Московский государственный университет им. М.В.Ломоносова
Научно-исследовательский вычислительный центр

А.С.Антонов

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ MPI

Издательство
Московского университета
2004

УДК 681.3.06
ББК – 018.2*32.973
А 72

Рецензенты:
зам. директора НИВЦ МГУ, член-корреспондент РАН Вл.В.Воеводин,
зам. директора НИИЯФ МГУ, доктор физико-математических наук
В.А.Ильин.

Антонов А.С.
А 72 **Параллельное программирование с использованием технологии MPI:** Учебное пособие. – М.: Изд-во МГУ, 2004. – 71 с.

ISBN 5-211-04907-1

Пособие предназначено для освоения практического курса параллельного программирования с использованием технологии MPI. В настоящее время технология MPI является основным средством программирования для кластерных систем и компьютеров с распределенной памятью, но может применяться также и на вычислительных системах других типов. Курс включает в себя описание большинства основных процедур стандарта MPI-1.1 с примерами их применения и практические сведения, которые могут потребоваться при написании реальных программ. Основное описание ведется с использованием вызовов процедур MPI из программ на языке Фортран, однако указаны также основные отличия в использовании вызовов аналогичных функций из программ на языке Си. Приводятся примеры небольших законченных параллельных программ, тексты которых можно скачать в сети Интернет со страницы http://parallel.ru/tech/tech_dev/MPI/examples/. В конце разделов приводятся контрольные вопросы и задания, которые можно использовать в процессе обучения. Пособие основано на курсе занятий, проведенных автором в июне 2003 года в компании «Шлюмберже».

Для студентов, аспирантов и научных сотрудников, чья деятельность связана с параллельными вычислениями.

УДК 681.3.06
ББК – 018.2*32.973

ISBN 5-211-04907-1

Содержание

Индекс по функциям MPI

Основные понятия

Наиболее распространенной технологией программирования для параллельных компьютеров с распределенной памятью в настоящее время является MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии данной технологии — *Message Passing Interface (интерфейс передачи сообщений)*. Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования на каждой вычислительной платформе, так и пользователем при создании своих программ. Современные реализации, чаще всего, соответствуют стандарту MPI версии 1.1. В 1997—1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения и в полном объеме не реализован ни на одной системе. Везде далее, если иного не оговорено, мы будем иметь дело со стандартом MPI-1.1.

MPI поддерживает работу с языками Фортран и Си. В данном пособии примеры и описания всех процедур будут даны с использованием языка Фортран. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи. Полная версия интерфейса содержит описание более 125 процедур и функций. Наша задача — объяснить идею технологии и помочь освоить необходимые на практике компоненты. Дополнительную информацию об интерфейсе MPI можно найти на тематической странице Информационно-аналитического центра по параллельным вычислениям в сети Интернет http://parallel.ru/tech/tech_dev/mpi.html.

Интерфейс MPI поддерживает создание параллельных программ в стиле MIMD (Multiple Instruction Multiple Data), что подразумевает объединение процессов с различными исходными текстами. Однако писать и отлаживать такие программы очень сложно, поэтому на практике программисты гораздо чаще используют *SPMD-модель (Single Program Multiple Data)* параллельного программирования, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Поскольку MPI является библиотекой, то при компиляции программы необходимо прилинковать соответствующие библиотечные модули. Это можно сделать в командной строке или воспользоваться предусмотренными в большинстве систем командами или скриптами **`mpicc`** (для программ на языке Си), **`mpiCC`** (для программ на языке Си++), и **`mpif77/mpif90`** (для программ на языках Фортран 77/90). Опция компилятора "**`-o name`**" позволяет задать имя

name для получаемого выполнимого файла, по умолчанию выполнимый файл называется **a.out**, например:

```
mpif77 -o program program.f
```

После получения выполнимого файла необходимо запустить его на требуемом количестве процессоров. Для этого обычно предоставляется команда запуска MPI-приложений **mpirun**, например:

```
mpirun -np N <программа с аргументами>,
```

где **N** - число процессов, которое должно быть не более разрешенного в данной системе числа процессов для одной задачи. После запуска одна и та же программа будет выполняться всеми запущенными процессами, результат выполнения в зависимости от системы будет выдаваться на терминал или записываться в файл с предопределенным именем.

Все дополнительные объекты: имена процедур, константы, предопределенные типы данных и т.п., используемые в MPI, имеют префикс **MPI_**. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. В языке Си, кроме того, является существенным регистр символов в названиях функций. Обычно в названиях функций MPI первая буква после префикса **MPI_** пишется в верхнем регистре, последующие буквы – в нижнем регистре, а названия констант MPI записываются целиком в верхнем регистре. Все описания интерфейса MPI собраны в файле **mpif.h** (**mpi.h**), поэтому в начале MPI-программы должна стоять директива **include 'mpif.h'** (**#include "mpi.h"** для программ на языке Си).

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается (в MPI-2.0 такая возможность появилась). Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать *группы процессов*, предоставляя им отдельную среду для общения — *коммуникатор*. Состав образуемых групп произволен. Группы могут полностью совпадать, входить одна в другую, не пересекаться или пересекаться частично. Процессы могут взаимодействовать только внутри некоторого коммуникатора, сообщения, отправленные в разных коммуникаторах, не пересекаются и не мешают друг другу. Коммуникаторы имеют в языке Фортран тип **INTEGER** (в языке Си – предопределенный тип **MPI_Comm**).

При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя **MPI_COMM_WORLD**. Этот коммуникатор существует всегда и служит для взаимодействия всех запущенных процессов MPI-программы. Кроме него при старте программы имеется коммуникатор **MPI_COMM_SELF**, содержащий только один текущий процесс, а также коммуникатор **MPI_COMM_NULL**, не содержащий ни одного процесса. Все взаимодействия процессов протекают в рамках определенного коммуникатора, сообщения, переданные в разных коммуникаторах, никак не мешают друг другу.

Каждый процесс MPI-программы имеет в каждой группе, в которую он входит, уникальный атрибут *номер процесса*, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда становятся понятными *два основных атрибута процесса: коммуникатор и номер в коммуникаторе*. Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до $n - 1$.

Основным способом общения процессов между собой является явная посылка сообщений. *Сообщение* — это набор данных некоторого типа. Каждое сообщение имеет несколько *атрибутов*, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и другие. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от **0** до **MPI_TAG_UP**, причем гарантируется, что **MPI_TAG_UP** не меньше **32767**. Для работы с атрибутами сообщений введен массив (в языке Си – структура), элементы которого дают доступ к их значениям.

В последнем аргументе (в языке Си – в возвращаемом значении функции) большинство процедур MPI возвращают информацию об успешности завершения. В случае успешного выполнения возвращается значение **MPI_SUCCESS**, иначе – код ошибки. Вид ошибки, которая произошла при выполнении процедуры, можно будет определить из ее описания. Предопределенные значения, соответствующие различным ошибочным ситуациям, перечислены в файле **mpif.h**.

Общие процедуры MPI

В данном разделе мы остановимся на общих процедурах MPI, не связанных с пересылкой данных. Большинство процедур этого раздела необходимы практически в каждой содержательной параллельной программе.

MPI_INIT(IERR)
INTEGER IERR
Инициализация параллельной части программы. Все другие процедуры MPI могут быть вызваны только после вызова **MPI_INIT**. Инициализация параллельной части для каждого приложения должна выполняться только один раз. В языке Си функции **MPI_Init** передаются указатели на аргументы командной строки программы **argc** и **argv**, из которых системой могут извлекаться и передаваться в параллельные процессы некоторые параметры запуска программы.

MPI_FINALIZE(IERR)
INTEGER IERR
Завершение параллельной части приложения. Все последующие обращения к любым процедурам MPI, в том числе к **MPI_INIT**, запрещены. К моменту вызова **MPI_FINALIZE** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Пример простейшей MPI-программы на языке Фортран выглядит следующим образом:

```
program example1
include 'mpif.h'
integer ierr
print *, 'Before MPI_INIT'
call MPI_INIT(ierr)
print *, 'Parallel section'
call MPI_FINALIZE(ierr)
print *, 'After MPI_FINALIZE'
end
```

В зависимости от реализации MPI строчки **'Before MPI_INIT'** и **'After MPI_FINALIZE'** может печатать либо один выделенный процесс, либо все запущенные процессы приложения. Строчку **'Parallel section'** должны напечатать все процессы. Порядок вывода строк с разных процессов может быть произвольным.

Общая схема MPI-программы на языке Си выглядит примерно следующим образом:

```
#include “mpi.h”
main(int argc, char **argv)
{
 …
 MPI_Init(&argc, &argv);
 …
 MPI_Finalize();
 …
}
```

Другие параллельные программы на языке Си с использованием технологии MPI можно найти, например, в Вычислительном полигоне: http://polygon.parallel.ru.

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

Процедура возвращает в аргументе **FLAG** значение **.TRUE.**, если вызвана из параллельной части приложения, и значение **.FALSE.** - в противном случае. Это единственная процедура MPI, которую можно вызвать до вызова **MPI_INIT**.

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

В аргументе **SIZE** процедура возвращает число параллельных процессов в коммуникаторе **COMM**.

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

В аргументе **RANK** процедура возвращает номер процесса в коммуникаторе **COMM**. Если процедура **MPI_COMM_SIZE** для того же коммуникатора **COMM** вернула значение **SIZE**, то значение, возвращаемое процедурой **MPI_COMM_RANK** через переменную **RANK**, лежит в диапазоне от **0** до **SIZE-1**.

В следующем примере каждый запущенный процесс печатает свой уникальный номер в коммуникаторе **MPI_COMM_WORLD** и число процессов в данном коммуникаторе.

```
      program example2
      include ‘mpif.h’
      integer ierr, size, rank
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      print *, ‘process ’, rank, ‘, size ’, size
      call MPI_FINALIZE(ierr)
      end
```

Строка, соответствующая вызову процедуры **print**, будет выведена столько раз, сколько процессов было порождено при запуске программы. Порядок появления строк заранее не определен и может быть, вообще говоря, любым. Гарантируется только то, что содержимое отдельных строк не будет перемешано друг с другом.

```
DOUBLE PRECISION MPI_WTIME(IERR)
INTEGER IERR
```

Эта функция возвращает на вызвавшем процессе астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса. Заметим, что эта функция возвращает результат своей работы не через параметры, а явным образом. Таймеры разных процессоров могут быть не синхронизированы и выдавать различные значения, это можно определить по значению параметра **MPI_WTIME_IS_GLOBAL** (**1** – синхронизированы, **0** - нет).

```
DOUBLE PRECISION MPI_WTICK(IERR)
INTEGER IERR
```

Функция возвращает разрешение таймера на вызвавшем процессе в секундах. Эта функция также возвращает результат своей работы не через параметры, а явным образом.

```
MPI_GET_PROCESSOR_NAME(NAME, LEN, IERR)
CHARACTER*(*) NAME
INTEGER LEN, IERR
```

Процедура возвращает в строке **NAME** имя узла, на котором запущен вызвавший процесс. В переменной **LEN** возвращается количество символов в имени, не превышающее значения константы **MPI_MAX_PROCESSOR_NAME**. С помощью этой процедуры можно определить, на какие именно физические процессоры были спланированы процессы MPI-приложения.

В следующей программе на каждом процессе определяются две характеристики системного таймера: его разрешение и время, требуемое на замер времени (для усреднения получаемого значения выполняется **NTIMES** замеров). Также в данном примере показано использование процедуры **MPI_GET_PROCESSOR_NAME**.

```
      program example3
      include 'mpif.h'
      integer ierr, rank, len, i, NTIMES
      parameter (NTIMES = 100)
      character*(MPI_MAX_PROCESSOR_NAME) name
      double precision time_start, time_finish, tick
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_GET_PROCESSOR_NAME(name, len, ierr)
      tick = MPI_WTICK(ierr)
      time_start = MPI_WTIME(ierr)
      do i = 1, NTIMES
         time_finish = MPI_WTIME(ierr)
      end do
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      print *, 'processor ', name(1:len),
     &         ', process ', rank, ': tick = ', tick,
     &         ', time = ', (time_finish-time_start)/NTIMES
      call MPI_FINALIZE(ierr)
      end
```

Задания

- Откомпилировать и проверить эффективность выполнения программы вычисления числа Пи на различном числе процессоров (программа обычно входит в качестве тестового примера в комплект поставки MPI и может находиться, например, в файлах **`/usr/local/examples/mpi/fpi.f`** или **`cpi.c`**).
- Можно ли в процессе работы MPI-программы порождать новые процессы, если в какой-то момент появились свободные процессоры?
- Может ли MPI-программа продолжать работу после аварийного завершения одного из процессов?
- Определить, сколько процессов выполняют текст программы до вызова процедуры **`MPI_INIT`** и после вызова процедуры **`MPI_FINALIZE`**.
- Определить, синхронизованы ли таймеры разных процессов конкретной системы.

