mutex_unlock(), то выбор потока, которому будет передан в управление мьютекс, зависит от планировщика задач. (Если тип мьютекса задан как PTHREAD_

MUTEX_RECURSIVE, то мьютекс станет свободным, когда счетчик будет равен 0.)

Выходные параметры:

• в случае успешного завершения функции pthread_mutex_lock() и pthread_mutex_unlock() возвращают 0; в противном случае возвращается код ошибки;

• функция pthread_mutex_trylock() возвратит 0, если блокировка мьютекса проведена успешно. В противном случае возвращается код ошибки.

Ошибки:

• функции pthread_mutex_lock() и pthread_mutex_trylock() завершаться с ошибкой в следующих случаях:

[EINVAL] — мьютекс был создан с атрибутом PTHREAD_PRIO_PROTECT, а приоритет вызывающего потока выше, чем текущий приоритет мьютекса;

• Функция pthread_mutex_trylock() завершится с ошибкой, если:

[EBUSY] — мьютекс не может быть захвачен, т.к. он уже заблокирован;

• функции pthread_mutex_lock(), pthread_mutex_trylock() и pthread_mutex_unlock() могут завершиться с ошибкой в одном из следующих случаев:

[EINVAL] — аргумент mutex ссылается на неинициализированный мьютекс;

[EAGAIN] — мьютекс не может быть захвачен, т.к. достигнуто ограничение на число рекурсивных блокировок;

• функция pthread_mutex_lock() может завершиться с ошибкой если:

[EDEADLK] — обнаружен дедлок или текущий поток уже захватил мьютекс;

• функция pthread_mutex_unlock() может завершиться с ошибкой, если:

[EPERM] — вызывающий поток не является владельцем мьютекса.

. pthread_self

Получение идентификатора потока

Синтаксис:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Функция pthread_self() возвращает уникальный идентификатор того потока, который вызвал эту функцию.

Выходной параметр:

• идентификатор потока.

Ошибки не определены.

Приложение 3
Учебный компьютерный класс, как средство реализации параллельных вычислений

Изучив теоретические разделы книги, читатель вправе задать вопрос, а как же можно реализовать полученные знания на практике? Еще несколько лет назад можно было бы только порекомендовать обратиться к владельцам суперкомпьютеров, но сегодня наш выбор весьма широк. Например, многие студенты и аспиранты с успехом используют кампусные вычислительные сети для своих исследований. Этот путь является наиболее простым с точки зрения преодоления административных преград, но он вряд ли позволит проводить серьезные эксперименты. Для этого потребуется, чтобы некоторая группа компьютеров была постоянно включена и не использовалась другими приложениями.

Хотелось бы обратить внимание читателей-студентов на возможность использования компьютерных классов в свободное от занятий время. Напомним, что подавляющее большинство высокопроизводительных вычислительных систем представленных в мировом рейтинге TOP500 относятся к кластерным системам. При этом в каждом университете имеется достаточное количество персональных компьютеров в классах, которые можно использовать в качестве узлов кластера. С архитектурной точки зрения такой кластер относится к системам типа CoPC (Cluster of PC). Чтобы не быть голословными приведем результаты исследования производительности типичного кластера такого типа.

Топология кластера представлена на рисунке П3.1. Он активно используется в учебном процессе на кафедре «Вычислительная техника» Московского института электронной техники. Единственным отличием от большинства подобных компьютерных классов является использование в качестве коммуникационной среды Gigabit Ethernet. Сеть кластера состоит из трех сегментов: *A*, *B* и *C*. Сегменты *A* и *C* используют 8-ми портовые коммутаторы (без вентиляторов), а сегмент *B* — 16 портовый коммутатор, который и объединяет все сегменты сети. Все коммутаторы фирмы Zyxel.

Несколько слов об узлах кластера. Это полноценные персональные компьютеры со следующими характеристиками:

- процессор Intel Pentium IV (Prescott) 2.4GHz
- оперативная память 512Mb DDR 333
- материнская плата Gigabyte 1000MK
- сетевая карта Compex RE100ATX/WOL
- видеоадаптер GeForce 2 MX400
- операционная система MS WINDOWS 2000.

