4. Параллельное программирование на основе MPI

Данный раздел посвящен рассмотрению методов параллельного программирования для вычислительных систем с распределенной памятью с использованием MPI.

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

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

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

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

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

Следует отметить, что попытки создания программных средств передачи данных между процессорами начались предприниматься практически сразу с появлением локальных компьютерных сетей – ряд таких средств, представлен, например, в Воеводин В.В. и Воеводин Вл.В. (2002), Buyya (1999), Andrews (2000) и многих других. Однако подобные средства часто были неполными и, самое главное, являлись несовместимыми. Таким образом, одна из самых серьезных проблем в программировании – переносимость программ при переводе программного обеспечения на другие компьютерные системы – проявлялась при разработке параллельных программ в самой максимальной степени. Как результат, уже с 90-х годов стали предприниматься усилия по стандартизации средств организации передачи сообщений в многопроцессорных вычислительных системах. Началом работ, непосредственно приведшей к появлению MPI, послужило проведение рабочего совещания по стандартам для передачи сообщений в среде распределенной памяти (the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992). По итогам совещания была образована рабочая группа, позднее преобразованная в международное сообщество MPI Forum, результатом деятельности которых явилось создание и принятие в 1994 г. стандарта *интерфейса передачи сообщений* (*message passing interface - MPI*) версии 1.0. В последующие годы стандарт MPI последовательно развивался. В 1997 г. был принят стандарт MPI версии 2.0.

Итак, теперь можно пояснить, что означает понятие MPI. Во-первых, MPI - это стандарт, которому должны удовлетворять средства организации передачи сообщений. Во-вторых, MPI – это программные средства, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI. Так, по стандарту, эти программные средства должны быть организованы в виде библиотек программных модулей (*библиотеки MPI*) и должны быть доступны для наиболее широко

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

используемых алгоритмических языков C и Fortran. Подобную "двойственность" MPI следует учитывать при использовании терминологии. Как правило, аббревиатура MPI используется для упоминания стандарта, а сочетание "библиотека MPI" указывает на ту или иную программную реализацию стандарта. Однако достаточно часто для краткости обозначение MPI используется и для библиотек MPI и, тем самым, для правильной интерпретации термина следует учитывать контекст.

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

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

4.1. MPI: основные понятия и определения

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

4.1.1. Понятие параллельной программы

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

Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (*модель SPMP*). Данный программный код, представленный в виде исполняемой программы, должен быть доступен в момент запуска параллельной программы на всех используемых процессорах. Исходный программный код для исполняемой программы разрабатывается на алгоритмических языках C или Fortran с использованием той или иной реализации библиотеки MPI.

Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов). Все процессы программы последовательно перенумерованы от 0 до $p-1$, где p есть общее количество процессов. Номер процесса именуется *рангом* процесса.

4.1.2. Операции передачи данных

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

Для выполнения парных операций могут использоваться разные *режимы передачи*, среди которых синхронный, блокирующий и др. – полное рассмотрение возможных режимов передачи будет выполнено в подразделе 4.3.

Как уже отмечалось ранее, стандарт MPI предусматривает необходимость реализации большинства основных коллективных операций передачи данных – см. подразделы 4.2 и 4.4.

4.1.3. Понятие коммуникаторов

Процессы параллельной программы объединяются в *группы*. Под *коммуникатором* в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных.

Как правило, парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов

коммуникатора. Как результат, указание используемого коммуникатора является обязательным для операций передачи данных в MPI.

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

При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (*intercommunicator*).

Подробное рассмотрение возможностей MPI для работы с группами и коммуникаторами будет выполнено в подразделе 4.6.

4.1.4. Типы данных

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

Подробное рассмотрение возможностей MPI для работы с производными типами данных будет выполнено в подразделе 4.5.

4.1.5. Виртуальные топологии

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

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

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

Кроме того, в MPI имеются средства и для формирования логических (виртуальных) топологий любого требуемого типа. Подробное рассмотрение возможностей MPI для работы с топологиями будет выполнено в подразделе 4.7.

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

- Описание функций и все приводимые примеры программ будут представлены на алгоритмическом языке C; особенности использования MPI для алгоритмического языка Fortran будут даны в п. 4.8.1,
- Краткая характеристика имеющихся реализаций библиотек MPI и общее описание среды выполнения MPI программ будут рассмотрены в п. 4.8.2,
- Основное изложение возможностей MPI будет ориентировано на стандарт версии 1.2 (*MPI-1*); дополнительные свойства стандарта версии 2.0 буду представлены в п. 4.8.3.

Приступая к изучению MPI, можно отметить, что, с одной стороны, MPI достаточно сложен – в стандарте MPI предусматривается наличие более 125 функций. С другой стороны, структура MPI является тщательно продуманной – разработка параллельных программ может быть начата уже после рассмотрения всего лишь 6 функций MPI. Все дополнительные возможности MPI могут осваиваться по мере роста сложности разрабатываемых алгоритмов и программ. Именное в таком стиле – от простого к сложному – и будет далее представлен весь учебный материал по MPI.

4.2. Введение в разработку параллельных программ с использованием MPI

4.2.1. Основы MPI

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

4.2.1.1 Инициализация и завершение MPI программ

Первой вызываемой функцией MPI должна быть функция:

```
int MPI_Init ( int *agrc, char ***argv );
```

для инициализации среды выполнения MPI-программы. Параметрами функции являются количество аргументов в командной строке и текст самой командной строки.

Последней вызываемой функцией MPI обязательно должна являться функция:

```
int MPI_Finalize (void);
```

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

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

Следует отметить:

1. Файл *mpi.h* содержит определения именованных констант, прототипов функций и типов данных библиотеки MPI,
2. Функции *MPI_Init* и *MPI_Finalize* являются обязательными и должны быть выполнены (и только один раз) каждым процессом параллельной программы,
3. Перед вызовом *MPI_Init* может быть использована функция *MPI_Initialized* для определения того, был ли ранее выполнен вызов *MPI_Init*.

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

4.2.1.2 Определение количества и ранга процессов

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

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

Для определения *ранга процесса* используется функция:

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

Как правило, вызов функций *MPI_Comm_size* и *MPI_Comm_rank* выполняется сразу после *MPI_Init*:

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

Следует отметить:

1. Коммуникатор *MPI_COMM_WORLD*, как отмечалось ранее, создается по умолчанию и представляет все процессы выполняемой параллельной программы,
2. Ранг, получаемый при помощи функции *MPI_Comm_rank*, является рангом процесса, выполнившего вызов этой функции, т.е. переменная *ProcRank* будет принимать различные значения в разных процессах.

4.2.1.3 Передача сообщений

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

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm),
где
 - buf   - адрес буфера памяти, в котором располагаются данные
           отправляемого сообщения,
 - count - количество элементов данных в сообщении,
 - type  - тип элементов данных пересылаемого сообщения,
 - dest  - ранг процесса, которому отправляется сообщение,
 - tag   - значение-тег, используемое для идентификации сообщений,
 - comm  - коммуникатор, в рамках которого выполняется передача данных.
```

Для указания типа пересылаемых данных в *MPI* имеется ряд базовых типов, полный список которых приведен в табл. 4.1.

Таблица 4.1. Базовые (предопределенные) типы данных MPI для алгоритмического языка C

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

Следует отметить:

1. Отправляемое сообщение определяется через указание блока памяти (*буфера*), в котором это сообщение располагается. Используемая для указания буфера триада

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

входит в состав параметров практически всех функций передачи данных,

2. Процессы, между которыми выполняется передача данных, в обязательном порядке должны принадлежать коммуникатору, указываемому в функции *MPI_Send*,

3. Параметр *tag* используется только при необходимости различения передаваемых сообщений, в противном случае в качестве значения параметра может быть использовано произвольное целое число (см. также описание функции *MPI_Recv*).

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

Пример использования функции будет представлен после описания функции *MPI_Recv*.

4.2.1.4 Прием сообщений

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

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

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

Следует отметить:

1. Буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения,
2. При необходимости приема сообщения от любого процесса-отправителя для параметра *source* может быть указано значение *MPI_ANY_SOURCE*,
3. При необходимости приема сообщения с любым тегом для параметра *tag* может быть указано значение *MPI_ANY_TAG*,
4. Параметр *status* позволяет определить ряд характеристик принятого сообщения:

```
  - status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения,
  - status.MPI_TAG    - тег принятого сообщения.
```

Функция

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

возвращает в переменной *count* количество элементов типа *type* в принятом сообщении.

Вызов функции *MPI_Recv* не должен согласовываться со временем вызова соответствующей функции передачи сообщения *MPI_Send* – прием сообщения может быть инициирован до момента, в момент или после момента начала отправки сообщения.

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

4.2.1.5 Первая параллельная программа с использованием MPI

Рассмотренный набор функций оказывается достаточным для разработки параллельных программ[2)]. Приводимая ниже программа является стандартным начальным примером для алгоритмического языка С.

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

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

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

Программа 4.1. Первая параллельная программа с использованием MPI

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

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

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

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

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

Следует отметить еще один важный момент – разрабатываемая с использованием MPI программа как в данном частном варианте, так и в самом общем случае используется для порождения всех процессов параллельной программы и, как результат, должна определять вычисления, выполняемые во всех этих процессах. Можно сказать, что MPI-программа является некоторым "*макро-кодом*", различные части которого используются разными процессами. Так, например, в приведенном примере программы выделенные двойной рамкой участки программного кода не выполняются одновременно ни в одном процессе. Первый выделенный участок с функцией приема *MPI_Send* исполняется только процессом с рангом 0, второй участок с функцией приема *MPI_Recv* используется всеми процессами, за исключением нулевого процесса.

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

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

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

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

В завершение обсуждения примера поясним использованный в MPI подход для контроля правильности выполнения функций - все функции MPI возвращают в качестве своего значения *код завершения*. При

успешном выполнении функции возвращаемый код равен *MPI_SUCCESS*. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций. Для выяснения типа обнаруженной ошибки используются предопределенные именованные константы, среди которых:

```
- MPI_ERR_BUFFER – неправильный указатель на буфер,
- MPI_ERR_COMM – неправильный коммуникатор,
- MPI_ERR_RANK – неправильный ранг процесса,
```

и др. – полный список констант для проверки кода завершения содержится в файле *mpi.h*.

4.2.2. Определение времени выполнение MPI-программы

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

Получение времени текущего момента выполнения программы обеспечивается при помощи функции:

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

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

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

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

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

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

4.2.3. Начальное знакомство с коллективными операциями передачи данных

Функции *MPI_Send* и *MPI_Recv*, рассмотренные в п. 4.2.1, обеспечивают возможность выполнения *парных операций* передачи данных между двумя процессами параллельной программы. Для выполнения коммуникационных *коллективных операций*, в которых принимают участие все процессы коммуникатора, в MPI предусмотрен специальный набор функций. В данном подразделе будут рассмотрены три такие функции, широко используемые даже при разработке сравнительно простых параллельных программ; полное же представление коллективных операций будет дано в подразделе 4.4.

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

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

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

4.2.3.1 Передача данных от одного процесса всем процессам программы

Первая проблема при выполнении рассмотренного параллельного алгоритма суммирования состоит в необходимости передачи значений вектора x всем процессам параллельной программы. Конечно, для

решения этой проблемы можно воспользоваться рассмотренными ранее функциями парных операций передачи данных:

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

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

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