Передача/прием сообщений между отдельными процессами

Практически все программы, написанные с использованием коммуникационной технологии MPI, должны содержать средства не только для порождения и завершения параллельных процессов, но и для взаимодействия запущенных

процессов между собой. Такое взаимодействие осуществляется в MPI посредством явной посылки сообщений.

Все процедуры передачи сообщений в MPI делятся на две группы. В одну группу входят процедуры, которые предназначены для взаимодействия только двух процессов программы. Такие операции называются индивидуальными или операциями типа точка-точка. Процедуры другой группы предполагают, что в операцию должны быть вовлечены все процессы некоторого коммуникатора. Такие операции называются коллективными.

Начнем описание процедур обмена сообщениями с обсуждения *операций типа точка-точка*. В таких взаимодействиях участвуют два процесса, причем один процесс является отправителем сообщения, а другой – получателем. Процесс-отправитель должен вызвать одну из процедур передачи данных и явно указать номер в некотором коммуникаторе процесса-получателя, а процесс-получатель должен вызвать одну из процедур приема с указанием того же коммуникатора, причем в некоторых случаях он может не знать точный номер процесса-отправителя в данном коммуникаторе.

Все процедуры данной группы, в свою очередь, так же делятся на два класса: процедуры с блокировкой (с синхронизацией) и процедуры без блокировки (асинхронные). Процедуры обмена с блокировкой приостанавливают работу процесса до выполнения некоторого условия, а возврат из асинхронных процедур происходит немедленно после инициализации соответствующей коммуникационной операции. Неаккуратное использование процедур с блокировкой может привести к возникновению тупиковой ситуации, поэтому при этом требуется дополнительная осторожность. Использование асинхронных операций к тупиковым ситуациям не приводит, однако требует более аккуратного использования массивов данных.

Передача/прием сообщений с блокировкой

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, MSGTAG, COMM, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, MSGTAG, COMM, IERR
```

Блокирующая посылка массива **BUF** с идентификатором **MSGTAG**, состоящего из **COUNT** элементов типа **DATATYPE**, процессу с номером **DEST** в коммуникаторе **COMM**. Все элементы посылаемого сообщения должны быть расположены подряд в буфере **BUF**. Операция начинается независимо от того, была ли инициализирована соответствующая процедура приема. При этом сообщение может быть скопировано как непосредственно в буфер приема, так и помещено в некоторый системный буфер (если это предусмотрено в MPI). Значение **COUNT** может быть нулем. Процессу разрешается передавать сообщение

самому себе, однако это небезопасно и может привести к возникновению тупиковой ситуации. Параметр **DATATYPE** имеет в языке Фортран тип **INTEGER** (в языке Си – предопределенный тип **MPI_Datatype**). Тип передаваемых элементов должен указываться с помощью предопределенных констант типа, перечисленных для языка Фортран в следующей таблице.

Тип данных в MPI	Тип данных в Фортране
MPI_INTEGER	**INTEGER**
MPI_REAL	**REAL**
MPI_DOUBLE_PRECISION	**DOUBLE PRECISION**
MPI_COMPLEX	**COMPLEX**
MPI_LOGICAL	**LOGICAL**
MPI_CHARACTER	**CHARACTER(1)**
MPI_BYTE	**8 бит, используется для передачи нетипизированных данных**
MPI_PACKED	**тип для упакованных данных**

Если используемый с MPI базовый язык имеет дополнительные типы данных, то соответствующие типы должны быть обеспечены и в MPI. Полный список предопределенных имен типов данных перечислен в файле **mpif.h** (**mpi.h**).

При пересылке сообщений можно использовать специальное значение **MPI_PROC_NULL** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения **MPI_SUCCESS**. Например, для пересылки сообщения процессу с номером на единицу больше можно воспользоваться следующим фрагментом:

```
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      next = rank+1
      if(next .eq. size) next = MPI_PROC_NULL
      call MPI_SEND(buf, 1, MPI_REAL, next,
     &              5, MPI_COMM_WORLD, ierr)
```

В этом случае процесс с последним номером не осуществит никакой реальной посылки данных, а сразу пойдет выполнять программу дальше.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из процедуры. Это означает, что после возврата из **MPI_SEND** можно использовать любые присутствующие в вызове данной процедуры переменные без опасения испортить передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу **DEST**, остается за разработчиками конкретной реализации MPI.

Следует специально отметить, что возврат из процедуры **MPI_SEND** не означает ни того, что сообщение получено процессом **DEST**, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший данный вызов. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове данной процедуры. Подобная неопределенность далеко не всегда устраивает пользователя. Чтобы расширить возможности передачи сообщений, в MPI введены дополнительные три процедуры. Все параметры у этих процедур такие же, как и у **MPI_SEND**, однако у каждой из них есть своя особенность.

MPI предоставляет следующие модификации процедуры передачи данных с блокировкой **MPI_SEND**:

- **MPI_BSEND** — передача сообщения *с буферизацией*. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в специальный буфер, и произойдет немедленный возврат из процедуры. Выполнение данной процедуры никак не зависит от соответствующего вызова процедуры приема сообщения. Тем не менее, процедура может вернуть код ошибки, если места под буфер недостаточно. О выделении массива для буферизации должен заботиться пользователь.
- **MPI_SSEND** — передача сообщения *с синхронизацией*. Выход из данной процедуры произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера посылки, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Использование передачи сообщений с синхронизацией может замедлить выполнение программы, но позволяет избежать наличия в системе большого количества не принятых буферизованных сообщений.
- **MPI_RSEND** — передача сообщения *по готовности*. Данной процедурой можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов процедуры, вообще говоря, является ошибочным и результат ее выполнения не определен. Гарантировать инициализацию приема сообщения перед вызовом процедуры **MPI_RSEND** можно с помощью операций, осуществляющих явную или неявную синхронизацию процессов (например, **MPI_BARRIER** или **MPI_SSEND**). Во многих реализациях процедура **MPI_RSEND** сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи данных.

Пользователь должен назначить на посылающем процессе специальный массив, который будет использоваться для буферизации сообщений при вызове процедуры **MPI_BSEND**.

```
MPI_BUFFER_ATTACH(BUF, SIZE, IERR)
<type> BUF(*)
INTEGER SIZE, IERR
```

Назначение массива **BUF** размера **SIZE** для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер. Ассоциированный с буфером массив не следует использовать в программе для других целей. Размер массива, выделяемого для буферизации, должен превосходить общий размер сообщения как минимум на величину, определяемую константой **MPI_BSEND_OVERHEAD**.

```
MPI_BUFFER_DETACH(BUF, SIZE, IERR)
<type> BUF(*)
INTEGER SIZE, IERR
```

Освобождение выделенного буферного массива для его использования в других целях. Процедура возвращает в аргументах **BUF** и **SIZE** адрес и размер освобождаемого массива. Вызвавший процедуру процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

Обычно в MPI выделяется некоторый объем памяти для буферизации посылаемых сообщений. Однако, чтобы не полагаться на особенности конкретной реализации, рекомендуется явно выделять в программе достаточный буфер для всех пересылок с буферизацией.

В следующем примере показано использование передачи сообщения с буферизацией. Для буферизации выделяется массив **buf**, после завершения пересылки он освобождается. Размер необходимого буфера определяется размером сообщения (одно целое число – 4 байта) плюс значение константы **MPI_BSEND_OVERHEAD**.

```
      program example4
      include 'mpif.h'
      integer BUFSIZE
      parameter (BUFSIZE = 4 + MPI_BSEND_OVERHEAD)
      byte buf(BUFSIZE)
      integer rank, ierr, ibufsize, rbuf
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      if(rank .eq. 0) then
         call MPI_BUFFER_ATTACH(buf, BUFSIZE, ierr)
         call MPI_BSEND(rank, 1, MPI_INTEGER, 1, 5,
     &                  MPI_COMM_WORLD, ierr)
         call MPI_BUFFER_DETACH(buf, ibufsize, ierr)
      end if
      if(rank .eq. 1) then
         call MPI_RECV(rbuf, 1, MPI_INTEGER, 0, 5,
     &                 MPI_COMM_WORLD, status, ierr)
         print *, 'Process 1 received ', rbuf, ' from process ',
     &            status(MPI_SOURCE)
      end if
      call MPI_FINALIZE(ierr)
      end
```

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, MSGTAG, COMM, STATUS, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, MSGTAG, COMM, IERR, STATUS(MPI_STATUS_SIZE)

Блокирующий прием в буфер **BUF** не более **COUNT** элементов сообщения типа **DATATYPE** с идентификатором **MSGTAG** от процесса с номером **SOURCE** в коммуникаторе **COMM** с заполнением массива атрибутов приходящего сообщения **STATUS**. Если число реально принятых элементов меньше значения **COUNT**, то гарантируется, что в буфере **BUF** изменятся только элементы, соответствующие элементам принятого сообщения. Если количество элементов в принимаемом сообщении больше значения **COUNT**, то возникает ошибка переполнения. Чтобы избежать этого, можно сначала определить структуру приходящего сообщения при помощи процедуры **MPI_PROBE** (**MPI_IPROBE**). Если нужно узнать точное число элементов в принимаемом сообщении, то можно воспользоваться процедурой **MPI_GET_COUNT**. Блокировка гарантирует, что после возврата из процедуры **MPI_RECV** все элементы сообщения уже будут приняты и расположены в буфере **BUF**.

Ниже приведен пример программы, в которой нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа. Если программа будет запущена с б**о**льшим числом процессов, то реально выполнять пересылки все равно станут только нулевой и первый процессы. Остальные процессы после их инициализации процедурой **MPI_INIT** напечатают начальные

значения переменных **a** и **b**, после чего завершатся, выполнив процедуру **MPI_FINALIZE**.

```
      program example5
      include 'mpif.h'
      integer ierr, size, rank
      real a, b
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      a = 0.0
      b = 0.0
      if(rank .eq. 0) then
         b = 1.0
         call MPI_SEND(b, 1, MPI_REAL, 1, 5,
     &                 MPI_COMM_WORLD, ierr);
         call MPI_RECV(a, 1, MPI_REAL, 1, 5,
     &                 MPI_COMM_WORLD, status, ierr);
      else
         if(rank .eq. 1) then
            a = 2.0
            call MPI_RECV(b, 1, MPI_REAL, 0, 5,
     &                    MPI_COMM_WORLD, status, ierr);
            call MPI_SEND(a, 1, MPI_REAL, 0, 5,
     &                    MPI_COMM_WORLD, ierr);
         end if
      end if
      print *, 'process ', rank,' a = ', a, ', b = ', b
      call MPI_FINALIZE(ierr)
      end
```

В следующем примере каждый процесс с четным номером посылает сообщение своему соседу с номером на единицу большим. Дополнительно поставлена проверка для процесса с максимальным номером, чтобы он не послал сообщение несуществующему процессу. Значения переменной **b** изменятся только на процессах с нечетными номерами.