В качестве среды параллельного программирования была выбрана свободно распространяемая реализация MPI — MPICH.NT (версия 1.2.2), доступная на сайте http://www-unix.mcs.anl.gov/mpi/mpich1/mpich-nt/.

Пиковая производительность кластера составляет 115,2 Gflops.

В качестве средства оценки производительности используется тест HPL (www.netlib.org/benchmark/hpl/).

Задача экспериментов состоит в том, чтобы показать, что компьютерный класс можно рассматривать не только как среду для организации распределенных вычислений, но и как мощную кластерную систему типа CoPC, пригодную для реализации параллельных приложений. Этим обстоятельством объясняется выбор теста Linpack.

Исследования производительности кластера проводились в несколько этапов:

- оценка производительности сегмента *A*(*C*);
- оценка производительности сегмента *B*;
- оценка производительности сегментов *A*+*C*;
- оценка производительности всего кластера — сегменты *A*+*B*+*C*.

Результаты исследований представлены в соответствующих таблицах. Проанализируем полученные данные.

Сегменты *A* и *C*, построенные на 8-ми портовых коммутаторах, показывают хорошую масштабируемость и линейную зависимость производительности от числа работающих узлов (табл. П3.1).

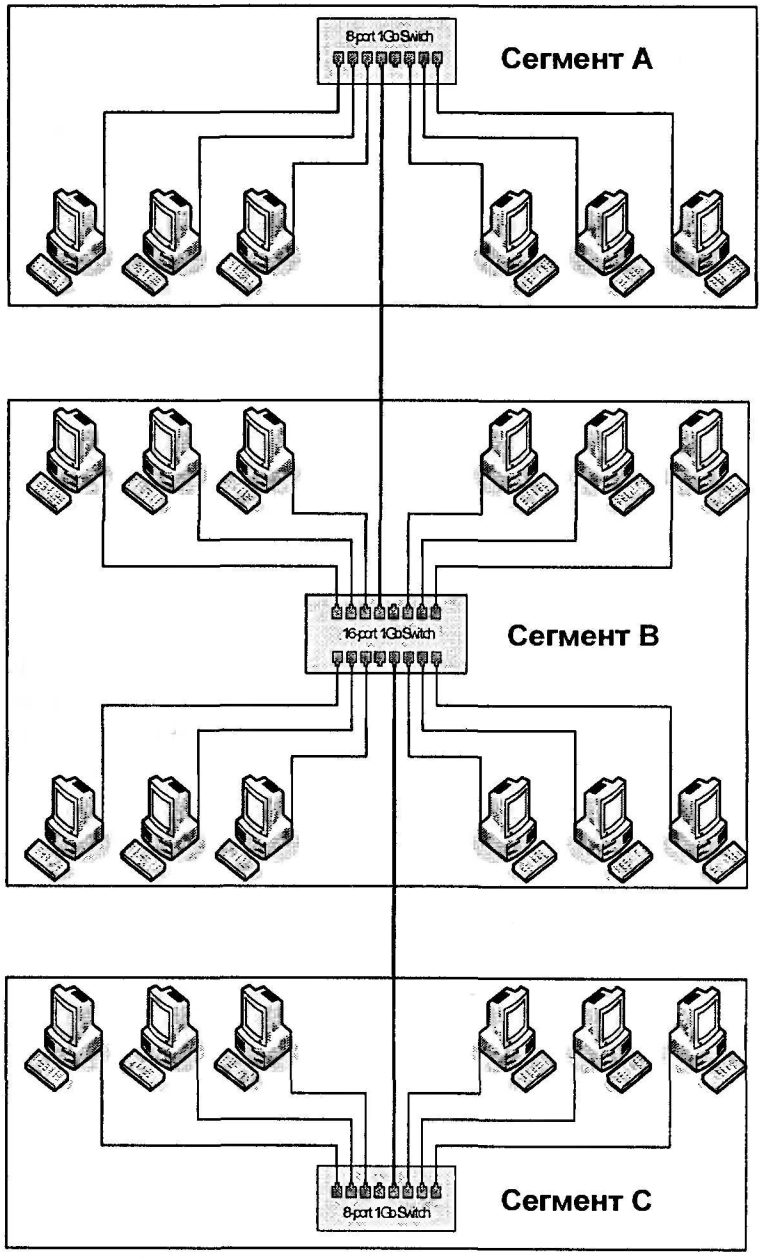


Рис. П3.1. Топология кластера CoPC

Считается, что для построения кластеров лучше использовать коммутаторы с большим числом портов. Теоретически, производительность сегмента *B* должна быть больше, чем суммарная производительность сегментов *A* и *C*. Данные по сегменту *B* приведены в табл. П3.2 и на рис. П3.4, а по сегментам *A* и *C* в