```
int MPI_Bcast(void *buf,int count,MPI_Datatype type,int root,MPI_Comm comm),
где
  - buf, count, type – буфер памяти с отправляемым сообщением (для процесса
      с рангом 0), и для приема сообщений для всех остальных процессов,
  - root - ранг процесса, выполняющего рассылку данных,
  - comm - коммуникатор, в рамках которого выполняется передача данных.
```

Функция *MPI_Bcast* осуществляет рассылку данных из буфера *buf*, содержащего *count* элементов типа *type* с процесса, имеющего номер *root*, всем процессам, входящим в коммуникатор *comm* (см. рис. 4.1).

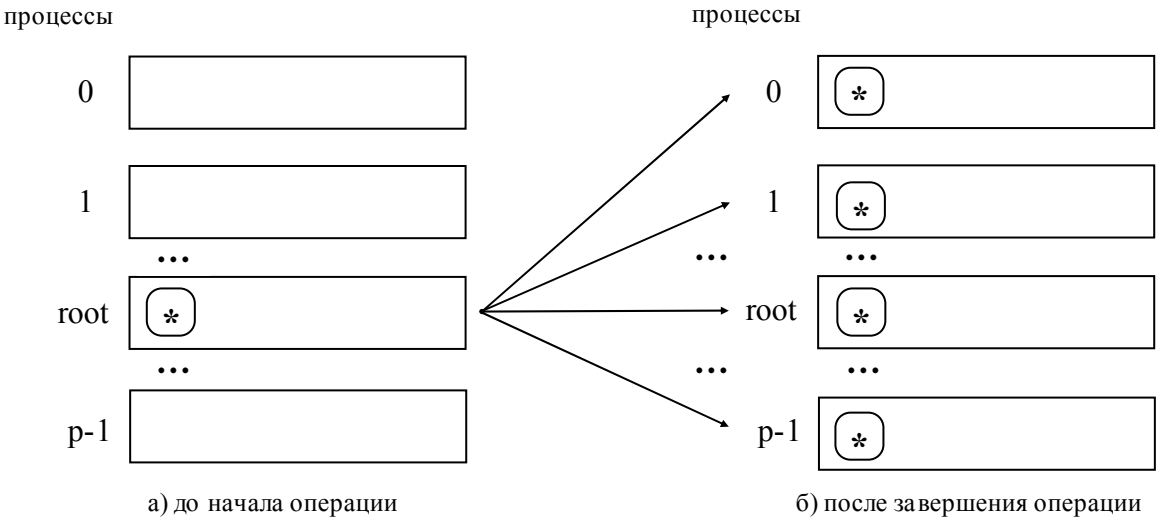


Рис. 4.1. Общая схема операции передачи данных от одного процесса всем процессам

Следует отметить:

1. Функция *MPI_Bcast* определяет коллективную операцию и, тем самым, при выполнении необходимых рассылок данных вызов функции *MPI_Bcast* должен быть осуществлен всеми процессами указываемого коммуникатора (см. далее пример программы),

2. Указываемый в функции *MPI_Bcast* буфер памяти имеет различное назначение в разных процессах. Для процесса с рангом *root*, с которого осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение. Для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных.

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

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

  // инициализация
  MPI_Init(&argc,&argv);
```

```
  MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);

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

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

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

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

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

Программа 4.2. Параллельная программа суммирования числовых значений

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

4.2.3.2 Передача данных от всех процессов одному процессу. Операции редукции

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

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

В качестве операций редукции данных могут быть использованы предопределенные в MPI операции – см. табл. 4.2.

Таблица 4.2. Базовые (предопределенные) типы операций MPI для функций редукции данных

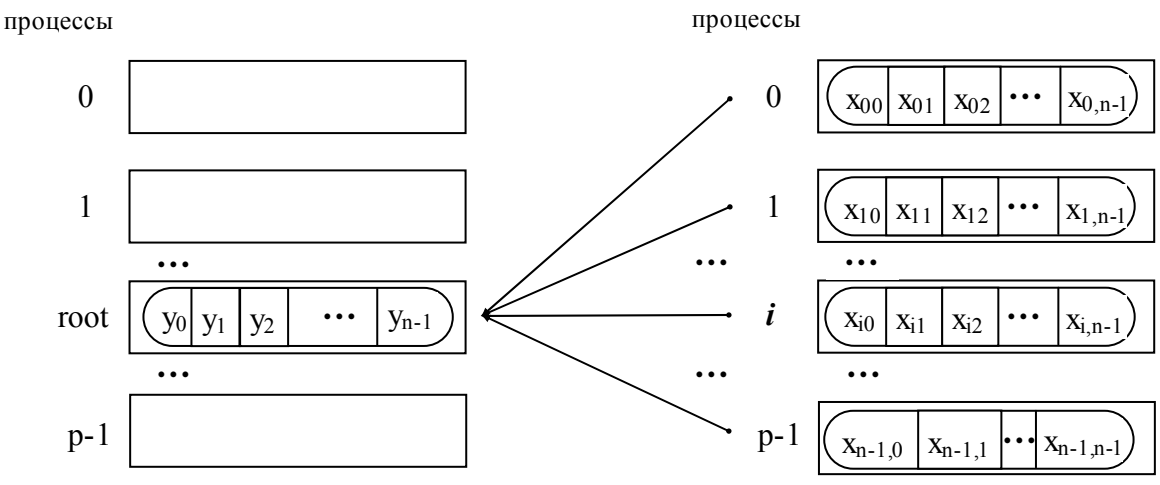
Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции "И" над значениями сообщений
MPI_BAND	Выполнение битовой операции "И" над значениями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

Помимо данного стандартного набора операций могут быть определены и новые дополнительные операции непосредственно самим пользователем библиотеки MPI – см., например, Немнюгин и Стесик (2002), Group, et al. (1994), Pacheco (1996).

Общая схема выполнения операции сбора и обработки данных на одном процессоре показана на рис. 4.2. Элементы получаемого сообщения на процессе *root* представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, т.е.

$$y_j = \overset{n-1}{\underset{i=0}{\otimes}} x_{ij},\ 0 \le j < n,$$

где $\otimes$ есть операция, задаваемая при вызове функции *MPI_Reduce* (для пояснения на рис. 4.3. показан пример выполнения операции редукции данных).



а) после завершения операции б) до начала операции

Рис. 4.2. Общая схема операции сбора и обработки на одном процессе данных от всех процессов

Следует отметить:

1. Функция *MPI_Reduce* определяет коллективную операцию и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора, все вызовы функции должны содержать одинаковые значения параметров *count*, *type*, *op*, *root*, *comm*,

2. Передача сообщений должна быть выполнена всеми процессами, результат операции будет получен только процессом с рангом *root*,

3. Выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования *MPI_SUM*, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно.

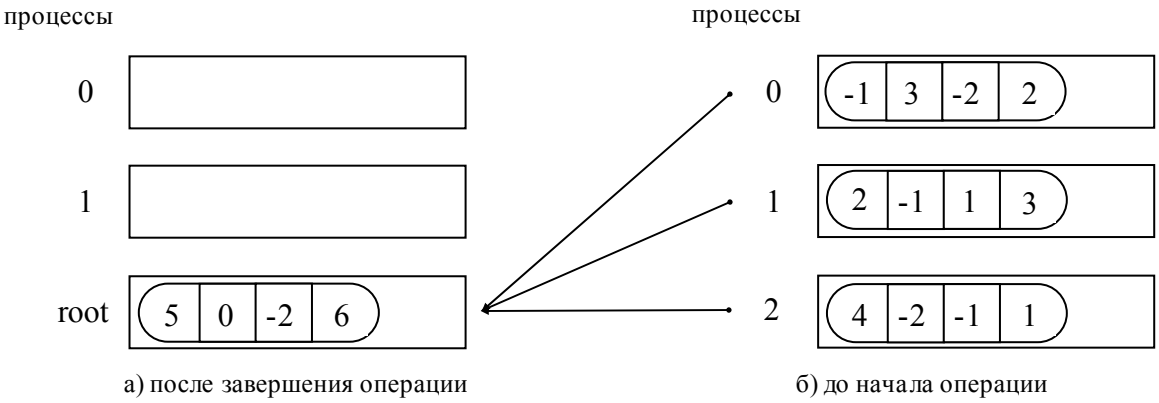


Рис. 4.3. Пример выполнения операции редукции при суммировании пересылаемых данных для трех процессов (в каждом сообщении 4 элемента, сообщения собираются на процессе с рангом 2)

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

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

4.2.3.3 Синхронизация вычислений

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

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

```
int MPI_Barrier(MPI_Comm comm);
```

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

4.3. Операции передачи данных между двумя процессами

Продолжим начатое в п. 4.2.1 изучение функций MPI для выполнения операций передачи данных между процессами параллельной программы.

4.3.1. Режимы передачи данных

Рассмотренная ранее функция *MPI_Send* обеспечивает так называемый *стандартный* (*Standard*) режим отправки сообщений, при котором (см. также п. 4.2.1.3):

- на время выполнения функции процесс-отправитель сообщения блокируется,

– после завершения функции буфер может быть использован повторно,

– состояние отправленного сообщения может быть различным - сообщение может располагаться в процессе-отправителе, может находиться в процессе передачи, может храниться в процессе-получателе или же может быть принято процессом-получателем при помощи функции *MPI_Recv*.

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

- *Синхронный* (*Synchronous*) *режим* состоит в том, что завершение функции отправки сообщения происходит только при получении от процесса-получателя подтверждения о начале приема отправленного сообщения, отправленное сообщение или полностью принято процессом-получателем или находится в состоянии приема,
- *Буферизованный* (*Buffered*) *режим* предполагает использование дополнительных системных буферов для копирования в них отправляемых сообщений; как результат, функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер,
- *Режим передачи по готовности* (*Ready*) может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

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

```
  - MPI_Ssend – функция отправки сообщения в синхронном режиме,
  - MPI_Bsend – функция отправки сообщения в буферизованном режиме,
  - MPI_Rsend – функция отправки сообщения в режиме по готовности.
```

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

Для использования буферизованного режима передачи должен быть создан и передан MPI буфер памяти для буферизации сообщений – используемая для этого функция имеет вид:

```
int MPI_Buffer_attach(void *buf, int size),
где
  - buf  - буфер памяти для буферизации сообщений,
  - size – размер буфера.
```

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

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

По практическому использованию режимов можно привести следующие рекомендации:

1. Режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема,
2. Стандартный и буферизованный режимы также выполняются достаточно быстро, но могут приводить к большим расходам ресурсов (памяти) – в целом может быть рекомендован для передачи коротких сообщений,
3. Синхронный режим является наиболее медленным, т.к. требует подтверждения приема. В тоже время, этот режим наиболее надежен – можно рекомендовать его для передачи длинных сообщений.

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

4.3.2. Организация неблокирующих обменов данными между процессорами

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

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

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

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

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