```
      program example6
      include 'mpif.h'
      integer ierr, size, rank, a, b
      integer status(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      a = rank
      b = -1
      if(mod(rank, 2) .eq. 0) then
         if(rank+1 .lt. size) then
C посылают все процессы, кроме последнего
            call MPI_Send(a, 1, MPI_INTEGER, rank+1, 5,
```

```
&                     MPI_COMM_WORLD, ierr);
    end if
 else
    call MPI_Recv(b, 1, MPI_INTEGER, rank-1, 5,
&                 MPI_COMM_WORLD, status, ierr);
 end if
 print *, 'process ', rank,' a = ', a, ', b = ', b
 call MPI_FINALIZE(ierr)
 end
```

При приеме сообщения вместо аргументов **SOURCE** и **MSGTAG** можно использовать следующие предопределенные константы:

- **MPI_ANY_SOURCE** — признак того, что подходит сообщение от любого процесса;
- **MPI_ANY_TAG** — признак того, что подходит сообщение с любым идентификатором.

При одновременном использовании этих двух констант будет принято сообщение с любым идентификатором от любого процесса.

Реальные атрибуты принятого сообщения всегда можно определить по соответствующим элементам массива **status**. В Фортране параметр **status** является целочисленным массивом размера **MPI_STATUS_SIZE**. Константы **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR** являются индексами по данному массиву для доступа к значениям соответствующих полей:

- **status(MPI_SOURCE)** — номер процесса-отправителя сообщения;
- **status(MPI_TAG)** — идентификатор сообщения;
- **status(MPI_ERROR)** — код ошибки.

В языке Си параметр **status** является структурой предопределенного типа **MPI_Status** с полями **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR**.

Обратим внимание на некоторую несимметричность операций посылки и приема сообщений. С помощью константы **MPI_ANY_SOURCE** можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

В стандарте оговорено, что если один процесс последовательно посылает два сообщения, соответствующие одному и тому же вызову **MPI_RECV**, другому процессу, то первым будет принято сообщение, которое было отправлено раньше. Вместе с тем, если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определен.

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERR)
INTEGER COUNT, DATATYPE, IERR, STATUS(MPI_STATUS_SIZE)
```
По значению параметра **STATUS** процедура определяет число **COUNT** уже принятых (после обращения к **MPI_RECV**) или принимаемых (после обращения к **MPI_PROBE** или **MPI_IPROBE**) элементов сообщения типа **DATATYPE**. Данная процедура, в частности, необходима для определения размера области памяти, выделяемой для хранения принимаемого сообщения.

```
MPI_PROBE(SOURCE, MSGTAG, COMM, STATUS, IERR)
INTEGER SOURCE, MSGTAG, COMM, IERR, STATUS(MPI_STATUS_SIZE)
```
Получение в массиве **STATUS** информации о структуре ожидаемого сообщения с идентификатором **MSGTAG** от процесса с номером **SOURCE** в коммуникаторе **COMM** с блокировкой. Возврата из процедуры не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Следует особо обратить внимание на то, что процедура определяет только факт прихода сообщения, но реально его не принимает. Если после вызова **MPI_PROBE** вызывается **MPI_RECV** с такими же параметрами, то будет принято то же самое сообщение, информация о котором была получена с помощью вызова процедуры **MPI_PROBE**.

Следующий пример демонстрирует применение процедуры **MPI_PROBE** для определения структуры приходящего сообщения. Процесс **0** ждет сообщения от любого из процессов **1** и **2** с одним и тем же тегом. Однако посылаемые этими процессами данные имеют разный тип. Для того чтобы определить, в какую переменную помещать приходящее сообщение, процесс сначала при помощи вызова **MPI_PROBE** определяет, от кого же именно поступило это сообщение. Следующий непосредственно после **MPI_PROBE** вызов **MPI_RECV** гарантированно примет нужное сообщение, после чего принимается сообщение от другого процесса.

```
      program example7
      include 'mpif.h'
      integer rank, ierr, ibuf, status(MPI_STATUS_SIZE)
      real rbuf
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      ibuf = rank
      rbuf = 1.0 * rank
      if(rank .eq. 1) call MPI_SEND(ibuf, 1, MPI_INTEGER, 0, 5,
     &                              MPI_COMM_WORLD, ierr)
      if(rank .eq. 2) call MPI_SEND(rbuf, 1, MPI_REAL, 0, 5,
     &                              MPI_COMM_WORLD, ierr)
      if(rank .eq. 0) then
         call MPI_PROBE(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD,
     &                  status, ierr)
```

```
    if(status(MPI_SOURCE) .EQ. 1) then
       call MPI_RECV(ibuf, 1, MPI_INTEGER, 1, 5,
&                    MPI_COMM_WORLD, status, ierr)
       call MPI_RECV(rbuf, 1, MPI_REAL, 2, 5,
&                    MPI_COMM_WORLD, status, ierr)
    else
       if(status(MPI_SOURCE) .EQ. 2) then
          call MPI_RECV(rbuf, 1, MPI_REAL, 2, 5,
&                       MPI_COMM_WORLD, status, ierr)
          call MPI_RECV(ibuf, 1, MPI_INTEGER, 1, 5,
&                       MPI_COMM_WORLD, status, ierr)
       end if
    end if
    print *, 'Process 0 recv ', ibuf, ' from process 1, ',
&                               rbuf, ' from process 2'
 end if
 call MPI_FINALIZE(ierr)
 end
```

В следующем примере моделируется последовательный обмен сообщениями между двумя процессами, замеряется время на одну итерацию обмена, определяется зависимость времени обмена от длины сообщения. Таким образом, определяются базовые характеристики коммуникационной сети параллельного компьютера: латентность (время на передачу сообщения нулевой длины) и максимально достижимая пропускная способность (количество мегабайт в секунду) коммуникационной сети, а также длина сообщений, на которой она достигается. Константа **NMAX** задает ограничение на максимальную длину посылаемого сообщения, а константа **NTIMES** определяет количество повторений для усреднения результата. Сначала посылается сообщение нулевой длины для определения латентности, затем длина сообщений удваивается, начиная с посылки одного элемента типа **real*8**.

```
program example8
include 'mpif.h'
integer ierr, rank, size, i, n, lmax, NMAX, NTIMES
parameter (NMAX = 1 000 000, NTIMES = 10)
double precision time_start, time, bandwidth, max
real*8 a(NMAX)
integer status(MPI_STATUS_SIZE)

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

time_start = MPI_WTIME(ierr)
n = 0
max = 0.0
lmax = 0
do while(n .le. NMAX)
   time_start = MPI_WTIME(ierr)
```

```
      do i = 1, NTIMES
         if(rank .eq. 0) then
            call MPI_SEND(a, n, MPI_DOUBLE_PRECISION, 1, 1,
&                         MPI_COMM_WORLD, ierr)
            call MPI_RECV(a, n, MPI_DOUBLE_PRECISION, 1, 1,
&                         MPI_COMM_WORLD, status, ierr)
         end if
         if(rank .eq. 1) then
            call MPI_RECV(a, n, MPI_DOUBLE_PRECISION, 0, 1,
&                         MPI_COMM_WORLD, status, ierr)
            call MPI_SEND(a, n, MPI_DOUBLE_PRECISION, 0, 1,
&                         MPI_COMM_WORLD, ierr)
         end if
      enddo
      time = (MPI_WTIME(ierr)-time_start)/2/NTIMES
      bandwidth = (8*n*1.d0/(2**20))/time
      if(max .lt. bandwidth) then
         max = bandwidth
         lmax = 8*n
      end if
      if(rank .eq. 0) then
         if(n .eq. 0) then
            print *, 'latency = ', time, ' seconds'
         else
            print *, 8*n, ' bytes, bandwidth =', bandwidth,
&                    ' Mb/s'
         end if
      end if
      if(n .eq. 0) then
         n = 1
      else
         n = 2*n
      end if
   end do
   if(rank .eq. 0) then
      print *, 'max bandwidth =', max, ' Mb/s , length =',
&             lmax, ' bytes'
   end if
   call MPI_FINALIZE(ierr)
   end
```

Передача/прием сообщений без блокировки

В MPI предусмотрен набор процедур для осуществления *асинхронной передачи данных*. В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без какой-либо остановки работы процессов. На фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции.

В принципе, данная возможность исключительно полезна для создания эффективных программ. В самом деле, программист знает, что в некоторый момент ему потребуется массив, который вычисляет другой процесс. Он заранее выставляет в программе асинхронный запрос на получение данного массива, а до того момента, когда массив реально потребуется, он может выполнять любую другую полезную работу. Опять же, во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений. Для завершения асинхронного обмена требуется вызов дополнительной процедуры, которая проверяет, завершилась ли операция, или дожидается ее завершения. Только после этого можно использовать буфер посылки для других целей без опасения запортить отправляемое сообщение.

Если есть возможность операции приема/передачи сообщений скрыть на фоне вычислений, то этим, вроде бы, надо безоговорочно пользоваться. Однако на практике не все всегда согласуется с теорией. Многое зависит от конкретной реализации. К сожалению, далеко не всегда асинхронные операции эффективно поддерживаются аппаратурой и системным окружением. Поэтому не стоит удивляться, если эффект от выполнения вычислений на фоне пересылок окажется нулевым или совсем небольшим. Сделанные замечания касаются только вопросов эффективности. В отношении предоставляемой функциональности асинхронные операции исключительно полезны, поэтому они присутствуют практически в каждой реальной программе.

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, MSGTAG, COMM, REQUEST,
IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, MSGTAG, COMM, REQUEST, IERR
```

Неблокирующая посылка из буфера **BUF COUNT** элементов сообщения типа **DATATYPE** с идентификатором **MSGTAG** процессу **DEST** коммуникатора **COMM**. Возврат из процедуры происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере **BUF**. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации, подтверждающей завершение данной посылки. Определить тот момент времени, когда можно повторно использовать буфер **BUF** без опасения испортить передаваемое сообщение, можно с помощью возвращаемого параметра **REQUEST** и процедур семейств **MPI_WAIT** и **MPI_TEST**. Параметр **REQUEST** имеет в языке Фортран тип **INTEGER** (в языке Си – предопределенный тип **MPI_Request**) и используется для идентификации конкретной неблокирующей операции.

Аналогично трем модификациям процедуры **MPI_SEND**, предусмотрены три дополнительных варианта процедуры **MPI_ISEND**:

- **MPI_IBSEND** — неблокирующая передача сообщения с буферизацией;
- **MPI_ISSEND** — неблокирующая передача сообщения с синхронизацией;
- **MPI_IRSEND** — неблокирующая передача сообщения по готовности.

К изложенной выше семантике работы этих процедур добавляется отсутствие блокировки.

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

Неблокирующий прием в буфер **BUF** не более **COUNT** элементов сообщения типа **DATATYPE** с идентификатором **MSGTAG** от процесса с номером **SOURCE** в коммуникаторе **COMM** с заполнением массива **STATUS**. В отличие от блокирующего приема, возврат из процедуры происходит сразу после инициализации процесса приема без ожидания получения всего сообщения и его записи в буфере **BUF**. Окончание процесса приема можно определить с помощью параметра **REQUEST** и процедур семейств **MPI_WAIT** и **MPI_TEST**.

Сообщение, отправленное любой из процедур **MPI_SEND**, **MPI_ISEND** и любой из трех их модификаций, может быть принято любой из процедур **MPI_RECV** и **MPI_IRECV**.

Обратим особое внимание на то, что до завершения неблокирующей операции не следует записывать в используемый массив данных!

```
MPI_IPROBE(SOURCE, MSGTAG, COMM, FLAG, STATUS, IERR)
LOGICAL FLAG
INTEGER SOURCE, MSGTAG, COMM, IERR, STATUS(MPI_STATUS_SIZE)
```

Получение в массиве **STATUS** информации о структуре ожидаемого сообщения с идентификатором **MSGTAG** от процесса с номером **SOURCE** в коммуникаторе **COMM** без блокировки. В параметре **FLAG** возвращается значение **.TRUE.**, если сообщение с подходящими атрибутами уже может быть принято (в этом случае действие процедуры полностью аналогично **MPI_PROBE**), и значение **.FALSE.**, если сообщения с указанными атрибутами еще нет.

```
MPI_WAIT(REQUEST, STATUS, IERR)
INTEGER REQUEST, IERR, STATUS(MPI_STATUS_SIZE)
```

Ожидание завершения асинхронной операции, ассоциированной с идентификатором **REQUEST** и запущенной вызовом процедуры **MPI_ISEND** или **MPI_IRECV**. Пока асинхронная операция не будет завершена, процесс, выполнивший процедуру **MPI_WAIT**, будет заблокирован. Для операции неблокирующего приема определяется параметр **STATUS**. После выполнения

процедуры идентификатор неблокирующей операции **REQUEST** устанавливается в значение **MPI_REQUEST_NULL**.

MPI_WAITALL(COUNT, REQUESTS, STATUSES, IERR)
INTEGER COUNT, REQUESTS(*), STATUSES(MPI_STATUS_SIZE,*), IERR
Ожидание завершения **COUNT** асинхронных операций, ассоциированных с идентификаторами массива **REQUESTS**. Для операций неблокирующих приемов определяются соответствующие параметры в массиве **STATUSES**. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива **STATUSES** будет установлено в соответствующее значение. После выполнения процедуры соответствующие элементы параметра **REQUESTS** устанавливаются в значение **MPI_REQUEST_NULL**.

Ниже показан пример фрагмента программы, в которой все процессы обмениваются сообщениями с ближайшими соседями в соответствии с топологией кольца при помощи неблокирующих операций. Заметим, что использование для этих целей блокирующих операций могло привести к возникновению тупиковой ситуации.