табл. П3.3. Сравнивая производительность 12 узлов для этих сегментов, можно сделать заключение о том, что для небольших кластеров можно использовать и коммутаторы с малым числом портов. При этом эффективность использования узлов практически не уменьшается.

Данные по всему кластеру представлены в таблице П3.4. Суммарная производительность, полученная на матрице размером 35100, составила 64,86 Gflops. Еще в сентябре 2005 года этот результат позволял входить в список самых мощных компьютеров СНГ TOP50. Отметим, что со 100 мегабитной коммутацией (наиболее характерной для компьютерных классов общего назначения) эта конфигурация демонстрировала производительность только 25 Gflops, что подчеркивает необходимость использования быстрой коммутации для достижения высокой производительности при параллельных вычислениях. Кластер показывает вполне удовлетворительную масштабируемость, во всем диапазоне исследований деградации не наблюдалось. Узлы кластера обеспечивают работу без сбоев в течение 8–12 часов, а вот 8-портовые коммутаторы требуют дополнительного охлаждения для устойчивой работы.

Полученные результаты подтверждают наш тезис о том, что обычный компьютерный класс в свободное от занятий время можно с успехом использовать для проведения широкого спектра исследований в различных областях науки.

Таблица П3.1. **Производительность сегмента *A* (*C*)**

Число узлов	Производительность (Gflops)	Размер матрицы	Конфигурация
1	3,9	6500	1×1
2	7,015	8550	1×2
3	10,08	11900	1×3
4	12,87	14500	1×4
5	15,64	15500	1×5
6	19,09	17500	2×3

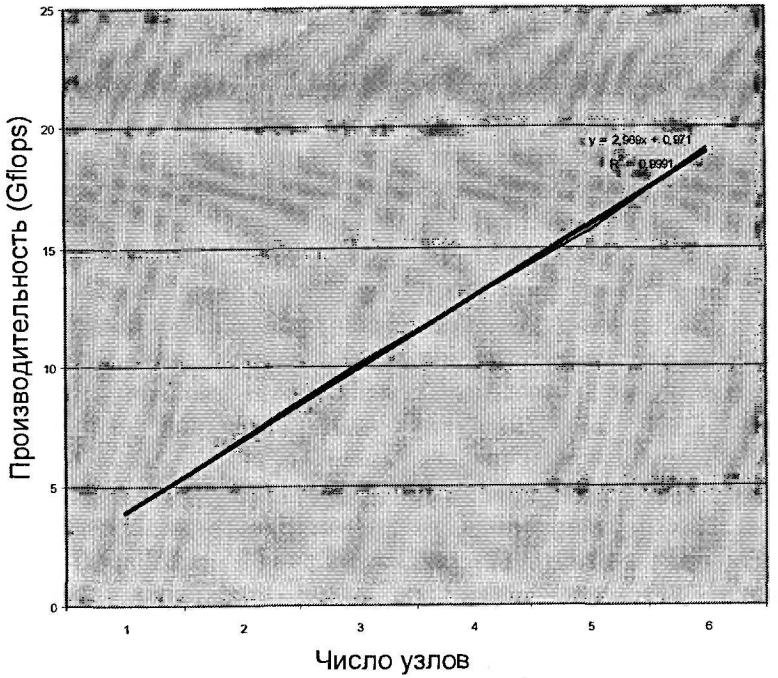


Рис. П3.2. Производительность сегмента *A* (*C*)