```
int MPI_Test( MPI_Request *request, int *flag, MPI_status *status),
где
  - request - дескриптор операции, определенный при вызове неблокирующей
              функции,
  - flag    - результат проверки (=true, если операция завершена),
  - status  - результат выполнения операции обмена (только для завершенной
              операции).
```

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

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

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

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

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

```
- MPI_Testall  - проверка завершения всех перечисленных операций обмена,
- MPI_Waitall  – ожидание завершения всех операций обмена,
- MPI_Testany  - проверка завершения хотя бы одной из перечисленных
                 операций обмена,
- MPI_Waitany  – ожидание завершения любой из перечисленных операций
                 обмена,
- MPI_Testsome - проверка завершения каждой из перечисленных операций
                 обмена,
- MPI_Waitsome - ожидание завершения хотя бы одной из перечисленных
                 операций обмена и оценка состояния по всем операциям.
```

Приведение простого примера использования неблокирующих функций достаточно затруднительно. Хорошей возможностью для освоения рассмотренных функций могут служить, например, параллельные алгоритмы матричного умножения (см. раздел 8).

4.3.3. Одновременное выполнение передачи и приема

Одной из часто выполняемых форм информационного взаимодействия в параллельных программах является обмен данными между процессами, когда для продолжения вычислений процессам необходимо

отправить данные одним процессам и, в то же время, получить сообщения от других процессов. Простейший вариант этой ситуации состоит, например, в обмене данными между двумя процессами. Реализация таких обменов при помощи обычных парных операций передачи данных неэффективна и достаточно трудоемка. Кроме того, такая реализация должна гарантировать отсутствие тупиковых ситуаций, которые могут возникать, например, когда два процесса начинают передавать сообщения друг другу с использованием блокирующих функций передачи данных.

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

```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype stype,int dest, int  stag,
                 void *rbuf,int rcount,MPI_Datatype rtype,int source,int rtag,
                 MPI_Comm comm, MPI_Status *status),
где
 - sbuf, scount, stype, dest,   stag - параметры передаваемого сообщения,
 - rbuf, rcount, rtype, source, rtag - параметры принимаемого сообщения,
 - comm   - коммуникатор, в рамках которого выполняется передача данных,
 - status – структура данных с информацией о результате выполнения операции.
```

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

В функции *MPI_Sendrecv* для передачи и приема сообщений применяются разные буфера. В случае же, когда сообщения имеют одинаковый тип, в MPI имеется возможность использования единого буфера:

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

Пример использование функций для одновременного выполнения операций передачи и приема приведен в разделе 8 при разработке параллельных программ матричного умножения.

4.4. Коллективные операции передачи данных

Как уже отмечалось ранее, под *коллективными операциями* в MPI понимаются операции над данными, в которых принимают участие все процессы используемого коммуникатора. Выделение основных видов коллективных операций было выполнено в разделе 3. Часть из коллективных операций уже была рассмотрена в п. 4.2.3 – это операции передачи от одного процесса всем процессам коммуникатора (*широковещательная рассылка*) и операции обработки данных, полученных на одном процессе от всех процессов (*редукция данных*).

Рассмотрим далее оставшиеся базовые коллективные операции передачи данных.

4.4.1. Обобщенная передача данных от одного процесса всем процессам

Обобщенная операция передачи данных от одного процесса всем процессам (*распределение данных*) отличается от широковещательной рассылки тем, что процесс передает процессам различающиеся данные (см. рис. 4.4). Выполнение данной операции может быть обеспечено при помощи функции:

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype stype,
                void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
где
 - sbuf, scount, stype - параметры передаваемого сообщения (scount определяет
      количество элементов, передаваемых на каждый процесс),
 - rbuf, rcount, rtype - параметры сообщения, принимаемого в процессах,
 - root – ранг процесса, выполняющего рассылку данных,
 - comm - коммуникатор, в рамках которого выполняется передача данных.
```

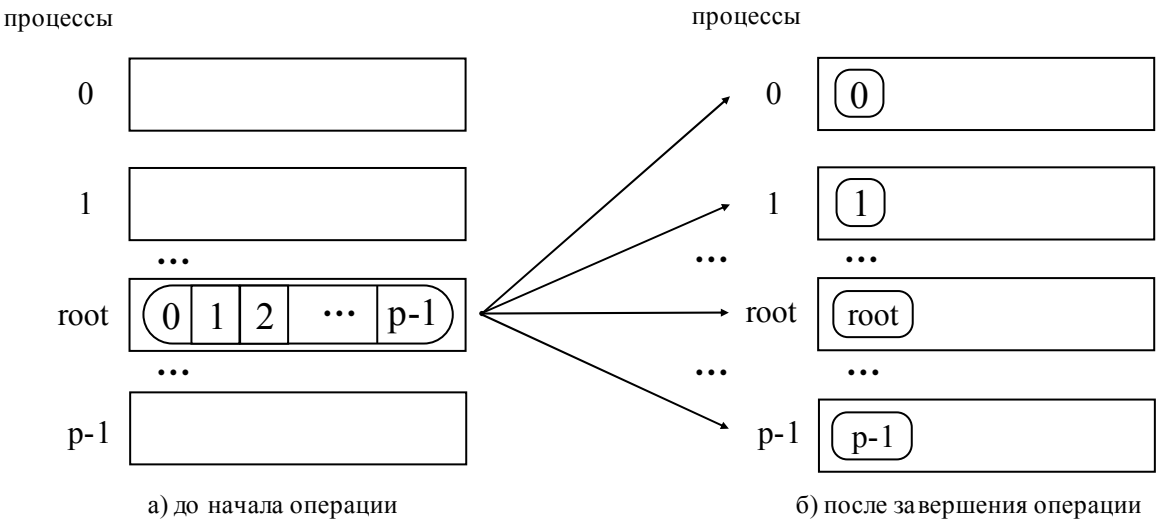


Рис. 4.4. Общая схема операции обобщенной передачи данных от одного процесса всем процессам

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

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

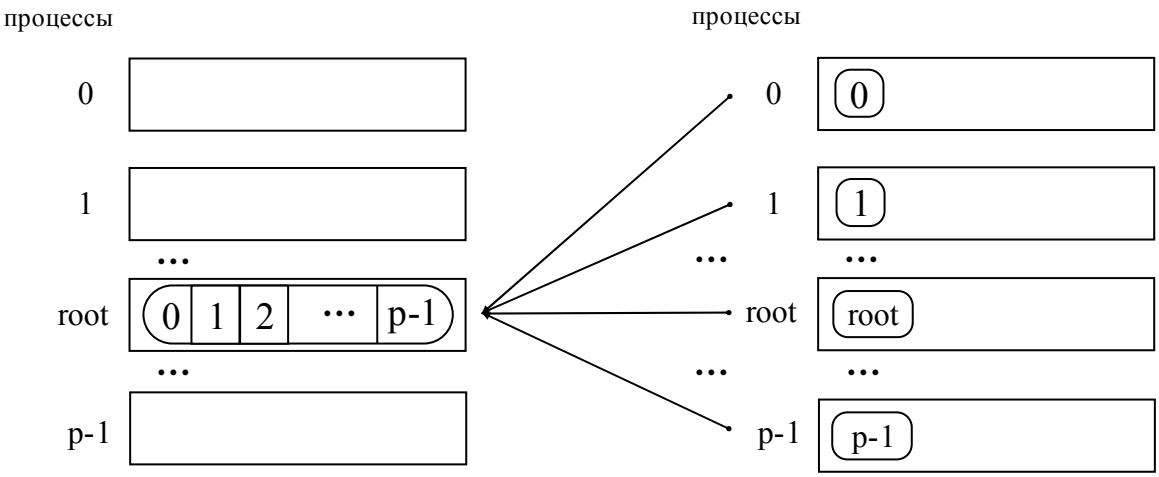
Отметим также, что функция *MPI_Scatter* передает всем процессам сообщения одинакового размера. Выполнение более общего варианта операции распределения данных, когда размеры сообщений для процессов могут быть разного размера, обеспечивается при помощи функции *MPI_Scatterv*.

Пример использования функции *MPI_Scatter* рассматривается в разделе 7 при разработке параллельных программ умножения матрицы на вектор.

4.4.2. Обобщенная передача данных от всех процессов одному процессу

Операция обобщенной передачи данных от всех процессоров одному процессу (*сбор данных*) является обратной к процедуре распределения данных (см. рис. 4.5). Для выполнения этой операции в MPI предназначена функция:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
               void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
где
 - sbuf, scount, stype - параметры передаваемого сообщения,
 - rbuf, rcount, rtype - параметры принимаемого сообщения,
 - root – ранг процесса, выполняющего сбор данных,
 - comm - коммуникатор, в рамках которого выполняется передача данных.
```



а) после завершения операции б) до начала операции

Рис. 4.5. Общая схема операции обобщенной передачи данных от всех процессов одному процессу

При выполнении функции *MPI_Gather* каждый процесс в коммуникаторе передает данные из буфера *sbuf* на процесс с рангом *root*. Процесс с рангом *root* собирает все получаемые данные в буфере *rbuf* (размещение данных в буфере осуществляется в соответствии с рангами процессов-отправителей сообщений). Для того, чтобы разместить все поступающие данные, размер буфера *rbuf* должен быть равен *scount* * *p* элементов, где *p* есть количество процессов в коммуникаторе *comm*.

Функция *MPI_Gather* также определяет коллективную операцию, и ее вызов при выполнении сбора данных должен быть обеспечен в каждом процессе коммуникатора.

Следует отметить, что при использовании функции *MPI_Gather* сборка данных осуществляется только на одном процессе. Для получения всех собираемых данных на каждом из процессов коммуникатора необходимо использовать *функцию сбора и рассылки*:

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

Выполнение общего варианта операции сбора данных, когда размеры передаваемых процессами сообщений могут быть различны, обеспечивается при помощи функций *MPI_Gatherv* и *MPI_Allgatherv*.

Пример использования функции *MPI_Gather* рассматривается в разделе 7 при разработке параллельных программ умножения матрицы на вектор.

4.4.3. Общая передача данных от всех процессов всем процессам

Передача данных от всех процессов всем процессам является наиболее общей операцией передачи данных (см. рис. 4.6). Выполнение данной операции может быть обеспечено при помощи функции:

```
int MPI_Alltoall(void *sbuf,int scount,MPI_Datatype stype,
                 void *rbuf,int rcount,MPI_Datatype rtype,MPI_Comm comm),
где
 - sbuf, scount, stype - параметры передаваемых сообщений,
 - rbuf, rcount, rtype - параметры принимаемых сообщений
 - comm - коммуникатор, в рамках которого выполняется передача данных.
```

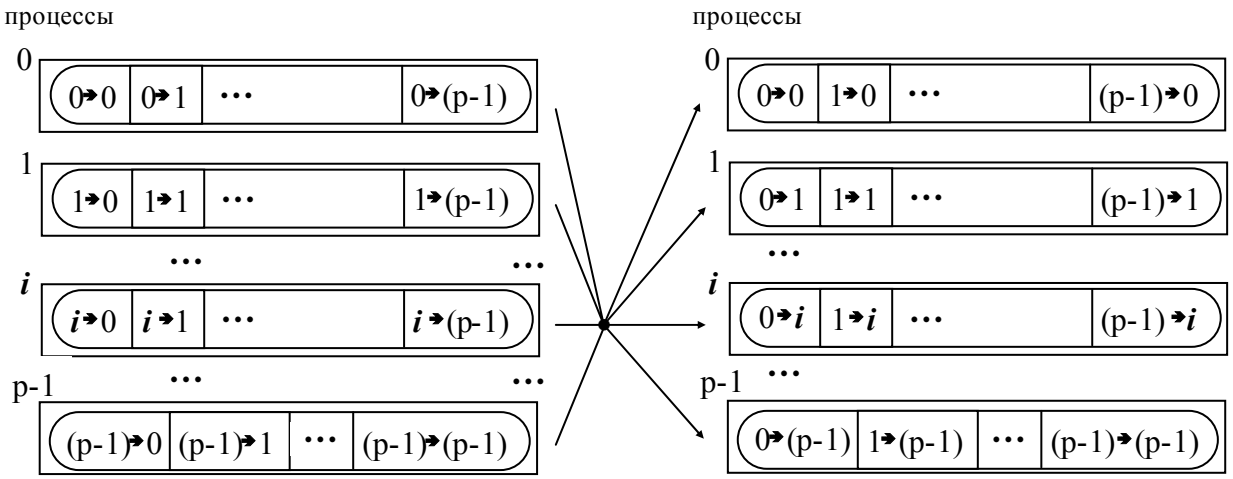


Рис. 4.6. Общая схема операции передачи данных от всех процессов всем процессам (сообщения показываются обозначениями вида ***i→j***, где *i* и *j* есть ранги передающих и принимающих процессов соответственно)

При выполнении функции *MPI_Alltoall* каждый процесс в коммуникаторе передает данные из *scount* элементов каждому процессу (общий размер отправляемых сообщений в процессах должен быть равен *scount* * *p* элементов, где *p* есть количество процессов в коммуникаторе *comm*) и принимает сообщения от каждого процесса.

Вызов функции *MPI_Alltoall* при выполнении операции общего обмена данными должен быть выполнен в каждом процессе коммуникатора.

Вариант операции общего обмена данных, когда размеры передаваемых процессами сообщений могут быть различны, обеспечивается при помощи функций *MPI_Alltoallv*.

Пример использования функции *MPI_Alltoall* рассматривается в разделе 7 при разработке параллельных программ умножения матрицы на вектор как задание для самостоятельного выполнения.

4.4.4. Дополнительные операции редукции данных

Рассмотренная в п. 4.2.3.2 функция *MPI_Reduce* обеспечивает получение результатов редукции данных только на одном процессе. Для получения результатов редукции данных на каждом из процессов коммуникатора необходимо использовать *функцию редукции и рассылки*:

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

Функция *MPI_AllReduce* выполняет рассылку между процессами всех результатов операции редукции. Возможность управления распределением этих данных между процессами предоставляется функций *MPI_Reduce_scatter*.

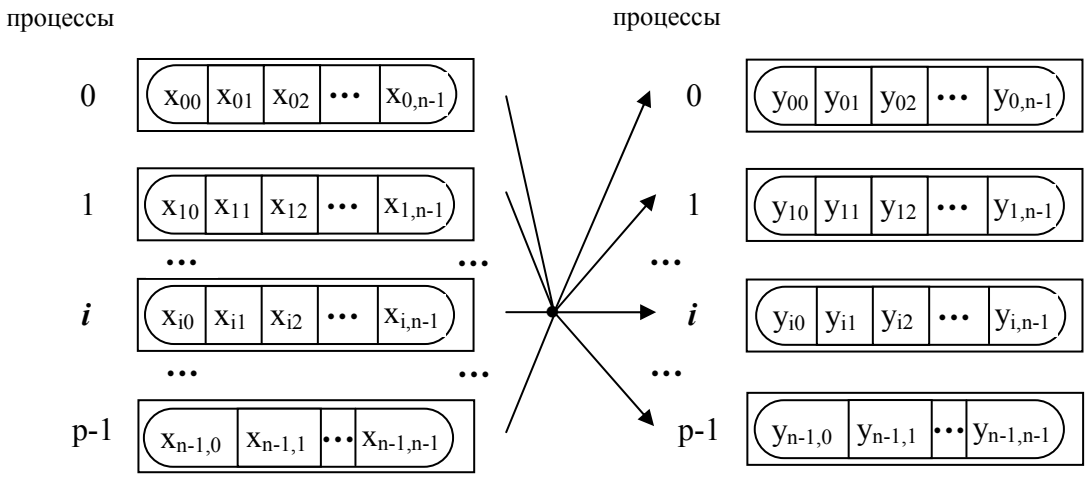
И еще один вариант операции сбора и обработки данных, при котором обеспечивается получение и всех частичных результатов редуцирования, может быть получен при помощи функции:

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

Общая схема выполнения функции *MPI_Scan* показана на рис. 4.7. Элементы получаемых сообщений представляют собой результаты обработки соответствующих элементов передаваемых процессами сообщений, при этом для получения результатов на процессе с рангом *i*, $0 \le i < n$, используются данные от процессов, ранг которых меньше или равен *i*,т.е.

$$y_{ij} = \overset{i}{\underset{k=0}{\otimes}} x_{kj},\ 0 \le i, j < n,$$

где $\otimes$ есть операция, задаваемая при вызове функции *MPI_Scan*.



а) до начала операции б) после завершения операции

Рис. 4.7. Общая схема операции редукции с получением частичных результатов обработки данных

4.4.5. Сводный перечень коллективных операций данных

Для удобства использования сводный перечень всего рассмотренного учебного материала о коллективных операциях передачи данных представлен в виде табл. 4.3.

Таблица 4.3. Сводный перечень учебного материала о коллективных операциях передачи данных

Вид коллективной операции	**Общее описание и оценка сложности**	**Функция MPI**	**Примеры использования**
Передача от одного процесса всем процессам (*широковещательная рассылка*)	п.3.2.5	**MPI_Bcast** п. 4.2.3.1	п. 4.2.3.1
Сбор и обработка данных на одном процессе от всех процессов (*редукция данных*)	пп.3.2.5, 3.2.6	**MPI_Reduce** п. 4.2.3.2	п. 4.2.3.2
- то же с рассылкой результатов всем процессам	пп.3.2.5, 3.2.6	**MPI_Allreduce** **MPI_Reduce_scatter** п. 4.4.4	
- то же с получением частичных результатов обработки	пп.3.2.5, 3.2.6	**MPI_Scan** п. 4.4.4	
Обобщенная передача от одного процесса всем процессам (*распределение данных*)	п.3.2.7	**MPI_Scatter** **MPI_Scatterv** п. 4.4.1	Раздел 7
Обобщенная передача от всех процессов одному процессу (*сбор данных*)	п.3.2.7	**MPI_Gather** **MPI_Gatherv** п. 4.4.2	Раздел 7
- то же с рассылкой результатов всем процессам	п.3.2.7	**MPI_Allgather** **MPI_Allgatherv** п. 4.4.2	
Общая передача данных от всех процессов всем процессам	п.3.2.8	**MPI_Alltoall** **MPI_Alltoallv** п. 4.4.3	Раздел 7

4.5. Производные типы данных в MPI

Во всех ранее рассмотренных примерах использования функций передачи данных предполагалось, что сообщения представляют собой некоторый непрерывный вектор элементов предусмотренного в MPI типа (список имеющихся в MPI типов представлен в табл. 4.1). Понятно, что в общем случае необходимые к пересылке данные могут рядом не располагаться и состоять из разного типа значений. Конечно, и в таких ситуациях разрозненные данные могут быть переданы с использованием нескольких сообщений, но такой способ решения не будет являться эффективным в силу накопления латентности множества выполняемых операций передачи данных. Другой возможный подход может состоять в предварительной упаковке передаваемых данных в формат того или иного непрерывного вектора, однако и здесь появляются лишние операции копирования данных, да и понятность таких операций передачи окажется далека от желаемой.

Для обеспечения бóльших возможностей при определении состава передаваемых сообщений в MPI предусмотрен механизм так называемых *производных типов данных*. Далее будут даны основные понятия используемого подхода, приведены возможные способы конструирования производных типов данных и рассмотрены функции упаковки и распаковки данных.

4.5.1. Понятие производного типа данных

В самом общем виде под *производным типом данных* в MPI можно понимать описание набора значений предусмотренного в MPI типа, причем в общем случае описываемые значения не обязательно непрерывно располагаются в памяти. Задание типа в MPI принято осуществлять при помощи *карты типа* (*type map*) в виде последовательности описаний входящих в тип значений, каждое отдельное значение описывается указанием типа и смещения адреса месторасположения от некоторого базового адреса, т.е.

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

Часть карты типа с указанием только типов значений именуется в MPI *сигнатурой типа*:

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

Сигнатура типа описывает, какие базовые типы данных образуют некоторый производный тип данных MPI и, тем самым, управляет интерпретацией элементов данных при передаче или получении сообщений. Смещения карты типа определяют, где находятся значения данных.

Поясним рассмотренные понятия на следующем примере. Пусть в сообщение должны входить значения переменных:

```
double a; /* адрес 24 */
double b; /* адрес 40 */
int    n; /* адрес 48 */
```

Тогда производный тип для описания таких данных должен иметь карту типа следующего вида:

```
{(MPI_DOUBLE,0),
 (MPI_DOUBLE,16),
 (MPI_INT,24)
}
```

Дополнительно для производных типов данных в MPI используется следующий ряд новых понятий:

- *нижняя граница* типа

$$lb(\mathbf{TypeMap}) = \min_j(disp_j),$$

- *верхняя граница* типа

$$ub(\mathbf{TypeMap}) = \max_j(disp_j + sizeof(type_j)) + \Delta,$$

- *протяженность* типа

$$extent(\mathbf{TypeMap}) = ub(\mathbf{TypeMap}) - lb(\mathbf{TypeMap}).$$

Согласно определению *нижняя граница* есть смещение для первого байта значений рассматриваемого типа данных. Соответственно *верхняя граница* представляет собой смещение для байта, располагающегося вслед за последним элементом рассматриваемого типа данных. При этом величина смещения для верхней границы может быть округлена вверх с учетом требований выравнивания адресов. Так, одно из самых общих требований, которые налагают реализации языков C и Fortran, состоит в том, чтобы адрес элемента был кратен длине этого элемента в байтах. Например, если тип *int* занимает четыре байта, то адрес на элемент типа *int* должен нацело делиться на четыре. Именно это требование и отражается в определении верхней границы типа данных MPI. Поясним данный момент на ранее рассмотренном примере набора переменных *a,b* и *n*, для которого нижняя граница равна 0, а верхняя граница принимает значение 32 (величина округления 6 или 4 в зависимости от размера типа *int*). Здесь следует отметить, что требуемое выравнивание определяется по типу первого элемента данных в карте типа.

Следует также указать на различие понятий протяженности и размера типа. Протяженность – это размер памяти в байтах, который нужно отводить для одного элемента производного типа. *Размер типа* данных - это число байтов, которые занимают данные (разность между адресами последнего и первого байтов данных). Различие в значениях протяженности и размера опять же в величине округления для выравнивания адресов. Так, в рассматриваемом примере размер типа равен 28, а протяженность – 32 (предполагается, что тип *int* занимает четыре байта).

Для получения значения протяженности и размера типа в MPI предусмотрены функции:

```
int MPI_Type_extent ( MPI_Datatype type, MPI_Aint *extent ),
int MPI_Type_size   ( MPI_Datatype type, MPI_Aint *size ).
```

Определение нижней и верхней границ типа может быть выполнено при помощи функций:

```
int MPI_Type_lb ( MPI_Datatype type, MPI_Aint *disp ),
int MPI_Type_ub ( MPI_Datatype type, MPI_Aint *disp ).
```

Важной и необходимой при конструировании производных типов является функция получения адреса переменной:

```
int MPI_Address ( void *location, MPI_Aint *address )
```

(следует отметить, что данная функция является переносимым вариантом средств получения адресов в алгоритмических языках C и Fortran).

4.5.2. Способы конструирования производных типов данных

Для снижения сложности в MPI предусмотрено несколько различных способов конструирования производных типов:

- ***Непрерывный*** способ позволяет определить непрерывный набор элементов существующего типа как новый производный тип,
- ***Векторный*** способ обеспечивает создание нового производного типа как набора элементов существующего типа, между элементами которого существуют регулярные промежутки по памяти. При этом, размер промежутков задается в числе элементов исходного типа, в то время, как в варианте ***H-векторного*** способа этот размер указывается в байтах,
- ***Индексный*** способ отличается от векторного метода тем, что промежутки между элементами исходного типа могут иметь нерегулярный характер,
- ***Структурный*** способ обеспечивает самое общее описание производного типа через явное указание карты создаваемого типа данных.

Далее перечисленные способы конструирования производных типов данных будут рассмотрены более подробно.

4.5.2.1 Непрерывный способ конструирования

При непрерывном способе конструирования производного типа данных в MPI используется функция:

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

Как следует из описания, новый тип *newtype* создается как *count* элементов исходного типа *oldtype*. Например, если исходный тип данных имеет карту типа

```
{ (MPI_INT,0),(MPI_DOUBLE,8) },
```

то вызов функции *MPI_Type_contiguous* с параметрами

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

приведет к созданию типа данных с картой типа

```
{ (MPI_INT,0),(MPI_DOUBLE,8),(MPI_INT,16),(MPI_DOUBLE,24) }.
```

В определенном плане наличие непрерывного способа конструирования является избыточным, поскольку использование аргумента *count* в процедурах MPI равносильно использованию непрерывного типа данных такого же размера.

4.5.2.2 Векторный способ конструирования

При векторном способе конструирования производного типа данных в MPI используются функции

```
int MPI_Type_vector ( int count, int blocklen, int stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
где
 - count    – количество блоков,
```

```
 - blocklen – размер каждого блока,
 - stride   – количество элементов, расположенных между двумя соседними
              блоками
 - oldtype  - исходный тип данных,
 - newtype  - новый определяемый тип данных.
```

```
int MPI_Type_hvector ( int count, int blocklen, MPI_Aint stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ).
```

Отличие способа конструирования, определяемого функцией *MPI_Type_hvector*, состоит лишь в том, что параметр *stride* для определения интервала между блоками задается в байтах, а не в элементах исходного типа данных.

Как следует из описания, при векторном способе новый производный тип создается как набор блоков из элементов исходного типа, при этом между блоками могут иметься регулярные промежутки по памяти. Приведем несколько примеров использования данного способа конструирования типов:

- Конструирование типа для выделения половины (только четных или только нечетных) строк матрицы размером *nxn*:

```
MPI_Type_vector ( n/2, n, 2*n, &StripRowType, &ElemType ),
```

- Конструирование типа для выделения столбца матрицы размером *nxn*:

```
MPI_Type_vector ( n, 1, n, &ColumnType, &ElemType ),
```

- Конструирование типа для выделения главной диагонали матрицы размером *nxn*:

```
MPI_Type_vector ( n, 1, n+1, &DiagonalType, &ElemType ).
```

С учетом характера приводимых примеров можно упомянуть имеющуюся в MPI возможность создания производных типов для описания подмассивов многомерных массивов при помощи функции (данная функция предусматривается стандартом MPI-2):

```
int MPI_Type_create_subarray ( int ndims, int *sizes, int *subsizes,
  int *starts, int order, MPI_Data_type oldtype, MPI_Datatype *newtype ),
где
 - ndims    – размерность массива,
 - sizes    – количество элементов в каждой размерности исходного массива,
 - subsizes – количество элементов в каждой размерности определяемого
              подмассива,
 - starts   – индексы начальных элементов в каждой размерности
              определяемого подмассива,
 - order    – параметр для указания необходимости переупорядочения,
 - oldtype  - тип данных элементов исходного массива,
 - newtype  - новый тип данных для описания подмассива.
```

4.5.2.3 Индексный способ конструирования

При индексном способе конструирования производного типа данных в MPI используются функции:

```
int MPI_Type_indexed ( int count, int blocklens[], int indices[],
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
где
 - count     – количество блоков,
 - blocklens – количество элементов в каждов блоке,
 - indices   – смещение каждого блока от начала типа (в количестве
               элементов исходного типа),
 - oldtype   - исходный тип данных,
 - newtype   - новый определяемый тип данных.
```

```
int MPI_Type_hindexed ( int count, int blocklens[], MPI_Aint indices[],
  MPI_Data_type oldtype, MPI_Datatype *newtype )
```

Как следует из описания, при индексном способе новый производный тип создается как набор блоков разного размера из элементов исходного типа, при этом между блоками могут иметься разные промежутки по памяти. Для пояснения данного способа можно привести пример конструирования типа для описания верхней треугольной матрицы размером *nxn*:

```
// конструирование типа для описания верхней треугольной матрицы
for ( i=0, i<n; i++ ) {
  blocklens[i] = n - i;
```

```
  indices[i]   = i * n + i;
}
MPI_Type_indexed ( n, blocklens, indices, &UTMatrixType, &ElemType ).
```

Как и ранее, способ конструирования, определяемый функцией *MPI_Type_hindexed*, отличается тем, что элементы *indices* для определения интервалов между блоками задаются в байтах, а не в элементах исходного типа данных.

Следует отметить, что существует еще одна дополнительная функция *MPI_Type_create_indexed_block* индексного способа конструирования для определения типов с блоками одинакового размера (данная функция предусматривается стандартом MPI-2).

4.5.2.4 Структурный способ конструирования

Как отмечалось ранее, структурный способ является самым общим методом конструирования производного типа данных при явном задании соответствующей карты типа. Использование такого способа производится при помощи функции:

```
int MPI_Type_struct ( int count, int blocklens[], MPI_Aint indices[],
  MPI_Data_type oldtypes[], MPI_Datatype *newtype ),
где
 - count      – количество блоков,
 - blocklens  – количество элементов в каждов блоке,
 - indices    – смещение каждого блока от начала типа (в байтах),
 - oldtypes   - исходные типы данных в каждом блоке в отдельности,
 - newtype    - новый определяемый тип данных.
```

Как следует из описания, структурный способ дополнительно к индексному методу позволяет указывать типы элементов для каждого блока в отдельности.

4.5.3. Объявление производных типов и их удаление

Рассмотренные в предыдущем пункте функции конструирования позволяют определить производный тип данных. Дополнительно перед использованием созданный тип *должен быть объявлен* при помощи функции:

```
int MPI_Type_commit (MPI_Datatype *type ).
```

При завершении использования производный тип должен быть аннулирован при помощи функции:

```
int MPI_Type_free (MPI_Datatype *type ).
```

4.5.4. Формирование сообщений при помощи упаковки и распаковки данных

Наряду с рассмотренными в п. 4.5.2 методами конструирования производных типов в MPI предусмотрен и явный способ сборки и разборки сообщений, содержащих значения разных типов и располагаемых в разных областях памяти.

Для использования данного подхода должен быть определен буфер памяти достаточного размера для сборки сообщения. Входящие в состав сообщения данные должны быть *упакованы* в буфер при помощи функции:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type,
  void *buf, int bufsize, int *bufpos, MPI_Comm comm),
где
 - data   – буфер памяти с элементами для упаковки,
 - count  – количество элементов в буфере,
 - type   – тип данных для упаковываемых элементов,
 - buf    - буфер памяти для упаковки,
 - buflen – размер буфера в байтах,
 - bufpos – позиция для начала записи в буфер (в байтах от начала буфера),
 - comm   - коммуникатор для упакованного сообщения.
```

Функция *MPI_Pack* упаковывает *count* элементов из буфера *data* в буфер упаковки *buf*, начиная с позиции *bufpos*. Общая схема процедуры упаковки показана на рис. 4.8а.

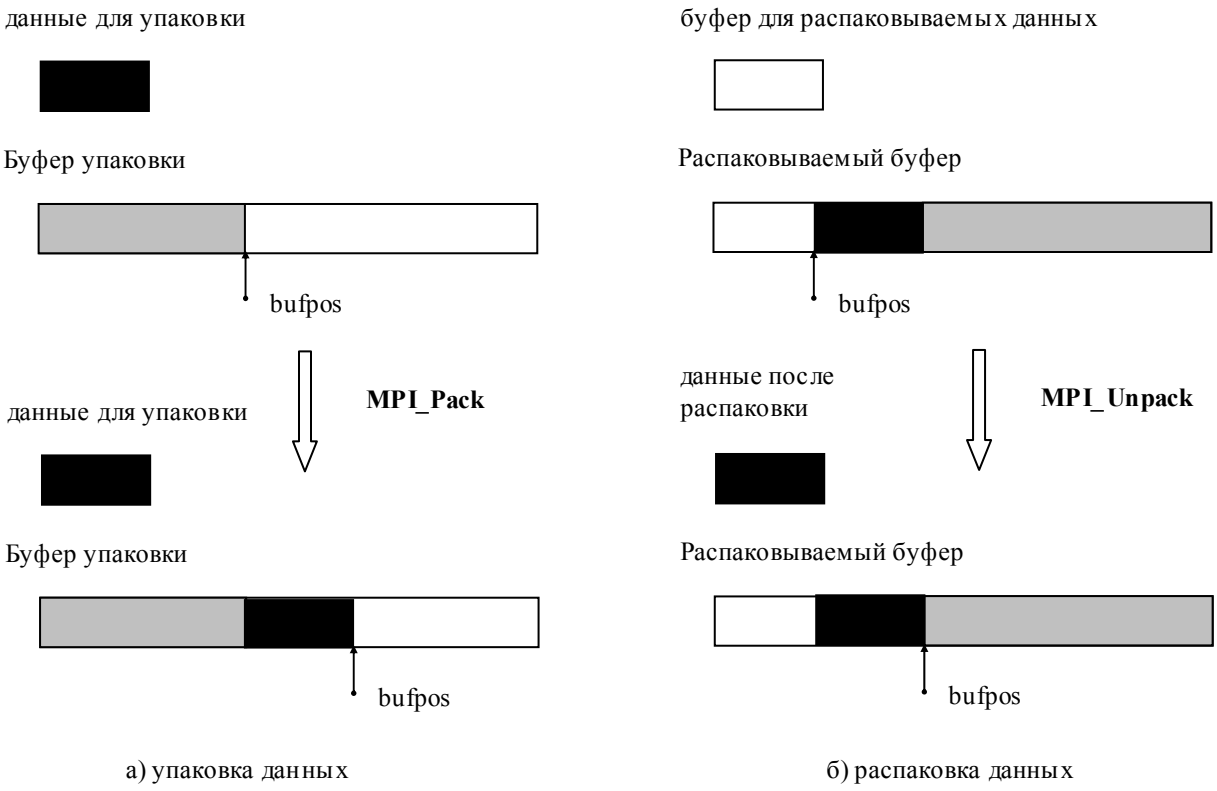


Рис. 4.8. Общая схема упаковки и распаковки данных

Начальное значение переменной *bufpos* должно быть сформировано до начала упаковки и далее устанавливается функцией *MPI_Pack*. Вызов функции *MPI_Pack* осуществляется последовательно для упаковки всех необходимых данных. Так, для ранее рассмотренного примера набора переменных *a,b* и *n*, для их упаковки необходимо выполнить:

```
bufpos = 0;
MPI_Pack(a,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(b,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(n,1,MPI_INT,buf,buflen,&bufpos,comm);
```

Для определения необходимого размера буфера для упаковки может быть использована функция:

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

которая в параметре *size* указывает необходимый размер буфера для упаковки *count* элементов типа *type*.