```
      program example9
      include 'mpif.h'
      integer ierr, rank, size, prev, next, reqs(4), buf(2)
      integer stats(MPI_STATUS_SIZE, 4)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      prev = rank - 1
      next = rank + 1
      if (rank .eq. 0) prev = size - 1
      if (rank .eq. size - 1) next = 0
      call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, 5,
     &               MPI_COMM_WORLD, reqs(1), ierr)
      call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, 6,
     &               MPI_COMM_WORLD, reqs(2), ierr)
      call MPI_ISEND(rank, 1, MPI_INTEGER, prev, 6,
     &               MPI_COMM_WORLD, reqs(3), ierr)
      call MPI_ISEND(rank, 1, MPI_INTEGER, next, 5,
     &               MPI_COMM_WORLD, reqs(4), ierr)
      call MPI_WAITALL(4, reqs, stats, ierr);
      print *, 'process ', rank,
     &         ' prev=', buf(1), ' next=', buf(2)
      call MPI_FINALIZE(ierr)
      end
```

MPI_WAITANY(COUNT, REQUESTS, INDEX, STATUS, IERR)
INTEGER COUNT, REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERR
Ожидание завершения одной из **COUNT** асинхронных операций, ассоциированных с идентификаторами **REQUESTS**. Если к моменту вызова завершились

несколько из ожидаемых операций, то случайным образом будет выбрана одна из них. Параметр **INDEX** содержит номер элемента в массиве **REQUESTS**, содержащего идентификатор завершенной операции. Для неблокирующего приема определяется параметр **STATUS**. После выполнения процедуры соответствующий элемент параметра **REQUESTS** устанавливается в значение **MPI_REQUEST_NULL**.

```
MPI_WAITSOME(INCOUNT, REQUESTS, OUTCOUNT, INDEXES, STATUSES,
IERR)
INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDEXES(*), IERR,
STATUSES(MPI_STATUS_SIZE,*)
```

Ожидание завершения хотя бы одной из **INCOUNT** асинхронных операций, ассоциированных с идентификаторами **REQUESTS**. Параметр **OUTCOUNT** содержит число завершенных операций, а первые **OUTCOUNT** элементов массива **INDEXES** содержат номера элементов массива **REQUESTS** с их идентификаторами. Первые **OUTCOUNT** элементов массива **STATUSES** содержат параметры завершенных операций (для неблокирующих приемов). После выполнения процедуры соответствующие элементы параметра **REQUESTS** устанавливаются в значение **MPI_REQUEST_NULL**.

В следующем примере демонстрируется схема использования процедуры **MPI_WAITSOME** для организации коммуникационной схемы "*master-slave*" (все процессы общаются с одним выделенным процессом). Все процессы кроме процесса **0** на каждой итерации цикла определяют с помощью вызова процедуры **slave** свою локальную часть массива **a**, после чего посылают ее главному процессу. Процесс **0** сначала инициализирует неблокирующие приемы от всех остальных процессов, после чего дожидается прихода хотя бы одного сообщения. Для пришедших сообщений процесс **0** вызывает процедуру обработки **master**, после чего снова выставляет неблокирующие приемы. Таким образом, процесс **0** обрабатывает те порции данных, которые готовы на данный момент. При этом для корректности работы программы нужно обеспечить, чтобы процесс **0** успевал обработать приходящие сообщения, то есть, чтобы процедура **slave** работала значительно дольше процедуры **master** (в противном случае и распараллеливание не имеет особого смысла). Кроме того, в примере написан бесконечный цикл, поэтому для конкретной программы нужно предусмотреть условие завершения.

```
      program example10
      include 'mpif.h'
      integer rank, size, ierr, N, MAXPROC
      parameter(N = 1000, MAXPROC = 128)
      integer req(MAXPROC), num, indexes(MAXPROC)
      integer statuses(MPI_STATUS_SIZE, MAXPROC)
      double precision a(N, MAXPROC)

      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      if(rank .ne. 0) then
         do while(.TRUE.)
            call slave(a, N)
            call MPI_SEND(a, N, MPI_DOUBLE_PRECISION, 0, 5,
     &                    MPI_COMM_WORLD, ierr)
         end do
      else
         do i = 1, size-1
            call MPI_IRECV(a(1, i), N, MPI_DOUBLE_PRECISION, i,
     &                     5, MPI_COMM_WORLD, req(i), ierr)
         end do
         do while(.TRUE.)
            call MPI_WAITSOME(size-1, req, num, indexes,
     &                        statuses, ierr)
            do i = 1, num
               call master(a(1, indexes(i)), N)
               call MPI_IRECV(a(1, indexes(i)), N,
     &                        MPI_DOUBLE_PRECISION,
     &                        indexes(i), 5, MPI_COMM_WORLD,
     &                        req(indexes(i)), ierr)
            end do
         end do
      end if
      call MPI_FINALIZE(ierr)
      end

      subroutine slave(a, n)
      double precision a
      integer n
C обработка локальной части массива a
      end

      subroutine master(a, n)
      double precision a
      integer n
C обработка массива a
      End
```

```
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
LOGICAL FLAG
INTEGER REQUEST, IERR, STATUS(MPI_STATUS_SIZE)
```
Проверка завершенности асинхронной операции **MPI_ISEND** или **MPI_IRECV**, ассоциированной с идентификатором **REQUEST**. В параметре **FLAG** возвращается значение **.TRUE.**, если операция завершена, и значение **.FALSE.** - в противном случае (в языке Си – **1** или **0** соответственно). Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра **STATUS**. После выполнения процедуры соответствующий элемент параметра **REQUEST** устанавливается в значение **MPI_REQUEST_NULL**.

```
MPI_TESTALL(COUNT, REQUESTS, FLAG, STATUSES, IERR)
LOGICAL FLAG
INTEGER COUNT, REQUESTS(*), STATUSES(MPI_STATUS_SIZE,*), IERR
```
Проверка завершенности **COUNT** асинхронных операций, ассоциированных с идентификаторами **REQUESTS**. В параметре **FLAG** процедура возвращает значение **.TRUE.** (в языке Си – **1**), если все операции, ассоциированные с указанными идентификаторами, завершены. В этом случае параметры сообщений будут указаны в массиве **STATUSES**. Если какая-либо из операций не завершилась, то возвращается **.FALSE.** (в языке Си – **0**), и определенность элементов массива **STATUSES** не гарантируется. После выполнения процедуры соответствующие элементы параметра **REQUESTS** устанавливаются в значение **MPI_REQUEST_NULL**.

```
MPI_TESTANY(COUNT, REQUESTS, INDEX, FLAG, STATUS, IERR)
LOGICAL FLAG
INTEGER COUNT, REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERR
```
Проверка завершенности хотя бы одной асинхронной операции, ассоциированной с идентификатором из массива **REQUESTS**.В параметре **FLAG** возвращается значение **.TRUE.** (в языке Си – **1**), если хотя бы одна из операций асинхронного обмена завершена, при этом **INDEX** содержит номер соответствующего элемента в массиве **REQUESTS**, а **STATUS** — параметры сообщения. В противном случае в параметре **FLAG** будет возвращено значение **.FALSE.** (в языке Си – **0**). Если к моменту вызова завершились несколько из ожидаемых операций, то случайным образом будет выбрана одна из них. После выполнения процедуры соответствующий элемент параметра **REQUESTS** устанавливается в значение **MPI_REQUEST_NULL**.

```
MPI_TESTSOME(INCOUNT, REQUESTS, OUTCOUNT, INDEXES, STATUSES,
IERR)
INTEGER INCOUNT, REQUESTS(*), OUTCOUNT, INDEXES(*), IERR,
STATUSES(MPI_STATUS_SIZE,*)
```

Аналог процедуры **MPI_WAITSOME**, но возврат происходит немедленно. Если ни одна из тестируемых операций к моменту вызова не завершилась, то значение **OUTCOUNT** будет равно нулю.

Следующий пример демонстрирует применение неблокирующих операций для реализации транспонирования квадратной матрицы, распределенной между процессами по строкам. Сначала каждый процесс локально определяет **nl** строк массива **a**. Затем при помощи неблокирующих операций **MPI_ISEND** и **MPI_IRECV** инициализируются все необходимые для транспонирования обмены данными. На фоне начинающихся обменов каждый процесс транспонирует свою локальную часть массива **a**. После этого процесс при помощи вызова процедуры **MPI_WAITANY** дожидается прихода сообщения от любого другого процесса и транспонирует полученную от данного процесса часть массива **a**. Обработка продолжается до тех пор, пока не будут получены сообщения от всех процессов. В конце исходный массив **a** и транспонированный массив **b** распечатываются.

```
      program example11
      include 'mpif.h'
      integer ierr, rank, size, N, nl, i, j
      parameter (N = 9)
      double precision a(N, N), b(N, N)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      nl = (N-1)/size+1
      call work(a, b, N, nl, size, rank)
      call MPI_FINALIZE(ierr)
      end

      subroutine work(a, b, n, nl, size, rank)
      include 'mpif.h'
      integer ierr, rank, size, n, MAXPROC, nl, i, j, ii, jj, ir
      parameter (MAXPROC = 64)
      double precision a(nl, n), b(nl, n), c
      integer irr, status(MPI_STATUS_SIZE), req(MAXPROC*2)
      do i = 1, nl
         do j = 1, n
            ii = i+rank*nl
            if(ii .le. n) a(i, j) = 100*ii+j
         end do
      end do

      do ir = 0, size-1
         if(ir .ne. rank)
     &      call MPI_IRECV(b(1, ir*nl+1), nl*nl,
```

```
&                     MPI_DOUBLE_PRECISION, ir,
&                     MPI_ANY_TAG, MPI_COMM_WORLD,
&                     req(ir+1), ierr)
 end do

 req(rank+1) = MPI_REQUEST_NULL

 do ir = 0, size-1
    if(ir .ne. rank)
&      call MPI_ISEND(a(1, ir*nl+1), nl*nl,
&                     MPI_DOUBLE_PRECISION, ir,
&                     1, MPI_COMM_WORLD,
&                     req(ir+1+size), ierr)
 end do

 ir = rank
 do i = 1, nl
    ii = i+ir*nl
    do j = i+1, nl
       jj = j+ir*nl
       b(i, jj) = a(j, ii)
       b(j, ii) = a(i, jj)
    end do
    b(i, ii) = a(i, ii)
 end do

 do irr = 1, size-1
    call MPI_WAITANY(size, req, ir, status, ierr)
    ir = ir-1
    do i = 1, nl
       ii = i+ir*nl
       do j = i+1, nl
          jj = j+ir*nl
          c = b(i, jj)
          b(i, jj) = b(j, ii)
          b(j, ii) = c
       end do
    end do
 end do

 do i = 1, nl
    do j = 1, N
       ii = i+rank*nl
       if(ii .le. n) print *, 'process ', rank,
&                    ': a(', ii, ', ', j, ') =', a(i,j),
&                    ', b(', ii, ', ', j, ') =', b(i,j)
    end do
 end do
 end
```

Отложенные запросы на взаимодействие

Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером. Часто в программе приходится многократно выполнять обмены с одинаковыми параметрами (например, в цикле). В этом случае можно один раз инициализировать операцию обмена и потом многократно ее запускать, не тратя на каждой итерации дополнительного времени на инициализацию и заведение соответствующих внутренних структур данных. Кроме того, таким образом несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой (впрочем, это совсем необязательно хорошо, поскольку может привести к перегрузке коммуникационной сети).

Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью отложенных запросов либо обычным способом, может быть принято как обычным способом, так и с помощью отложенных запросов.