Таблица П3.2. **Производительность сегмента *B***

Число узлов	Производительность (Gflops)	Размер матрицы	Конфигурация
1	3,9	6500	1×1
2	7,071	8500	1×2
3	10,37	11900	1×3
4	13,12	14550	1×4
5	15,95	15450	1×5
6	19,14	17500	2×3
7	21,39	19000	1×7
8	24,84	20200	2×4
9	25,89	22050	1×9
10	31,38	23550	2×5
11	30,32	24450	1×11
12	36,07	25400	2×6

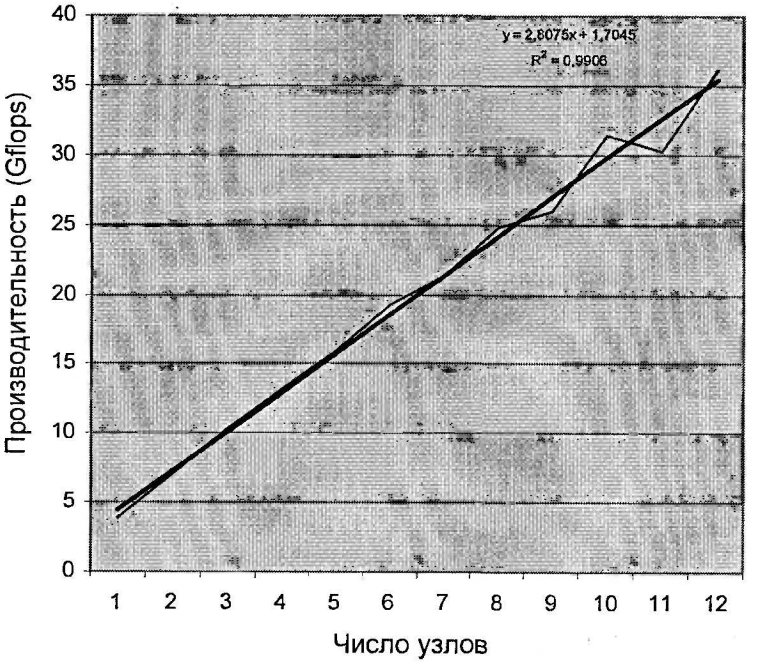


Рис П3.3. Производительность сегмента *B*

Таблица П3.3. **Производительность сегментов *A*+*C***

Число узлов	Производительность (Gflops)	Размер матрицы	Конфигурация
1	3,9	6500	1×1
2	7,015	8550	1×2
3	10,08	11900	1×3
4	12,87	14500	1×4
5	15,64	15500	1×5
6	19,09	17500	2×3
7	21,13	19100	1×7
8	23,54	22150	2×4
9	25,4	22000	1×9
10	30,57	23650	2×5
11	30,11	24250	1×11
12	35,09	25400	2×6