После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа *MPI_PACKED*.

После получения сообщения с типом *MPI_PACKED* данные могут быть распакованы при помощи функции:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos,
  void *data, int count, MPI_Datatype type, MPI_Comm comm),
где
 - buf    - буфер памяти с упакованными данными,
 - buflen - размер буфера в байтах,
 - bufpos - позиция начала данных в буфере (в байтах от начала буфера),
 - data   - буфер памяти для распаковываемых данных,
 - count  - количество элементов в буфере,
 - type   - тип распаковываемых данных,
 - comm   - коммуникатор для упакованного сообщения.
```

Функция *MPI_Unpack* распаковывает начиная с позиции *bufpos* очередную порцию данных из буфера *buf* и помещает распакованные данные в буфер *data*. Общая схема процедуры распаковки показана на рис. 4.8б.

Начальное значение переменной *bufpos* должно быть сформировано до начала распаковки и далее устанавливается функцией *MPI_Unpack*. Вызов функции *MPI_Unpack* осуществляется последовательно для распаковки всех упакованных данных, при этом порядок распаковки должен соответствовать порядку

упаковки. Так, для ранее рассмотренного примера упаковки для распаковки упакованных данных необходимо выполнить:

```
bufpos = 0;
MPI_Pack(buf,buflen,&bufpos,a,1,MPI_DOUBLE,comm);
MPI_Pack(buf,buflen,&bufpos,b,1,MPI_DOUBLE,comm);
MPI_Pack(buf,buflen,&bufpos,n,1,MPI_INT,comm);
```

В заключение выскажем ряд рекомендаций по использованию упаковки для формирования сообщений. Поскольку такой подход приводит к появлению дополнительных действий по упаковке и распаковке данных, то данный способ может быть оправдан при сравнительно небольших размерах сообщений и при малом количестве повторений. Упаковка и распаковка может оказаться полезной при явном использовании буферов для буферизованного способа передачи данных.

4.6. Управление группами процессов и коммуникаторами

Рассмотрим теперь возможности MPI по управлению группами процессов и коммуникаторами.

Для изложения последующего материала напомним ряд понятий и определений, приведенных в начале данного раздела.

Процессы параллельной программы объединяются в *группы*. В группу могут входить все процессы параллельной программы; с другой стороны, в группе может находиться только часть имеющихся процессов. Соответственно, один и тот же процесс может принадлежать нескольким группам. Управление группами процессов предпринимается для создания на их основе коммуникаторов.

Под *коммуникатором* в MPI понимается специально создаваемый служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных. Как правило, парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов коммуникатора. Создание коммуникаторов предпринимается для уменьшения области действия коллективных операций и для устранения взаимовлияния разных выполняемых частей параллельной программы. Важно еще раз подчеркнуть – коммуникационные операции, выполняемые с использованием разных коммуникаторов, являются независимыми и не влияют друг на друга.

Все имеющиеся в параллельной программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (*intercommunicator*). Взаимодействие между процессами разных групп оказывается необходимым в достаточно редких ситуациях, в данном учебном материале не рассматривается и может служить темой для самостоятельно изучения – см., например, Немнюгин и Стесик (2002), Group, et al. (1994), Pacheco (1996).

4.6.1. Управление группами

Группы процессов могут быть созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, связанная с предопределенным коммуникатором MPI_COMM_WORLD.

Для получения группы, связанной с существующим коммуникатором, используется функция:

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

Далее, на основе существующих групп, могут быть созданы новые группы:

- создание новой группы *newgroup* из существующей группы *oldgroup*, которая будет включать в себя *n* процессов, ранги которых перечисляются в массиве *ranks*:

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

- создание новой группы *newgroup* из группы *oldgroup*, которая будет включать в себя *n* процессов, ранги которых не совпадают с рангами, перечисленными в массиве *ranks*:

```
int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks,MPI_Group *newgroup).
```

Для получения новых групп над имеющимися группами процессов могут быть выполнены операции объединения, пересечения и разности:

- создание новой группы *newgroup* как объединения групп *group1* и *group2*:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

- создание новой группы *newgroup* как пересечения групп *group1* и *group2*:

```
int MPI_Group_intersection ( MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup ),
```

- создание новой группы *newgroup* как разности групп *group1* и *group2*:

```
int MPI_Group_difference ( MPI_Group group1, MPI_Group group2,
  MPI_Group *newgroup ).
```

При конструировании групп может оказаться полезной специальная пустая группа MPI_COMM_EMPTY.

Ряд функций MPI обеспечивает получение информации о группе процессов:

- получение количества процессов в группе:

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

- получение ранга текущего процесса в группе:

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

После завершения использования группа должна быть удалена:

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

(выполнение данной операции не затрагивает коммуникаторы, в которых используется удаляемая группа).

4.6.2. Управление коммуникаторами

Отметим прежде всего, что в данном пункте рассматривается управление *интракоммуникаторами*, используемыми для операций передачи данных внутри одной группы процессов. Как отмечалось ранее, применение *интеркоммуникаторов* для обменов между группами процессов выходит за пределы данного учебного материала.

Для создания новых коммуникаторов применимы два основных способа их получения:

- дублирование уже существующего коммуникатора:

```
int MPI_Comm_dup ( MPI_Comm oldcom, MPI_comm *newcomm ),
```

- создание нового коммуникатора из подмножества процессов существующего коммуникатора:

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group, MPI_Comm *newcomm).
```

Дублирование коммуникатора может использоваться, например, для устранения возможности пересечения по тегам сообщений в разных частях параллельной программы (в т.ч. и при использовании функций разных программных библиотек).

Следует отметить также, что операция создания коммуникаторов является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Для пояснения рассмотренных функций можно привести пример создания коммуникатора, в котором содержатся все процессы, кроме процесса, имеющего ранг 0 в коммуникаторе MPI_COMM_WORLD (такой коммуникатор может быть полезен для поддержки схемы организации параллельных вычислений "менеджер - исполнители" – см. раздел 6):

```
MPI_Group WorldGroup, WorkerGroup;
MPI_Comm Workers;
int ranks[1];
ranks[0] = 0;
// получение группы процессов в MPI_COMM_WORLD
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
// создание группы без процесса с рангом 0
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);
// Создание коммуникатора по группе
MPI_Comm_create(MPI_COMM_WORLD,WorkerGroup,&Workers);
...
MPI_Group_free(&WorkerGroup);
MPI_Comm_free(&Workers);
```

Быстрый и полезный способ одновременного создания нескольких коммуникаторов обеспечивает функция:

```
int MPI_Comm_split ( MPI_Comm oldcomm, int split, int key,
  MPI_Comm *newcomm ),
где
 - oldcomm – исходный коммуникатор,
```

```
- split   – номер коммуникатора, которому должен принадлежать процесс,
- key     – порядок ранга процесса в создаваемом коммуникаторе,
- newcomm – создаваемый коммуникатор.
```

Создание коммуникаторов относится к коллективным операциям и, тем самым, вызов функции *MPI_Comm_split* должен быть выполнен в каждом процессе коммуникатора *oldcomm*. В результате выполнения функции процессы разделяются на непересекающиеся группы с одинаковыми значениями параметра *split*. На основе сформированных групп создается набор коммуникаторов. При создании коммуникаторов для рангов процессов выбирается такой порядок нумерации, чтобы он соответствовал порядку значений параметров *key* (процесс с большим значением параметра *key* должен иметь больший ранг).

В качестве примера можно рассмотреть задачу представления набора процессов в виде двумерной решетки. Пусть $p=q*q$ есть общее количество процессов, следующий далее фрагмент программы обеспечивает получение коммуникаторов для каждой строки создаваемой топологии:

```
MPI_Comm comm;
int rank, row;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
row = rank/q;
MPI_Comm_split(MPI_COMM_WORLD,row,rank,&comm);
```

При выполнении данного примера, например, при $p=9$, процессы с рангами (0,1,2) образуют первый коммуникатор, процессы с рангами (3,4,5) – второй и т.д.

После завершения использования коммуникатор должен быть удален:

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

4.7. Виртуальные топологии

Под *топологией* вычислительной системы обычно понимается структура узлов сети и линий связи между этими узлами. Топология может быть представлена в виде графа, в котором вершины есть процессоры (процессы) системы, а дуги соответствуют имеющимся линиям (каналам) связи.

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

Понятно, что физическая топология системы является аппаратно реализуемой и изменению не подлежит (хотя существуют и программируемые средства построения сетей). Но, оставляя неизменной физическую основу, мы можем организовать логическое представление любой необходимой *виртуальной топологии*. Для этого достаточно, например, сформировать тот или иной механизм дополнительной адресации процессов.

Использование виртуальных процессов может оказаться полезным в силу ряда разных причин. Виртуальная топология, например, может больше соответствовать имеющейся структуре линий передачи данных. Использование виртуальных топологий может заметно упростить в ряде случаев представление и реализацию параллельных алгоритмов.

В MPI поддерживаются два вида топологий - *прямоугольная решетка* произвольной размерности (*декартова топология*) и топология *графа* любого произвольного вида. Следует отметить, что имеющиеся в MPI функции обеспечивают лишь получение новых логических систем адресации процессов, соответствующих формируемым виртуальным топологиям. Выполнение же всех коммуникационных операций должно осуществляться, как и ранее, при помощи обычных функций передачи данных с использованием исходных рангов процессов.

4.7.1. Декартовы топологии (решетки)

Декартовы топологии, в которых множество процессов представляется в виде прямоугольной *решетки* (см. п. 1.4.1 и рис. 1.7), а для указания процессов используется декартова система координат, широко применяются во многих задачах для описания структуры имеющихся информационных зависимостей. В числе примеров таких задач – матричные алгоритмы (см. разделы 7 и 8) и сеточные методы решения дифференциальных уравнений в частных производных (см. раздел 12).