```
MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, MSGTAG, COMM, REQUEST,
IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, MSGTAG, COMM, REQUEST, IERR
```

Формирование *отложенного запроса* на посылку сообщения. Сама операция пересылки при этом не начинается!

Аналогично трем модификациям процедур **MPI_SEND** и **MPI_ISEND**, предусмотрены три дополнительных варианта процедуры **MPI_SEND_INIT**:

- **MPI_BSEND_INIT** — формирование отложенного запроса на передачу сообщения с буферизацией;
- **MPI_SSEND_INIT** — формирование отложенного запроса на передачу сообщения с синхронизацией;
- **MPI_RSEND_INIT** — формирование отложенного запроса на передачу сообщения по готовности.

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, MSGTAG, COMM,
REQUEST, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, MSGTAG, COMM, REQUEST, IERR
```

Формирование отложенного запроса на прием сообщения. Сама операция приема при этом не начинается!

```
MPI_START(REQUEST, IERR)
INTEGER REQUEST, IERR
```
Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра **REQUEST**. Операция запускается как неблокирующая.

```
MPI_STARTALL(COUNT, REQUESTS, IERR)
INTEGER COUNT, REQUESTS, IERR
```
Инициализация **COUNT** отложенных запросов на выполнение операций обмена, соответствующих значениям первых **COUNT** элементов массива **REQUESTS**. Операции запускаются как неблокирующие.

В отличие от неблокирующих операций, по завершении выполнения операции, запущенной при помощи отложенного запроса на взаимодействие, значение параметра **REQUEST** (**REQUESTS**) сохраняется и может использоваться в дальнейшем!

```
MPI_REQUEST_FREE(REQUEST, IERR)
INTEGER REQUEST, IERR
```
Данная процедура удаляет структуры данных, связанные с параметром **REQUEST**. После ее выполнения параметр **REQUEST** устанавливается в значение **MPI_REQUEST_NULL**. Если операция, связанная с этим запросом, уже выполняется, то она будет завершена.

В следующем примере инициализируются отложенные запросы на операции двунаправленного обмена с соседними процессами в кольцевой топологии. Сами операции запускаются на каждой итерации последующего цикла. По завершении цикла отложенные запросы удаляются.

```
 prev = rank - 1
 next = rank + 1
 if(rank .eq. 0) prev = size - 1
 if(rank .eq. size - 1) next = 0
 call MPI_RECV_INIT(rbuf(1), 1, MPI_REAL, prev, 5,
&                   MPI_COMM_WORLD, reqs(1), ierr)
 call MPI_RECV_INIT(rbuf(2), 1, MPI_REAL, next, 6,
&                   MPI_COMM_WORLD, reqs(2), ierr)
 call MPI_SEND_INIT(sbuf(1), 1, MPI_REAL, prev, 6,
&                   MPI_COMM_WORLD, reqs(3), ierr)
 call MPI_SEND_INIT(sbuf(2), 1, MPI_REAL, next, 5,
&                   MPI_COMM_WORLD, reqs(4), ierr)
 do i=...
    sbuf(1)=...
    sbuf(2)=...
    call MPI_STARTALL(4, reqs, ierr)
    ...
    call MPI_WAITALL(4, reqs, stats, ierr);
    ...
 end do
```

```
call MPI_REQUEST_FREE(reqs(1), ierr)
call MPI_REQUEST_FREE(reqs(2), ierr)
call MPI_REQUEST_FREE(reqs(3), ierr)
call MPI_REQUEST_FREE(reqs(4), ierr)
```

Тупиковые ситуации (deadlock)

Использование блокирующих процедур приема и посылки связано с возможным возникновением *тупиковой ситуации*. Предположим, что работают два параллельных процесса, и они должны обменяться данными. Было бы вполне естественно в каждом процессе сначала воспользоваться процедурой **MPI_SEND**, а затем процедурой **MPI_RECV**. Но именно этого и не стоит делать. Дело в том, что мы заранее не знаем, как реализована процедура **MPI_SEND**. Если разработчики для гарантии корректного повторного использования буфера посылки заложили схему, при которой посылающий процесс ждет начала приема, то возникнет классический тупик. Первый процесс не может вернуться из процедуры посылки, поскольку второй не начинает прием сообщения. А второй процесс не может начать прием сообщения, поскольку сам по похожей причине застрял на посылке.

Еще хуже ситуация, когда оба процесса сначала попадают на блокирующую процедуру приема **MPI_RECV**, а лишь затем на посылку данных. В таком случае тупик возникнет независимо от способа реализации процедур приема и посылки данных.

процесс **0**:	процесс **1**:
MPI_RECV от процесса 1 **MPI_SEND процессу 1**	**MPI_RECV от процесса 0** **MPI_SEND процессу 0**

Возникает тупик!

процесс **0**:	процесс **1**:
MPI_SEND процессу 1 **MPI_RECV от процесса 1**	**MPI_SEND процессу 0** **MPI_RECV от процесса 0**

Может возникнуть тупик!

Рассмотрим различные способы разрешения тупиковых ситуаций.

1. Простейшим вариантом разрешения тупиковой ситуации будет изменение порядка следования процедур посылки и приема сообщения на оном из процессов, как показано ниже.

процесс **0**:	процесс **1**:
MPI_SEND процессу 1 **MPI_RECV от процесса 1**	**MPI_RECV от процесса 0** **MPI_SEND процессу 0**

Тупик не возникает!

2. Другим вариантом разрешения тупиковой ситуации может быть использование неблокирующих операций. Заменим вызов процедуры приема сообщения с блокировкой на вызов процедуры **MPI_IRECV**. Расположим его перед вызовом процедуры **MPI_SEND**, т.е. преобразуем фрагмент следующим образом:

Процесс **0**:	процесс **1**:
MPI_SEND процессу 1 **MPI_RECV от процесса 1**	**MPI_IRECV от процесса 0** **MPI_SEND процессу 0** **MPI_WAIT**

Тупик не возникает!

В такой ситуации тупик гарантированно не возникнет, поскольку к моменту вызова процедуры **MPI_SEND** запрос на прием сообщения уже будет выставлен, а значит, передача данных может начаться. При этом рекомендуется выставлять процедуру **MPI_IRECV** в программе как можно раньше, чтобы раньше предоставить возможность начала пересылки и максимально использовать преимущества асинхронности.

3. Третьим вариантом разрешения тупиковой ситуации может быть использование процедуры **MPI_SENDRECV**.

```
MPI_SENDRECV(SBUF, SCOUNT, STYPE, DEST, STAG, RBUF, RCOUNT,
RTYPE, SOURCE, RTAG, COMM, STATUS, IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, DEST, STAG, RCOUNT, RTYPE, SOURCE,
RTAG, COMM, STATUS(MPI_STATUS_SIZE), IERR
```

Процедура выполняет совмещенные прием и передачу сообщений с блокировкой. По вызову данной процедуры осуществляется посылка **SCOUNT** элементов типа **STYPE** из массива **SBUF** с тегом **STAG** процессу с номером **DEST** в коммуникаторе **COMM** и прием в массив **RBUF** не более **RCOUNT** элементов типа **RTYPE** с тегом **RTAG** от процесса с номером **SOURCE** в коммуникаторе **COMM**. Для принимаемого сообщения заполняется параметр **STATUS**. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Буферы передачи и приема данных не должны пересекаться. Гарантируется, что при этом тупиковой ситуации не возникает. Сообщение, отправленное операцией **MPI_SENDRECV**, может быть принято обычным образом, и операция **MPI_SENDRECV** может принять сообщение, отправленное обычной операцией.

```
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, STAG, SOURCE,
RTAG, COMM, STATUS, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, STAG, SOURCE, RTAG, COMM,
STATUS(MPI_STATUS_SIZE), IERR
```

Совмещенные прием и передача сообщений с блокировкой через общий буфер **BUF**. Принимаемое сообщение не должно превышать по размеру отправляемое сообщение, а передаваемые и принимаемые данные должны быть одного типа.

В следующем примере операции двунаправленного обмена с соседними процессами в кольцевой топологии производятся при помощи двух вызовов процедуры **MPI_SENDRECV**. При этом гарантированно не возникает тупиковой ситуации.

```
      program example12
      include 'mpif.h'
      integer ierr, rank, size, prev, next, buf(2)
      integer status1(MPI_STATUS_SIZE), status2(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      prev = rank - 1
      next = rank + 1
      if(rank .eq. 0) prev = size - 1
      if(rank .eq. size - 1) next = 0
      call MPI_SENDRECV(rank, 1, MPI_INTEGER, prev, 6,
     &                  buf(2), 1, MPI_INTEGER, next, 6,
     &                  MPI_COMM_WORLD, status2, ierr)
      call MPI_SENDRECV(rank, 1, MPI_INTEGER, next, 5,
     &                  buf(1), 1, MPI_INTEGER, prev, 5,
     &                  MPI_COMM_WORLD, status1, ierr)
      print *, 'process ', rank,
     &         ' prev=', buf(1), ' next=', buf(2)
      call MPI_FINALIZE(ierr)
      end
```

Задания

- Какими атрибутами обладает в MPI каждое посылаемое сообщение?
- Можно ли сообщение, отправленное с помощью блокирующей операции посылки, принять неблокирующей операцией приема?
- Что гарантирует блокировка при отправке/приеме сообщений?
- Можно ли в качестве тегов при посылке различных сообщений в программе всегда использовать одно и то же число?
- Как принять любое сообщение от любого процесса?

- Как принимающий процесс может определить длину полученного сообщения?
- Можно ли при посылке сообщения использовать константы **MPI_ANY_SOURCE** и **MPI_ANY_TAG**?
- Можно ли, не принимая сообщения, определить его атрибуты?
- Будет ли корректна программа, в которой посылающий процесс указывает в качестве длины буфера число **10**, а принимающий процесс - число **20**? Если да, то сколько элементов массива будет реально переслано между процессами?
- Сравнить эффективность реализации различных видов пересылок данных с блокировкой (**MPI_SEND**, **MPI_BSEND**, **MPI_SSEND**, **MPI_RSEND**) между двумя выделенными процессорами.
- Что означает завершение операции для различных видов пересылки данных с блокировкой?
- Определить максимально допустимую длину посылаемого сообщения в данной реализации MPI.
- Реализовать скалярное произведение распределенных между процессорами векторов.
- Сравнить эффективность реализации пересылок данных между двумя выделенными процессорами с блокировкой и без блокировки.
- Определить, возможно ли в данной реализации MPI совмещение асинхронных пересылок данных и выполнения арифметических операций.
- Как с помощью процедуры **MPI_TEST** смоделировать функциональность процедуры **MPI_WAIT**?
- В чем состоят различия в использовании процедур **MPI_WAITALL**, **MPI_WAITANY** и **MPI_WAITSOME**? Как смоделировать их функциональность при помощи процедуры **MPI_WAIT**?
- Что произойдет при осуществлении обмена данными с процессом **MPI_PROC_NULL**?
- Реализовать при помощи посылки сообщений типа точка-точка следующие схемы коммуникации процессов:
    - передача данных по кольцу, два варианта: "эстафетная палочка" (очередной процесс дожидается сообщения от предыдущего и потом посылает следующему) и "сдвиг" (одновременные посылка и прием сообщений);
    - master-slave (все процессы общаются с одним выделенным процессом);
    - пересылка данных от каждого процесса каждому.
- Исследовать эффективность коммуникационных схем из предыдущего задания в зависимости от числа использованных процессов и объема пересылаемых данных, изучить возможности оптимизации.

- Определить выигрыш, который можно получить при использовании отложенных запросов на взаимодействие.
- Сравнить эффективность реализации функции **MPI_SENDRECV** с моделированием той же функциональности при помощи неблокирующих операций.

Коллективные взаимодействия процессов

В *операциях коллективного взаимодействия* процессов участвуют все процессы коммуникатора. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки. Асинхронных коллективных операций в MPI нет.

В коллективных операциях можно использовать те же коммуникаторы, что и были использованы для операций типа точка-точка. MPI гарантирует, что сообщения, вызванные коллективными операциями, никак не повлияют на выполнение других операций и не пересекутся с сообщениями, появившимися в результате индивидуального взаимодействия процессов.

Вообще говоря, нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций (кроме процедуры **MPI_BARRIER**). Если какой-то процесс завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (если это возможно по смыслу операции).

В коллективных операциях не используются идентификаторы сообщений (теги). Таким образом, коллективные операции строго упорядочены согласно их появлению тексте программы.

```
MPI_BARRIER(COMM, IERR)
INTEGER COMM, IERR
```

Процедура используется для барьерной синхронизации процессов. Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора **COMM** не выполнят эту процедуру. Только после того, как последний процесс коммуникатора выполнит данную процедуру, все процессы будут разблокированы и продолжат выполнение дальше. Данная процедура является коллективной. Все процессы должны вызвать

MPI_BARRIER, хотя реально исполненные вызовы различными процессами коммуникатора могут быть расположены в разных местах программы.