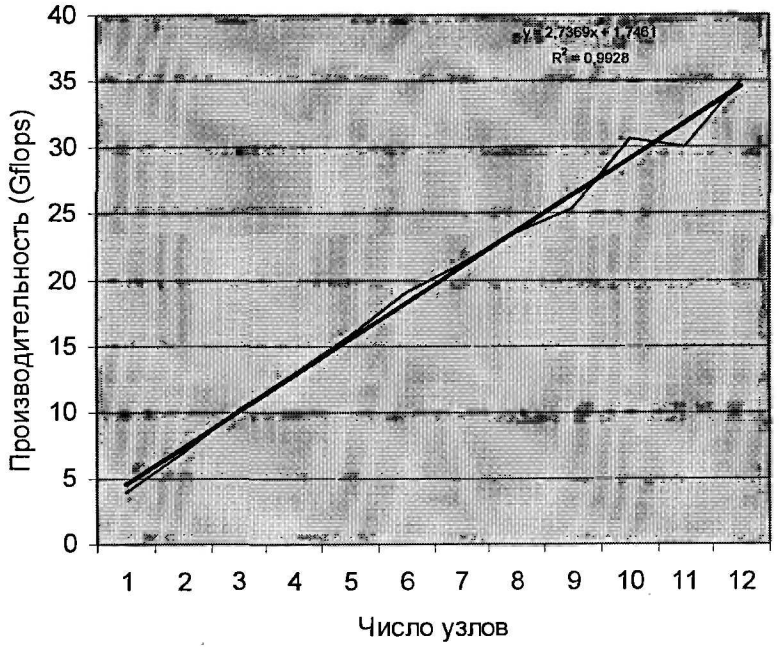


Рис. П3.4. Производительность сегментов *A+C*

Таблица П3.4. **Производительность кластера (сегменты *A+B+C*)**

Число узлов	Производительность (Gflops)	Размер матрицы	Конфигурация
1	3,9	6500	1×1
2	7,015	8550	1×2
3	10,08	11900	1×3
4	12,87	14500	1×4
5	15,64	15500	1×5
6	19,09	17500	2×3
7	21,43	19000	1×7
8	25,46	20150	2×4
9	25,96	22000	1×9
10	31,31	23450	2×5
11	30,66	24350	1×11

Окончание табл. П3.4

Число узлов	Производительность (Gflops)	Размер матрицы	Конфигурация
12	36,12	25400	2×6
13	34,32	26500	1×13
14	41,25	27700	2×7
15	38,19	28900	3×5
16	46,06	29600	2×8
17	36,3	30750	1×17
18	51,42	31500	2×8
19	42,36	32400	1×19
20	57,84	33400	2×10
21	52,97	34000	3×7
22	58,1	30600	2×11
23	51,96	31050	1×23
24	64,86	35100	2×12

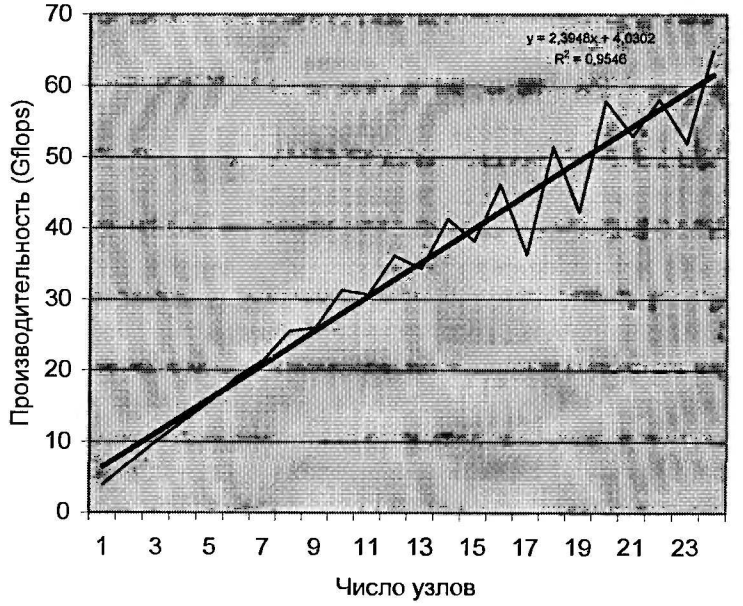


Рис. П3.5. Производительность кластера (сегменты *A+B+C*)

Приложение 4
ЯЗЫК ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ MPC

Язык программирования mpC — это расширение языка Си, разработанное специально для параллельных вычислений на обычных сетях рабочих станций. Так как сети рабочих станций редко состоят из компьютеров одинаковой производительности, то в язык были введены специальные средства, поддерживающие неоднородные платформы. Однако, опыт разработки приложений на mpC показал, что язык может с успехом использоваться и для программирования суперкомпьютеров. Основным достоинством среды mpC является легкость разработки и отладки программ по сравнению с традиционными инструментами параллельного программирования.