Для создания декартовой топологии (решетки) в MPI предназначена функция:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims, int *periods,
  int reorder, MPI_Comm *cartcomm),
```

```
где:
 - oldcomm  - исходный коммуникатор,
 - ndims    - размерность декартовой решетки,
 - dims     - массив длины ndims, задает количество процессов в каждом
              измерении решетки,
 - periods  - массив длины ndims, определяет, является ли решетка
              периодической вдоль каждого измерения,
 - reorder  - параметр допустимости изменения нумерации процессов,
 - cartcomm - создаваемый коммуникатор с декартовой топологией процессов.
```

Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

Для пояснения назначения параметров функции *MPI_Cart_create* рассмотрим пример создания двухмерной решетки *4x4*, в которой строки и столбцы имеют кольцевую структуру (за последним процессом следует первый процесс):

```
// создание двухмерной решетки 4x4
MPI_Comm GridComm;
int dims[2], periods[2], reorder = 1;
dims[0]    = dims[1]    = 4;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reoreder, &GridComm);
```

Следует отметить, что в силу кольцевой структуры измерений сформированная в рамках примера топология является *тором*.

Для определения декартовых координат процесса по его рангу можно воспользоваться функцией:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int *coords),
где:
 - comm   - коммуникатор с топологией решетки,
 - rank   - ранг процесса, для которого определяются декартовы координаты,
 - ndims  - размерность решетки,
 - coords - возвращаемые функцией декартовы координаты процесса.
```

Обратное действие – определение ранга процесса по его декартовым координатам – обеспечивается при помощи функции:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
где
 - comm   - коммуникатор с топологией решетки,
 - coords - декартовы координаты процесса,
 - rank   - возвращаемый функцией ранг процесса.
```

Полезная во многих приложениях процедура разбиения решетки на подрешетки меньшей размерности обеспечивается при помощи функции:

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
где:
 - comm    - исходный коммуникатор с топологией решетки,
 - subdims - массив для указания, какие измерения должны остаться
             в создаваемой подрешетке,
 - newcomm - создаваемый коммуникатор с подрешеткой.
```

Операция создания подрешеток также является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора. В ходе своего выполнения функция *MPI_Cart_sub* определяет коммуникаторы для каждого сочетания координат фиксированных измерений исходной решетки.

Для пояснения функции *MPI_Cart_sub* дополним ранее рассмотренный пример создания двухмерной решетки и определим коммуникаторы с декартовой топологией для каждой строки и столбца решетки в отдельности:

```
// создание коммуникаторов для каждой строки и столбца решетки
MPI_Comm RowComm, ColComm;
int subdims[2];
// создание коммуникаторов для строк
subdims[0] = 0;  // фиксации измерения
subdims[1] = 1;  // наличие данного измерения в подрешетке
MPI_Cart_sub(GridComm, subdims, &RowComm);
```

```
// создание коммуникаторов для столбцов
subdims[0] = 1;
subdims[1] = 0;
MPI_Cart_sub(GridComm, subdims, &ColComm);
```

В приведенном примере для решетки размером 4x4 создаются 8 коммуникаторов, по одному для каждой строки и столбца решетки. Для каждого процесса определяемые коммуникаторы *RowComm* и *ColComm* соответствуют строке и столбцу процессов, к которым данный процесс принадлежит.

Дополнительная функция *MPI_Cart_shift* обеспечивает поддержку процедуры последовательной передачи данных по одному из измерений решетки (*операция сдвига данных* - см. раздел 3). В зависимости от периодичности измерения решетки, по которому выполняется сдвиг, различаются два типа данной операции:

– *Циклический сдвиг* на *k* элементов вдоль измерения решетки – в этой операции данные от процесса *i* пересылаются процессу *(i+k) mod dim*, где *dim* есть размер измерения, вдоль которого производится сдвиг,

– *Линейный сдвиг* на *k* позиций вдоль измерения решетки – в этом варианте операции данные от процессора *i* пересылаются процессору *i+k* (если таковой существует).

Функция *MPI_Cart_shift* обеспечивает получение рангов процессов, с которыми текущий процесс (процесс, вызвавший функцию *MPI_Cart_shift*) должен выполнить обмен данными:

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp,
  int *source, int *dst),
где:
 - comm   – коммуникатор с топологией решетки,
 - dir    - номер измерения, по которому выполняется сдвиг,
 - disp   - величина сдвига (<0 – сдвиг к началу измерения),
 - source – ранг процесса, от которого должны быть получены данные,
 - dst    - ранг процесса которому должны быть отправлены данные.
```

Следует отметить, что функция *MPI_Cart_shift* только определяет ранги процессов, между которыми должен быть выполнен обмен данными в ходе операции сдвига. Непосредственная передачами данных, может быть выполнена, например, при помощи функции *MPI_Sendrecv*.

4.7.2. Топологии графа

Сведения по функциям MPI для работы с виртуальными топологиями типа граф будут рассмотрены более кратко – дополнительная информация может быть получена, например, Немнюгин и Стесик (2002), Group, et al. (1994), Pacheco (1996).

Для создания коммуникатора с топологией типа граф в MPI предназначена функция:

```
int MPI_Graph_create(MPI_Comm oldcomm, int nnodes, int *index, int *edges,
  int reorder, MPI_Comm *graphcomm),
где:
 - oldcomm  - исходный коммуникатор,
 - nnodes   - количество вершин графа,
 - index    - количество исходящих дуг для каждой вершины,
 - edges    - последовательный список дуг графа,
 - reorder  - параметр допустимости изменения нумерации процессов,
 - cartcomm – создаваемый коммуникатор с топологией типа граф.
```

Операция создания топологии является коллективной и, тем самым, должна выполняться всеми процессами исходного коммуникатора.

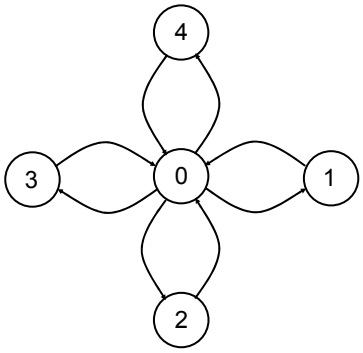


Рис. 4.9. Пример графа для топологии типа звезда

Для примера создадим топологию графа со структурой, представленной на рис. 4.9. В этом случае количество процессов равно 5, порядки вершин (количества исходящих дуг) принимают значения (4,1,1,1,1), а матрица инцидентности (номера вершин, для которых дуги являются входящими) имеет вид:

```
Процессы   Линии связи
  0        1, 2, 3, 4
  1           0
  2           0
  3           0
  4           0
```

Для создания топологии с графом данного вида необходимо выполнить следующий программный код:

```
// создание топологии типа звезда
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Приведем еще две полезные функции для работы с топологиями графа. Количество соседних процессов, в которых от проверяемого процесса есть выходящие дуги, может быть получено при помощи функции:

```
int MPI_Graph_neighbors_count(MPI_Comm comm,int rank, int *nneighbors).
```

Получение рангов соседних вершин обеспечивается функцией:

```
int MPI_Graph_neighbors(MPI_Comm comm,int rank,int mneighbors, int *neighbors),
```

где *mneighbors* есть размер массива *neighbors*.

4.8. Дополнительные сведения о MPI

4.8.1. Разработка параллельных программ с использованием MPI на алгоритмическом языке Fortran

При разработке параллельных программ с использованием MPI на алгоритмическом языке Fortran существует не так много особенностей по сравнению с применением алгоритмического языка C:

1. Подпрограммы библиотеки MPI являются процедурами и, тем самым, вызываются при помощи оператора вызова процедур CALL,

2. Коды завершения передаются через дополнительный параметр целого типа, располагаемый на последнем месте в списке параметров процедур,

3. Переменная status является массивом целого типа из *MPI_STATUS_SIZE* элементов,

4. Типы *MPI_Comm* и *MPI_Datatype* представлены целых типом INTEGER.

В качестве принятых соглашений при разработке программ на языке Fortran рекомендуется записывать имена подпрограмм с использованием прописных символов.

В качестве примера приведем вариант программы из п. 4.2.1.5 на алгоритмическом языке Fortran.

```
PROGRAM MAIN
  include 'mpi.h'
  INTEGER PROCNUM, PROCRANK, RECVRANK, IERR
  INTEGER STATUS(MPI_STATUS_SIZE)
  CALL MPI_Init(IERR)
  CALL MPI_Comm_size(MPI_COMM_WORLD, PROCNUM, IERR)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, PROCRANK IERR)
  IF ( PROCRANK.EQ.0 )THEN
    ! Действия, выполняемые только процессом с рангом 0
    PRINT *,"Hello from process ", PROCRANK
    DO i = 1, PROCNUM-1
      CALL MPI_RECV(RECVRANK, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, STATUS, IERR)
      PRINT *,"Hello from process ", RECVRANK
    END DO
  ELSE ! Сообщение, отправляемое всеми процессами,
       ! кроме процесса с рангом 0
    CALL MPI_SEND(PROCRANK,1,MPI_INT,0,0,MPI_COMM_WORLD,IERR)
  END IF
```

```
  MPI_FINALIZE(IERR);
  STOP
END
```

4.8.2. Общая характеристика среды выполнения MPI-программ

Для проведения параллельных вычислений в вычислительной системе должна быть установлена среда выполнения MPI-программ, которая бы обеспечивала разработку, компиляцию, компоновку и выполнение параллельных программ. Для выполнения первой части перечисленных действий - разработки, компиляции, компоновки – как правило, достаточно обычных средств разработки программ (таких, например, как Microsoft Visual Studio), необходимо лишь наличие той или иной библиотеки MPI. Для выполнения же параллельных программ от среды выполнения требуется ряд дополнительных возможностей, среди которых наличие средств указания используемых процессоров, операций удаленного запуска программ и т.п. Крайне желательно также наличие в среде выполнения средств профилирования, трассировки и отладки параллельных программ.

Здесь, к сожалению, стандартизация во многом заканчивается. Существует несколько различных сред выполнения MPI-программ. В большинстве случаев, эти среды создаются совместно с разработкой тех или иных вариантов MPI-библиотек. Обычно выбор реализации MPI-библиотеки, установка среды выполнения и подготовка инструкций по использованию осуществляется администратором вычислительной системы. Как правило, информационные ресурсы сети Интернет, на которых располагаются свободно используемые реализации MPI, и промышленные версии MPI содержат исчерпывающую информацию о процедурах установки MPI, и выполнение всех необходимых действий не составляет большого труда.

Запуск MPI-программы также зависит от среды выполнения, но в большинстве случаев данное действие выполняется при помощи команды **mpirun**. В числе возможных параметров этой команды:

– *Режим выполнения* – локальный или многопроцессорный. Локальный режим обычно указывается при помощи ключа *–localonly*. При выполнении параллельной программы в локальном режиме все процессы программы располагаются на компьютере, с которого был произведен запуск программы. Такой способ выполнения чрезвычайно полезен для начальной проверки работоспособности и отладки параллельной программы; часть такой работы может быть выполнена даже на отдельном компьютере вне рамок многопроцессорной вычислительной системы;

– *Количество процессов*, которые необходимо создать при запуске параллельной программы,

– *Состав используемых процессоров*, определяемый тем или иным конфигурационным файлом;

– *Исполняемый файл* параллельной программы;

– *Командная строка* с параметрами для выполняемой программы.