В следующем примере функциональность процедуры **MPI_BARRIER** моделируется при помощи отложенных запросов на взаимодействие. Для усреднения результатов производится **NTIMES** операций обмена, в рамках каждой из них все процессы должны послать сообщение процессу с номером **0**, после чего получить от него ответный сигнал, означающий, что все процессы дошли до этой точки в программе. Использование отложенных запросов позволяет инициализировать посылку данных только один раз, а затем использовать на каждой итерации цикла. Далее время на моделирование сравнивается со временем на синхронизацию при помощи самой стандартной процедуры **MPI_BARRIER**.

```
      program example13
      include 'mpif.h'
      integer ierr, rank, size, MAXPROC, NTIMES, i, it
      parameter (MAXPROC = 128, NTIMES = 10000)
      integer ibuf(MAXPROC)
      double precision time_start, time_finish
      integer req(2*MAXPROC), statuses(MPI_STATUS_SIZE, MAXPROC)
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      if(rank .eq. 0) then
         do i = 1, size-1
            call MPI_RECV_INIT(ibuf(i), 0, MPI_INTEGER, i, 5,
     &                         MPI_COMM_WORLD, req(i), ierr)
            call MPI_SEND_INIT(rank, 0, MPI_INTEGER, i, 6,
     &                         MPI_COMM_WORLD, req(size+i),
     &                         ierr)
         end do
         time_start = MPI_WTIME(ierr)
         do it = 1, NTIMES
            call MPI_STARTALL(size-1, req, ierr)
            call MPI_WAITALL(size-1, req, statuses, ierr)
            call MPI_STARTALL(size-1, req(size+1), ierr)
            call MPI_WAITALL(size-1, req(size+1), statuses,
     &                       ierr)
         end do
       else
         call MPI_RECV_INIT(ibuf(1), 0, MPI_INTEGER, 0, 6,
     &                      MPI_COMM_WORLD, req(1), ierr)
         call MPI_SEND_INIT(rank, 0, MPI_INTEGER, 0, 5,
     &                      MPI_COMM_WORLD, req(2), ierr)
         time_start = MPI_WTIME(ierr)
         do it = 1, NTIMES
            call MPI_START(req(2), ierr)
            call MPI_WAIT(req(2), statuses, ierr)
            call MPI_START(req(1), ierr)
```

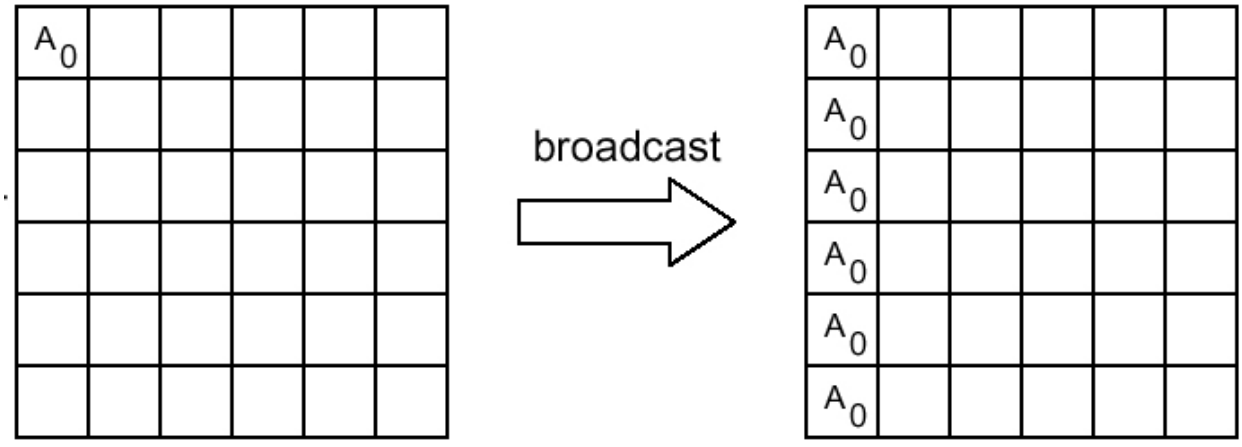
```
            call MPI_WAIT(req(1), statuses, ierr)
         end do
       end if
       time_finish = MPI_WTIME(ierr)-time_start
       print *, 'rank = ', rank, ' all time = ',
     &         (time_finish)/NTIMES

       time_start = MPI_WTIME(ierr)
       do it = 1, NTIMES
          call MPI_BARRIER(MPI_COMM_WORLD,ierr)
       enddo
       time_finish = MPI_WTIME(ierr)-time_start
       print *, 'rank = ', rank, ' barrier time = ',
     &         (time_finish)/NTIMES
       call MPI_FINALIZE(ierr)
       end
```

```
MPI_BCAST(BUF, COUNT, DATATYPE, ROOT, COMM, IERR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERR
```

Рассылка **COUNT** элементов данных типа **DATATYPE** из массива **BUF** от процесса **ROOT** всем процессам данного коммуникатора **COMM**, включая сам рассылающий процесс. При возврате из процедуры содержимое буфера **BUF** процесса **ROOT** будет скопировано в локальный буфер каждого процесса коммуникатора **COMM**. Значения параметров **COUNT**, **DATATYPE**, **ROOT** и **COMM** должны быть одинаковыми у всех процессов.

Следующая схема иллюстрирует действие процедуры **MPI_BCAST**. Здесь, также как и в дальнейших схемах по вертикали изображаются разные процессы, участвующие в коллективной операции, а по горизонтали – расположенные на них блоки данных.



Например, для того чтобы переслать от процесса **2** всем остальным процессам приложения массив **buf** из **100** целочисленных элементов, нужно, чтобы во всех процессах встретился следующий вызов:

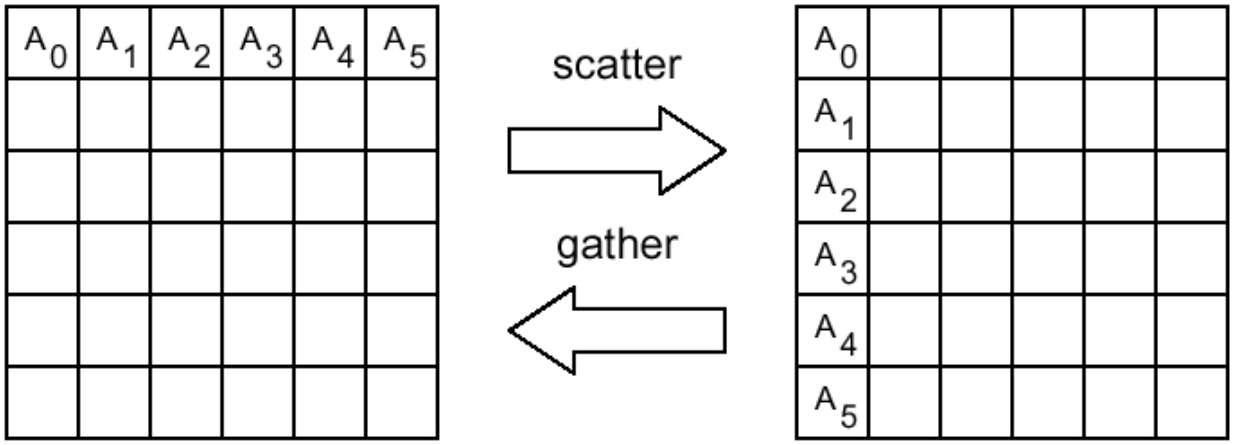
```
      call MPI_BCAST(buf, 100, MPI_INTEGER,
     &               2, MPI_COMM_WORLD, ierr)
```

```
MPI_GATHER(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, ROOT, COMM,
IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, RCOUNT, RTYPE, ROOT, COMM, IERR
```

Сборка **SCOUNT** элементов данных типа **STYPE** из массивов **SBUF** со всех процессов коммуникатора **COMM** в буфере **RBUF** процесса **ROOT**. Каждый процесс, включая **ROOT**, посылает содержимое своего буфера **SBUF** процессу **ROOT**. Собирающий процесс сохраняет данные в буфере **RBUF**, располагая их в порядке возрастания номеров процессов.

На процессе **ROOT** существенными являются значения всех параметров, а на остальных процессах — только значения параметров **SBUF**, **SCOUNT**, **STYPE**, **ROOT** и **COMM**. Значения параметров **ROOT** и **COMM** должны быть одинаковыми у всех процессов. Параметр **RCOUNT** у процесса **ROOT** обозначает число элементов типа **RTYPE**, принимаемых не от всех процессов в сумме, а от каждого процесса.

Следующая схема иллюстрирует действие процедуры **MPI_GATHER**.



Например, для того чтобы процесс **2** собрал в массив **rbuf** по **10** целочисленных элементов массивов **buf** со всех процессов приложения, нужно, чтобы во всех процессах встретился следующий вызов:

```
      call MPI_GATHER(buf, 10, MPI_INTEGER,
     &                rbuf, 10, MPI_INTEGER,
     &                2, MPI_COMM_WORLD, ierr)
```

```
MPI_GATHERV(SBUF, SCOUNT, STYPE, RBUF, RCOUNTS, DISPLS, RTYPE,
ROOT, COMM, IERR)
<type> SBUF(*), RBUF(*)
```

```
INTEGER SCOUNT, STYPE, RCOUNTS(*), DISPLS(*), RTYPE, ROOT, COMM,
IERR
```
Сборка различного количества данных из массивов **SBUF**. Порядок расположения данных в результирующем буфере **RBUF** задает массив **DISPLS**.

RCOUNTS – целочисленный массив, содержащий количество элементов, передаваемых от каждого процесса (индекс равен рангу посылающего процесса, размер массива равен числу процессов в коммуникаторе **COMM**).

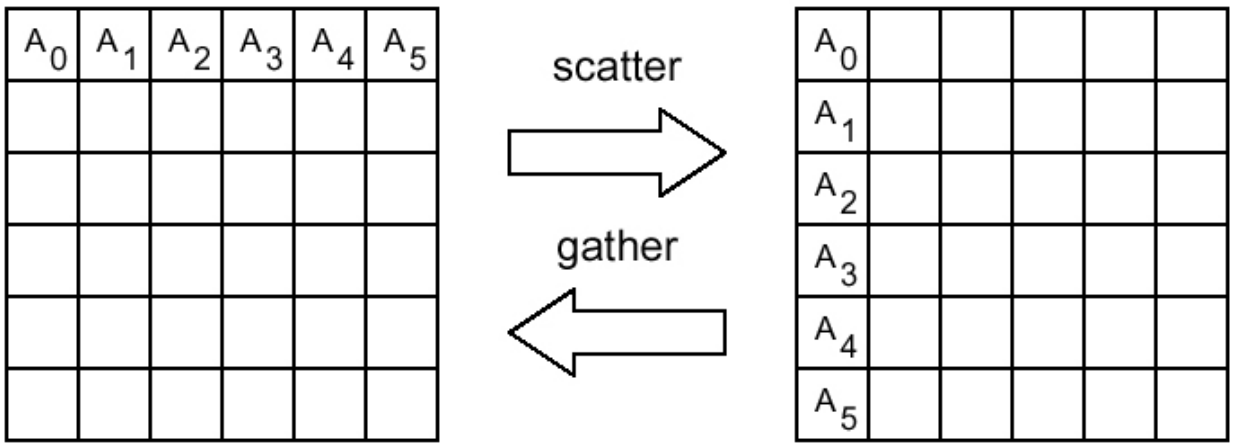
DISPLS – целочисленный массив, содержащий смещения относительно начала массива **RBUF** (индекс равен рангу посылающего процесса, размер массива равен числу процессов в коммуникаторе **COMM**).

Данные, посланные процессом **J-1**, размещаются в **J**-ом блоке буфера **RBUF** на процессе **ROOT**, который начинается со смещением в **DISPLS(J)** элементов типа **RTYPE** с начала буфера.

```
MPI_SCATTER(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, ROOT,
COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, RCOUNT, RTYPE, ROOT, COMM, IERR
```
Процедура **MPI_SCATTER** по своему действию является обратной к **MPI_GATHER**. Она осуществляет рассылку по **SCOUNT** элементов данных типа **STYPE** из массива **SBUF** процесса **ROOT** в массивы **RBUF** всех процессов коммуникатора **COMM**, включая сам процесс **ROOT**. Можно считать, что массив **SBUF** делится на равные части по числу процессов, каждая из которых состоит из **SCOUNT** элементов типа **STYPE**, после чего **I**-я часть посылается (**I-1**)-му процессу.