Основным понятием языка является *объект сетевого типа* или просто *сеть*. Сетью называется совокупностью процессоров, которая может быть использована для описания объектов данных или распределения вычислений. Процессоры сети являются виртуальными: их количество может отличаться от количества вычислительных узлов в системе. Оптимальное отображение виртуальных процессоров на вычислительные узлы выполняется динамически системой поддержки времени выполнения.

Простейшая сеть, обозначаемая зарезервированным идентификатором `host`, состоит только из одного главного процессора. Другим базовым вариантом сети является сеть предопределенного типа `SimpleNet`. С помощью этого сетевого типа можно описать одномерную сеть из заданного количества процессоров. Например сеть `mynet` из `N` процессоров определяется следующим образом:

```
net SimpleNet(N) mynet;
```

Созданную сеть можно использовать для распределения данных и вычислений. Для этого блок вычислений или объект данных, распределенные по сети, предваряется ее идентификатором, заключенным в квадратные скобки. Например, следующая программа выполняет печать значения на каждом из узлов сети `mynet`:

```
1:  #include <mpc.h>
2:  int [*]main() {
3:    net SimpleNet(N) mynet;
4:    [mynet]printf("Hello, world!\n");
5:  }
```

Рассмотрим программу нахождения значения числа π методом Монте-Карло. Используется следующий подход: с помощью генератора случайных чисел получаем пару чисел (в диапазоне от 0 до 1), которые будем рассматривать как координаты и некоторой точки. Теперь, если взять первый квадрант круга единичного радиуса, можно ответить на вопрос – попала эта точка в круг или за его границу. Проводя испытания, будем подсчитывать число точек попавших в круг (оценка площади круга). Общее число бросков даст нам оценку площади описанного квадрата. Для нахождения приближенного значения числа π достаточно найти отношение оценок площадей квадрата и круга.

Перейдем к рассмотрению текста программы. В начале подключаются заголовочные файлы библиотек и определяются основные переменные:

```
1:  #include "mpc.h"
2:  #include "time.h"
3:  #include "math.h"
4:  #include "stdlib.h"
5:  #include "stdio.h"
6:  #define N 3
7:  int [*]main(int argc, char* argv[])
8:  {
9:    net SimpleNet(N) mynet;
10:   double Cpx,cpy;
11:   double CurrentRadius;
12:   long Counter;
13:   long npart = 100000000;
```

```
14:   long [mynet]npartc = 0;
15:   int pn;
16:   double StartTime,EndTime;
17:   double PII,PII_Mismartch,PII_Real;
```

Среди этих переменных — сеть mynet, которая используется далее для выполнения итераций метода Монте-Карло:

```
18:   [mynet]:{
19:     time_t ti;
20:     StartTime = MPC_Wtime();
21:     pn = I coordof mynet;
22:     srand(time(&ti)*pn);
23:     for(Counter=0;Counter<npart;
            Counter++){
24:       Cpx=rand()/(RAND_MAX+1.0);
25:       cpy=rand()/(RAND_MAX+1.0);
26:     CurrentRadius= Cpx*Cpx + cpy * cpy;
27:    if (CurrentRadius <= 1) {
28:           npartc++;
29:       }
30:     }
```

Функция MPC_Barrier предназначена для синхронизации и аналогична по функциональность барьерной синхронизации в MPI:

```
31:   ([(N)mynet])MPC_Barrier();
```

В строке 32 производится суммирование значений переменной npartc, хранимых на различных узлах сети mynet. Для этого используется специальная операция mpC — постфиксная редукция. В постфиксной редукции бинарная операция записывается в квадратных скобках за именем переменной. В результате применения такой операции вычисляется значение, полученное применением этой операции к значениям переменной на различных узлах сети.

```
32:   npartc=npartc[+];
33:   ([(N)mynet])MPC_Barrier();
```