Существует и значительное количество других параметров, но они обычно используются при разработке достаточно сложных параллельных программ – их описание может быть получено в справочной информации по соответствующей среде выполнения MPI-программ.

При запуске программы на нескольких компьютерах исполняемый файл программы должен быть скопирован на все эти компьютеры или же должен находиться на общем доступном для всех компьютеров ресурсе.

Дополнительная информация по средам выполнения параллельных программ на кластерных системах может быть получена, например, в Sterling (2001, 2002). Следует отметить также, что элементы стандартизации среды выполнения MPI-программ содержатся в стандарте MPI-2.

4.8.3. Дополнительные возможности стандарта MPI-2

Как уже отмечалось, стандарт MPI-2 был принят в 1997 г. Несмотря на достаточно большой период времени, прошедший с тех пор, использование данного варианта стандарта все еще ограничено. Среди основных причин такой ситуации можно назвать обычный консерватизм разработчиков программного обеспечения, сложность реализации нового стандарта и т.п. Важный момент состоит также в том, что возможностей MPI-1 оказывается достаточным для реализации многих параллельных алгоритмов, а сфера применимости дополнительных возможностей MPI-2 оказывается не столь широкой.

Для знакомства со стандартом MPI-2 может быть рекомендован информационный ресурс **http://www.mpiforum.org**, а также работа Group, et al. (1999b). Здесь же дадим краткую характеристику дополнительных возможностей стандарта MPI-2:

– *Динамическое порождение процессов*, при котором процессы параллельной программы могут создаваться и уничтожаться в ходе выполнения,

– *Одностороннее взаимодействие процессов*, что позволяет быть инициатором операции передачи и приема данных только одному процессу,

– *Параллельный ввод/вывод*, обеспечивающий специальный интерфейс для работы процессов с файловой системой,

– *Расширенные коллективные операции*, в числе которых, например, процедуры для взаимодействия процессов из нескольких коммуникаторов одновременно,

– *Интерфейс для алгоритмического языка C++*.

4.9. Краткий обзор раздела

Данный раздел посвящен рассмотрению методов параллельного программирования для вычислительных систем с распределенной памятью с использованием MPI.

В самом начале раздела отмечается, что MPI - *интерфейс передачи данных* (*message passing interface*) - является в настоящий момент времени одним из основных подходов для разработки параллельных программ для вычислительных систем с распределенной памятью. Использование MPI позволяет распределить вычислительную нагрузку и организовать информационное взаимодействие (*передачу данных*) между процессорами. Сам термин MPI означает, с одной стороны, стандарт, которому должны удовлетворять средства организации передачи сообщений, а, с другой стороны, обозначает программные библиотеки, которые обеспечивают возможность передачи сообщений и при этом соответствуют всем требованиям стандарта MPI.

В подразделе 4.1 рассматривается ряд понятий и определений, являющихся основополагающими для стандарта MPI. Так, дается представление *параллельной программы* как множества одновременно выполняемых *процессов*. При этом процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). Далее приводится краткая характеристика понятий для операций передачи сообщений, типов данных, коммуникаторов и виртуальных топологий.

В подразделе 4.2 проводится быстрое и простое введение в разработку параллельных программ с использованием MPI. Излагаемого в подразделе материала достаточно для начала разработки параллельных программ разного уровня сложности.

В подразделе 4.3 излагается материал, связанный с *операциями передачи данных между двумя процессами*. В данном подразделе подробно излагаются имеющиеся в MPI режимы выполнения операций – стандартный, синхронный, буферизованный, по готовности. Для всех рассмотренных операций обсуждается возможность организации неблокирующих обменов данными между процессами.

В подразделе 4.4 рассматриваются *коллективные операции передачи данных*. Изложение материала соответствует последовательности изучения коммуникационных операций, использованной в разделе 3. Основной результат данного подраздела состоит в том, что MPI обеспечивает поддержку практически всех основных операций информационных обменов между процессами.

В подразделе 4.5 излагается материал, связанный с использованием в MPI *производных типов данных*. В подразделе представлены все основные способы конструирования производных типов – непрерывный, векторный, индексный и структурный. Обсуждается также возможность явного формирования сложных сообщений при помощи упаковки и распаковки данных.

В подразделе 4.6 обсуждаются вопросы *управления процессами и коммуникаторами*. Рассматриваемые в подразделе возможности MPI позволяют управлять областями действия коллективных операций и исключить взаимовлияние разных выполняемых частей параллельной программы.

В подразделе 4.7 рассматриваются возможности MPI по использованию *виртуальных топологий*. В подразделе представлены топологии, поддерживаемые MPI - *прямоугольная решетка* произвольной размерности (*декартова топология*) и топология *графа* любого необходимого вида.

В подразделе 4.8 приводятся дополнительные сведения о MPI. В их числе обсуждаются вопросы разработки параллельных программ с использованием MPI на алгоритмическом языке Fortran, дается краткая характеристика сред выполнения MPI-программ и приводится обзор дополнительных возможностей стандарта MPI-2.

4.10. Обзор литературы

Имеется ряд источников, в которых может быть получена информация о MPI. Прежде всего, это информационный ресурс Интернет с описанием стандарта MPI: http://www.mpiforum.org. Одна из наиболее распространенных реализаций MPI - библиотека MPICH представлена на http://www-unix.mcs.anl.gov/mpi/mpich (библиотека MPICH2 с реализацией стандарта MPI-2 содержится на http://www-unix.mcs.anl.gov/mpi/mpich2). Русскоязычные материалы о MPI имеются на сайте http://www.parallel.ru.

Среди опубликованных изданий могут быть рекомендованы работы Group, et al. (1994), Pacheco (1996), Snir, et al. (1996), Group, et al. (1999a). Описание стандарта MPI-2 может быть получено в Group, et al.

(1999b). Среди русскоязычных изданий могут быть рекомендованы работы Воеводин В.В. и Воеводин Вл.В. (2002), Немнюгина и Стесик (2002), Корнеева (2003).

Следует отметить также работу Quinn (2003), в которой изучение MPI проводится на примере ряда типовых задач параллельного программирования – матричных вычислений, сортировки, обработки графов и др.

4.11. Контрольные вопросы

1. Какой минимальный набор средств является достаточным для организации параллельных вычислений в системах с распределенной памятью?
2. В чем состоит важность стандартизации средств передачи сообщений?
3. Что следует понимать под параллельной программой?
4. В чем различие понятий процесса и процессора?
5. Какой минимальный набор функций MPI позволяет начать разработку параллельных программ?
6. Как описываются передаваемые сообщения?
7. Как можно организовать прием сообщений от конкретных процессов?
8. Как определить время выполнения MPI программы?
9. В чем различие парных и коллективных операций передачи данных?
10. Какая функция MPI обеспечивает передачу данных от одного процесса всем процессам?
11. Что понимается под операцией редукции?
12. В каких ситуациях следует применять барьерную синхронизацию?
13. Какие режимы передачи данных поддерживаются в MPI?
14. Как организуется неблокирующий обмен данными в MPI?
15. В чем состоит понятие тупика? В каких ситуациях функция одновременного выполнения передачи и приема гарантирует отсутствие тупиковых ситуаций?
16. Какие коллективные операции передачи данных предусмотрены в MPI?
17. Что понимается под производным типом данных в MPI?
18. Какие способы конструирования типов имеются в MPI?
19. В каких ситуациях может быть полезна упаковка и распаковка данных?
20. Что понимается в MPI под коммуникатором?
21. Для чего может потребоваться создание новых коммуникаторов?
22. Что понимается в MPI под виртуальной топологией?
23. Какие виды топологий предусмотрены в MPI?
24. Для чего может оказаться полезным использование виртуальных топологий?
25. В чем состоят особенности разработки параллельных программ с использованием MPI на алгоритмическом языке Fortran?
26. Какие основные дополнительные возможности предусмотрены в стандарте MPI-2?

4.12. Задачи и упражнения

Подраздел 4.2.

1. Разработайте программу для нахождения минимального (максимального) значения среди элементов вектора.
2. Разработайте программу для вычисления скалярного произведения двух векторов.
3. Разработайте программу, в которой два процесса многократно обмениваются сообщениями длиной n байт. Выполните эксперименты и оцените зависимость времени выполнения операции данных от длины сообщения. Сравните с теоретическими оценками, построенными по модели Хокни.

Подраздел 4.3.

4. Подготовьте варианты ранее разработанных программ с разными режимами выполнения операций передачи данных. Сравните времена выполнения операций передачи данных при разных режимах работы.
5. Подготовьте варианты ранее разработанных программ с использованием неблокирующего способа выполнения операций передачи данных. Оцените необходимое количество вычислительных операций, для того чтобы полностью совместить передачу данных и вычисления. Разработайте программу, в которой бы полностью отсутствовали задержки вычислений из-за ожидания передаваемых данных.

6. Выполните задание 3 с использованием операции одновременного выполнения передачи и приема данных. Сравните результаты вычислительных экспериментов.

Подраздел 4.4.

7. Разработайте программу-пример для каждой имеющейся в MPI коллективной операции.

8. Разработайте реализации коллективных операций при помощи парных обменов между процессами. Выполните вычислительные эксперименты и сравните времена выполнения разработанных программ и функций MPI для коллективных операций.

9. Разработайте программу, выполните эксперименты и сравните результаты для разных алгоритмов реализации операции сбора, обработки и рассылки данных всех процессам (функция MPI_Allreduce).

Подраздел 4.5.

10. Разработайте программу-пример для каждого имеющегося в MPI способа конструирования производных типов данных.

11. Разработайте программу-пример с использованием функций упаковки и распаковки данных. Выполните эксперименты и сравните с результатами при использовании производных типов данных.

12. Разработайте производные типы данных для строк, столбцов, диагоналей матриц.

13. Разработайте программу-пример для каждой из рассмотренных функций для управления процессами и коммуникаторами.

14. Разработайте программу для представления множества процессов в виде прямоугольной решетки. Создайте коммуникаторы для каждой строки и столбца процессов. Выполните коллективную операцию для всех процессов и для одного из созданных коммуникаторов. Сравните время выполнения операции.

15. Изучите самостоятельно и разработайте программы-примеры для передачи данных между процессами разных коммуникаторов.

Подраздел 4.7.

16. Разработайте программу-пример для декартовой топологии.

17. Разработайте программу-пример для топологии графа.

18. Разработайте подпрограммы для создания некоторого набора дополнительных виртуальных топологий (звезда, дерево и др.).

Литература

Воеводин В.В., Воеводин Вл.В. (2002). Параллельные вычисления. – СПб.: БХВ-Петербург.

Гергель, В.П., Стронгин, Р.Г. (2001). Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ (2 изд., 2003).

Корнеев В.В. (2003) Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований,2003

Немнюгин С., Стесик О. (2002). Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург.

Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.

Gropp, W., Lusk, E., Skjellum, A. (1999a). Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.

Gropp, W., Lusk, E., Thakur, R. (1999b). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996). MPI: The Complete Reference. - MIT Press, Boston, 1996.