На процессе **ROOT** существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров **RBUF**, **RCOUNT**, **RTYPE**, **SOURCE** и **COMM**. Значения параметров **SOURCE** и **COMM** должны быть одинаковыми у всех процессов.

Следующая схема иллюстрирует действие процедуры **MPI_SCATTER**.



В следующем примере процесс **0** определяет массив **sbuf**, после чего рассылает его по одному столбцу всем запущенным процессам приложения. Результат на каждом процессе располагается в массиве **rbuf**.

```
      real sbuf(size, size), rbuf(size)
      if(rank .eq. 0) then
         do 1 i = 1, size
            do 1 j = 1, size
 1             sbuf(i, j)=...
      end if
      if (numtasks .eq. size) then
         call MPI_SCATTER(sbuf, size, MPI_REAL,
     &                    rbuf, size, MPI_REAL,
     &                    0, MPI_COMM_WORLD, ierr)
      end if
```

```
MPI_SCATTERV(SBUF, SCOUNTS, DISPLS, STYPE, RBUF, RCOUNT, RTYPE,
ROOT, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNTS(*), DISPLS(*), STYPE, RCOUNT, RTYPE, ROOT, COMM,
IERR
```

Рассылка различного количества данных из массива **SBUF**. Начало порций рассылаемых данных задает массив **DISPLS**.

SCOUNTS – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе **COMM**).

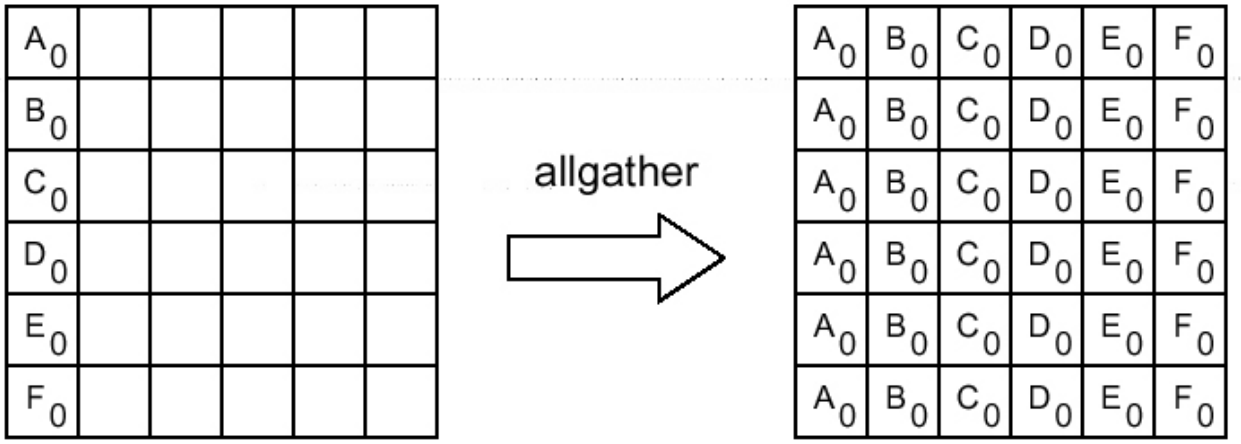
DISPLS – целочисленный массив, содержащий смещения относительно начала массива **SBUF** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе **COMM**).

Данные, посылаемые процессом **ROOT** процессу **J-1**, размещены в **J**-ом блоке буфера **SBUF**, который начинается со смещением в **DISPLS(J)** элементов типа **STYPE** с начала буфера **SBUF**.

```
MPI_ALLGATHER(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, COMM,
IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, RCOUNT, RTYPE, COMM, IERR
```

Сборка данных из массивов **SBUF** со всех процессов коммуникатора **COMM** в буфере **RBUF** каждого процесса. Данные сохраняются в порядке возрастания номеров процессов. Блок данных, посланный процессом **J-1**, размещается в **J**-ом блоке буфера **RBUF** принимающего процесса. Операцию можно рассматривать как **MPI_GATHER**, при которой результат получается на всех процессах коммуникатора **COMM**.

Следующая схема иллюстрирует действие процедуры **MPI_ALLGATHER**.



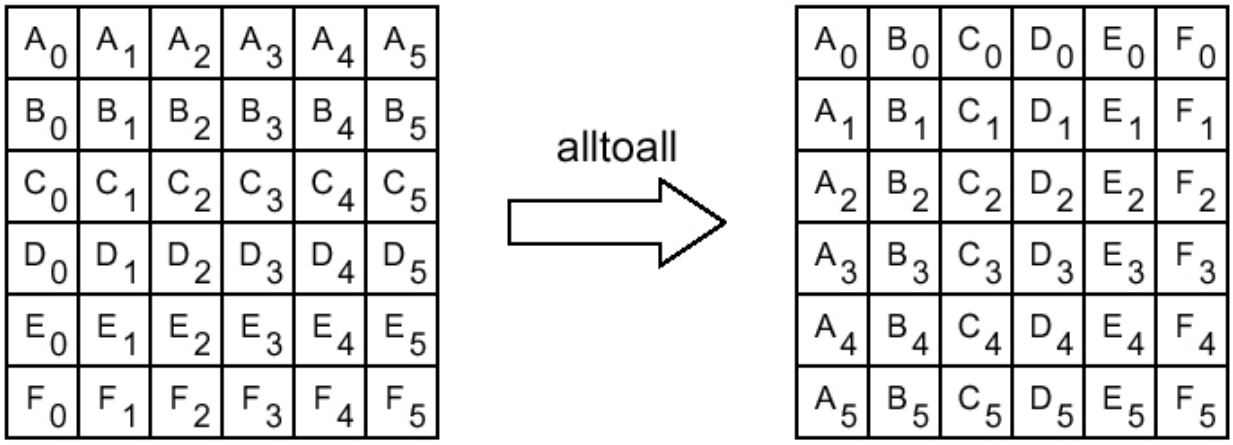

```
MPI_ALLGATHERV(SBUF, SCOUNT, STYPE, RBUF, RCOUNTS, DISPLS,
RTYPE, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, RCOUNTS(*), DISPLS(*), RTYPE, COMM, IERR
```

Сборка на всех процессах коммуникатора **COMM** различного количества данных из массивов **SBUF**. Порядок расположения данных в массиве **RBUF** задает массив **DISPLS**.

```
MPI_ALLTOALL(SBUF, SCOUNT, STYPE, RBUF, RCOUNT, RTYPE, COMM,
IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNT, STYPE, RCOUNT, RTYPE, COMM, IERR
```

Рассылка каждым процессом коммуникатора **COMM** различных порций данных всем другим процессам. **J**-й блок данных буфера **SBUF** (**I-1**)-го процесса попадает в **I**-й блок данных буфера **RBUF** (**J-1**)-го процесса.

Следующая схема иллюстрирует действие процедуры **MPI_ALLTOALL**.




```
MPI_ALLTOALLV(SBUF, SCOUNTS, SDISPLS, STYPE, RBUF, RCOUNTS,
RDISPLS, RTYPE, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER SCOUNTS(*), SDISPLS(*), STYPE, RCOUNTS(*), RDISPLS(*),
RTYPE, COMM, IERR
```

Рассылка со всех процессов коммуникатора **COMM** различного количества данных всем процессам данного коммуникатора. Размещение данных в буфере **SBUF** отсылающего процесса определяется массивом **SDISPLS**, а размещение данных в буфере **RBUF** принимающего процесса определяется массивом **RDISPLS**.

```
MPI_REDUCE(SBUF, RBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERR
```

Выполнение **COUNT** независимых глобальных операций **OP** над соответствующими элементами массивов **SBUF**. Результат выполнения операции **OP** над **I**-ми элементами массивов **SBUF** всех процессов коммуникатора **COMM** получается в **I**-ом элементе массива **RBUF** процесса **ROOT**.

В MPI предусмотрен ряд предопределенных глобальных операций, они задаются следующими константами:

- **MPI_MAX**, **MPI_MIN** – определение максимального и минимального значения;
- **MPI_MINLOC**, **MPI_MAXLOC**– определение максимального и минимального значения и их местоположения;
- **MPI_SUM**, **MPI_PROD** – вычисление глобальной суммы и глобального произведения;
- **MPI_LAND**, **MPI_LOR**, **MPI_LXOR** – логические “И”, “ИЛИ”, исключающее “ИЛИ”;
- **MPI_BAND**, **MPI_BOR**, **MPI_BXOR** – побитовые “И”, “ИЛИ”, исключающее “ИЛИ”.

Кроме того, программист может задать свою функцию для выполнения глобальной операции при помощи процедуры **MPI_OP_CREATE**.

В следующем примере операция глобального суммирования моделируется при помощи схемы сдваивания с использованием пересылок данных типа точка-точка. Эффективность такого моделирования сравнивается с использованием коллективной операции **MPI_REDUCE**.

```
 program example14
 include 'mpif.h'
 integer ierr, rank, i, size, n, nproc
 parameter (n = 1 000 000)
 double precision time_start, time_finish
 double precision a(n), b(n), c(n)
 integer status(MPI_STATUS_SIZE)
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 nproc = size
 do i = 1, n
    a(i) = 1.d0/size
 end do
 call MPI_BARRIER(MPI_COMM_WORLD, ierr)
 time_start = MPI_WTIME(ierr)
 do i = 1, n
    c(i) = a(i)
 end do
 do while (nproc .gt. 1)
    if(rank .lt. nproc/2) then
       call MPI_RECV(b, n, MPI_DOUBLE_PRECISION,
&                    nproc-rank-1, 1, MPI_COMM_WORLD,
&                    status, ierr)
       do i = 1, n
          c(i) = c(i) + b(i)
       end do
    else if(rank .lt. nproc) then
       call MPI_SEND(c, n, MPI_DOUBLE_PRECISION,
&                    nproc-rank-1, 1, MPI_COMM_WORLD, ierr)
    end if
    nproc = nproc/2
 end do
 do i = 1, n
    b(i) = c(i)
 end do
 time_finish = MPI_WTIME(ierr)-time_start
 if(rank .eq. 0) print *, 'model b(1)=', b(1)
 print *, 'rank=', rank, ' model time =', time_finish

 do i = 1, n
    a(i) = 1.d0/size
 end do
 call MPI_BARRIER(MPI_COMM_WORLD, ierr)
```

```
      time_start = MPI_WTIME(ierr)
      call MPI_REDUCE(a, b, n, MPI_DOUBLE_PRECISION, MPI_SUM, 0,
     &                MPI_COMM_WORLD, ierr)
      time_finish = MPI_WTIME(ierr)-time_start
      if(rank .eq. 0) print *, 'reduce b(1)=', b(1)
      print *, 'rank=', rank, ' reduce time =', time_finish
      call MPI_FINALIZE(ierr)
      end
```

```
MPI_ALLREDUCE(SBUF, RBUF, COUNT, DATATYPE, OP, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERR
```

Выполнение **COUNT** независимых глобальных операций **OP** над соответствующими элементами массивов **SBUF**. Отличие от процедуры **MPI_REDUCE** в том, что результат получается в массиве **RBUF** каждого процесса.