После очередной синхронизации на главном процессоре выводятся на печать вычисленные значения:

```
34:   [host]:
35:   {
36:     PII_Real = 3.14159265358932385;
37:     PII=(double)((npartc*4.0)/
            (npart*N));
38:     PII_Mismartch=fabs(PII_Real-PII);
39:     EndTime = MPC_Wtime();
40:     printf("Time: %4.17f sec\n",
              EndTime-StartTime);
41:     printf("value of PI: %4.17lf\n",
              PII_Real);
42:     printf("PI calculated: %4.17lf\n",
              PII);
43:   }
44: }
```

Мы привели лишь очень краткое изложение основных возможностей mpC. В заключении отметим, что язык mpC был создан в Институте системного программирования РАН группой исследователей под руководством А.Л. Ластовецкого. Дополнительную информацию о языке mpC и можно взять из его монографии Parallel Computing on Heterogeneous Networks, а документацию и последние версии — с веб-сайта http://hcl.ucd.ie/Software/mpC+++HeteroMPI.

Сергей Андреевич Лупин
Михаил Анатольевич Посыпкин

ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

Корректор *Н.Н. Морозова*
Компьютерная верстка *С.Ч. Соколовского*
Оформление серии *К.В. Пономарева*

Сдано в набор 05.07.2007. Подписано в печать 03.10.2007.
Формат $60 \times 90^1/_{16}$. Бумага офсетная. Гарнитура «Таймс».
Печать офсетная. Усл. печ. л. 13. Уч.-изд. л. 13,5.
Тираж 2000 экз. Заказ № 8295.

ЛР № 071629 от 20.04.98
Издательский Дом «ФОРУМ»
101000, Москва-Центр, Колпачный пер., 9а
Тел./факс: (495) 625-39-27
E-mail: forum-books@mail.ru

ЛР № 070824 от 21.01.93
Издательский Дом «ИНФРА-М»
127282, Москва, Полярная ул., 31в
Тел.: (495) 380-05-40; факс: (495) 363-92-12
E-mail: books@infra-m.ru
http://www.infra-m.ru

По вопросам приобретения книг обращайтесь:

Отдел продаж «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 363-42-60
Факс: (495) 363-92-12
E-mail: books@infra-m.ru

Центр комплектования библиотек
119019, Москва, ул. Моховая, д.16
(Российская государственная библиотека, кор. К)
Тел.: (495) 202-93-15

Магазин «Библиосфера» (розничная продажа)
109147, Москва, ул. Марксистская, д. 9
Тел.: (495) 670-52-18, (495) 670-52-19

Отпечатано с готовых диапозитивов в ОАО ордена «Знак Почета»
«Смоленская областная типография им. В. И. Смирнова».
214000, г. Смоленск, проспект им. Ю. Гагарина, 2.

ФОРУМ

КНИГИ

ФОРУМ

Издательского Дома «ФОРУМ»

право
•
экономика
•
психология
•
педагогика
•
техническая
литература
•
информационные
и компьютерные
технологии

УЧЕБНИКИ
для вузов,
техникумов,
колледжей и лицеев
ПОСОБИЯ
для поступающих
в вузы
СПРАВОЧНАЯ
ЛИТЕРАТУРА

Приглашаем к сотрудничеству

ПО ВОПРОСАМ ПРИОБРЕТЕНИЯ
ЛИТЕРАТУРЫ И
С ПРЕДЛОЖЕНИЯМИ
ПО ИЗДАНИЯМ
ПРОСИМ ОБРАЩАТЬСЯ
ПО АДРЕСУ:
101000, Г. МОСКВА-ЦЕНТР,
КОЛПАЧНЫЙ ПЕР., 9А.
ТЕЛ.:(495) 625-39-27,
ФАКС:(495) 625-39-27,
E-MAIL: FORUM-BOOKS@MAIL.RU