В следующем примере каждый процесс вычисляет построчные суммы элементов локального массива **a**, после чего полученные суммы со всех процессов складываются при помощи процедуры **MPI_ALLREDUCE**, и результат получается в массиве **r** на всех процессах приложения.

```
      do i = 1, n
         s(i) = 0.0
      end do
      do i = 1, n
         do j = 1, m
            s(i) = s(i)+a(i, j)
         end do
      end do
      call MPI_ALLREDUCE(s, r, n, MPI_REAL, MPI_SUM,
     &                   MPI_COMM_WORLD, IERR)
```

```
MPI_REDUCE_SCATTER(SBUF, RBUF, RCOUNTS, DATATYPE, OP, COMM,
IERR)
<type> SBUF(*), RBUF(*)
INTEGER RCOUNTS(*), DATATYPE, OP, COMM, IERR
```

Выполнение $\sum_{\mathbf{I}}$ **RCOUNTS(I)** независимых глобальных операций **OP** над соответствующими элементами массивов **SBUF**. Функционально это эквивалентно тому, что сначала выполняются глобальные операции, затем результат рассылается по процессам. **I**-ый процесс получает **(I+1)**-ую порцию результатов из **RCOUNTS(I+1)** элементов и помещает в массив **RBUF**. Массив **RCOUNTS** должен быть одинаковым на всех процессах коммуникатора **COMM**.

```
MPI_SCAN(SBUF, RBUF, COUNT, DATATYPE, OP, COMM, IERR)
<type> SBUF(*), RBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERR
```

Выполнение **COUNT** независимых частичных глобальных операций **OP** над соответствующими элементами массивов **SBUF**. **I**-ый процесс выполняет **COUNT** глобальных операций над соответствующими элементами массива **SBUF** процессов с номерами от **0** до **I** включительно и помещает полученный результат в массив **RBUF**. Полный результат глобальной операции получается в массиве **RBUF** последнего процесса.

```
MPI_OP_CREATE(FUNC, COMMUTE, OP, IERR)
EXTERNAL FUNC
LOGICAL COMMUTE
INTEGER OP, IERR
```

Создание пользовательской глобальной операции **OP**, которая будет вычисляться функцией **FUNC**. Создаваемая операция должна быть ассоциативной, а если параметр **COMMUTE** равен **.TRUE.**, то она должна быть также и коммутативной. Если параметр **COMMUTE** равен **.FALSE.**, то порядок выполнения глобальной операции строго фиксируется согласно увеличению номеров процессов, начиная с процесса с номером **0**.

```
FUNCTION FUNC(INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

Таким образом задается интерфейс пользовательской функции для создания глобальной операции. Первый аргумент операции берется из параметра **INVEC**, второй аргумент – из параметра **INOUTVEC**, а результат возвращается в параметре **INOUTVEC**. Параметр **LEN** задает количество элементов входного и выходного массивов, а параметр **TYPE** – тип входных и выходных данных. В пользовательской функции не должны производиться никакие обмены данными с использованием вызовов процедур MPI.

```
MPI_OP_FREE(OP, IERR)
INTEGER OP, IERR
```

Уничтожение пользовательской глобальной операции. По выполнении процедуры переменной **OP** присваивается значение **MPI_OP_NULL**.

Следующий пример демонстрирует задание пользовательской функции для использования в качестве глобальной операции. Задается функция **smod5**, вычисляющая поэлементную сумму по модулю **5** векторов целочисленных аргументов. Данная функция объявляется в качестве глобальной операции **op** в вызове процедуры **MPI_OP_CREATE**, затем используется в процедуре **MPI_REDUCE**, после чего удаляется с помощью вызова процедуры **MPI_OP_FREE**.

```
      program example15
```

```
 include 'mpif.h'
 integer ierr, rank, i, n
 parameter (n = 1 000)
 integer a(n), b(n)
 integer op
 external smod5
 call MPI_INIT(ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 do i = 1, n
    a(i) = i + rank
 end do
 print *, 'process ', rank, ' a(1) =', a(1)
 call MPI_OP_CREATE(smod5, .TRUE., op, ierr)
 call MPI_REDUCE(a, b, n, MPI_INTEGER, op, 0,
&                MPI_COMM_WORLD, ierr)
 call MPI_OP_FREE(op, ierr)
 if(rank .eq. 0) print *, ' b(1) =', b(1)
 call MPI_FINALIZE(ierr)
 end

 integer function smod5(in, inout, l, type)
 integer l, type
 integer in(l), inout(l), i
 do i = 1, l
    inout(i) = mod(in(i)+inout(i), 5)
 end do
 return
 end
```

Задания

- Чем коллективные операции отличаются от взаимодействий типа точка-точка?
- Верно ли, что в коллективных взаимодействиях участвуют все процессы приложения?
- Могут ли возникать конфликты между обычными сообщениями, посылаемыми процессами друг другу, и сообщениями коллективных операций? Если да, как они разрешаются?
- Можно ли при помощи процедуры **MPI_RECV** принять сообщение, посланное процедурой **MPI_BCAST**?
- Смоделировать барьерную синхронизацию при помощи пересылок точка-точка и сравнить эффективность такой реализации и стандартной процедуры **MPI_BARRIER**.
- В чем различие в функциональности процедур **MPI_BCAST** и **MPI_SCATTER**?
- Смоделировать глобальное суммирование методом сдваивания и сравнить эффективность такой реализации с использованием стандартной процедуры **MPI_REDUCE**.

- Смоделировать процедуру **MPI_ALLREDUCE** при помощи процедур **MPI_REDUCE** и **MPI_BCAST**.
- Напишите свой вариант процедуры **MPI_GATHER**, используя функции посылки сообщений типа точка-точка.
- Подумайте, как организовать коллективный асинхронный обмен данными, аналогичный функции: а) **MPI_REDUCE**; б) **MPI_ALLTOALL**.
- Исследовать масштабируемость (зависимость времени выполнения от числа процессов) различных коллективных операций на конкретной системе.

Группы и коммуникаторы

В MPI существуют широкие возможности для операций над группами процессов и коммуникаторами. Это бывает необходимо, во-первых, чтобы дать возможность некоторой группе процессов работать над своей независимой подзадачей. Во-вторых, если особенность алгоритма такова, что только часть процессов должна обмениваться данными, бывает удобно завести для их взаимодействия отдельный коммуникатор. В-третьих, при создании библиотек подпрограмм нужно гарантировать, что пересылки данных в библиотечных модулях не пересекутся с пересылками в основной программе. Решение этих задач можно обеспечить в полном объеме только при помощи создания нового независимого коммуникатора.

Операции с группами процессов

Группа – это упорядоченное множество процессов. Каждому процессу в группе сопоставлено целое число – *ранг* или *номер*. **MPI_GROUP_EMPTY** – пустая группа, не содержащая ни одного процесса. **MPI_GROUP_NULL** – значение, используемое для ошибочной группы.

Новые группы можно создавать как на основе уже существующих групп, так и на основе коммуникаторов, но в операциях обмена могут использоваться только коммуникаторы. Базовая группа, из которой создаются все остальные группы процессов, связана с коммуникатором **MPI_COMM_WORLD**, в нее входят все процессы приложения. Операции над группами процессов являются локальными, в них вовлекается только вызвавший процедуру процесс, а выполнение не требует межпроцессного обмена данными. Любой процесс может производить операции над любыми группами, в том числе над такими, которые не содержат данный процесс. При операциях над группами может получиться пустая группа **MPI_GROUP_EMPTY**.

```
MPI_COMM_GROUP(COMM, GROUP, IERR)
INTEGER COMM, GROUP, IERR
```
Получение группы **GROUP**, соответствующей коммуникатору **COMM**. В языке Си параметр **GROUP** имеет предопределенный тип **MPI_Group**. Поскольку изначально существует единственный нетривиальный коммуникатор **MPI_COMM_WORLD**, сначала нужно получить соответствующую ему группу процессов. Это можно сделать при помощи следующего вызова:

```
call MPI_COMM_GROUP(MPI_COMM_WORLD, group, ierr)
```

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERR
```
Создание группы **NEWGROUP** из **N** процессов прежней группы **GROUP** с рангами **RANKS(1),…,RANKS(N)**, причем рангу **RANKS(I)** в старой группе соответствует ранг **I-1** в новой группе. При **N=0** создается пустая группа **MPI_GROUP_EMPTY**. Возможно использование этой процедуры для задания нового порядка процессов в группе.

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERR
```
Создание группы **NEWGROUP** из процессов группы **GROUP**, исключая процессы с рангами **RANKS(1),…,RANKS(N)**, причем порядок оставшихся процессов в новой группе соответствует порядку процессов в старой группе. При **N=0** создается группа, идентичная старой группе.

В следующем примере создается две непересекающихся группы процессов **group1** и **group2** на основе процессов группы **group**. В каждую из создаваемых групп войдет примерно половина процессов прежней группы (при нечетном числе процессов в группу **group2** войдет на один процесс больше). Порядок нумерации процессов во вновь создаваемых группах сохранится.

```
size1 = size/2
do i = 1, size1
   ranks(i) = i-1
enddo
call MPI_GROUP_INCL(group, size1, ranks, group1, ierr)
call MPI_GROUP_EXCL(group, size1, ranks, group2, ierr)
```

Следующие три процедуры определяют операции над группами процессов, как над множествами. Из-за особенностей нумерации процессов ни объединение, ни пересечение групп не коммутативны, но ассоциативны.

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```
Создание группы **NEWGROUP** из пересечения групп **GROUP1** и **GROUP2**. Полученная группа содержит все процессы группы **GROUP1**, входящие также в группу **GROUP2** и упорядоченные, как в первой группе.

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```
Создание группы **NEWGROUP** из объединения групп **GROUP1** и **GROUP2**. Полученная группа содержит все процессы группы **GROUP1** в прежнем порядке, за которыми следуют процессы группы **GROUP2**, не вошедшие в группу **GROUP1**, также в прежнем порядке.

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERR
```
Создание группы **NEWGROUP** из разности групп **GROUP1** и **GROUP2**. Полученная группа содержит все элементы группы **GROUP1**, не входящие в группу **GROUP2** и упорядоченные, как в первой группе.

Например, пусть в группу **gr1** входят процессы **0**, **1**, **2**, **4**, **5**, а в группу **gr2** - процессы **0**, **2**, **3** (нумерация процессов задана в группе, соответствующей коммуникатору **MPI_COMM_WORLD**). Тогда после вызовов

```
call MPI_GROUP_INTERSECTION(gr1, gr2, newgr1, ierr)
call MPI_GROUP_UNION(gr1, gr2, newgr2, ierr)
call MPI_GROUP_DIFFERENCE(gr1, gr2, newgr3, ierr)
```

в группу **newgr1** входят процессы **0**, **2**;
в группу **newgr2** входят процессы **0**, **1**, **2**, **4**, **5**, **3**;
в группу **newgr3** входят процессы **1**, **4**, **5**.

Порядок нумерации процессов в полученных группах соответствует порядку их перечисления.

```
MPI_GROUP_SIZE(GROUP, SIZE, IERR)
INTEGER GROUP, SIZE, IERR
```
Определение количества **SIZE** процессов в группе **GROUP**.

```
MPI_GROUP_RANK(GROUP, RANK, IERR)
INTEGER GROUP, RANK, IERR
```
Определение номера процесса **RANK** в группе **GROUP**. Если вызвавший процесс не входит в группу **GROUP**, то возвращается значение **MPI_UNDEFINED**.

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2,
IERR)
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERR
```
В массиве **RANKS2** возвращаются ранги в группе **GROUP2** процессов с рангами **RANKS1** в группе **GROUP1**. Параметр **N** задает число процессов, для которых нужно определить ранги.

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERR)
INTEGER GROUP1, GROUP2, RESULT, IERR
```

Сравнение групп **GROUP1** и **GROUP2**. Если группы **GROUP1** и **GROUP2** полностью совпадают, то в параметре **RESULT** возвращается значение **MPI_IDENT**. Если группы отличаются только рангами процессов, то возвращается значение **MPI_SIMILAR**. Иначе возвращается значение **MPI_UNEQUAL**.

```
MPI_GROUP_FREE(GROUP, IERR)
INTEGER GROUP, IERR
```

Уничтожение группы **GROUP**. После выполнения процедуры переменная **GROUP** принимает значение **MPI_GROUP_NULL**. Если с этой группой к моменту вызова процедуры уже выполняется какая-то операция, то она будет завершена.

В следующем примере все процессы приложения разбиваются на две непересекающиеся примерно равные группы **group1** и **group2**. При нечетном числе процессов в группе **group2** может оказаться на один процесс больше, тогда последний процесс из данной группы не должен обмениваться данными ни с одним процессом из группы **group1**. С помощью вызовов процедуры **MPI_GROUP_TRANSLATE_RANKS** каждый процесс находит процесс с тем же номером в другой группе и обменивается с ним сообщением через коммуникатор **MPI_COMM_WORLD** при помощи вызова процедуры **MPI_SENDRECV**. В конце программы не нужные далее группы уничтожаются с помощью вызовов процедур **MPI_GROUP_FREE**.

```
      program example16
      include 'mpif.h'
      integer ierr, rank, i, size, size1
      integer a(4), b(4)
      integer status(MPI_STATUS_SIZE)
      integer group, group1, group2
      integer ranks(128), rank1, rank2, rank3
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_GROUP(MPI_COMM_WORLD, group, ierr)
      size1 = size/2
      do i = 1, size1
         ranks(i) = i-1
      enddo
      call MPI_GROUP_INCL(group, size1, ranks, group1, ierr)
      call MPI_GROUP_EXCL(group, size1, ranks, group2, ierr)
      call MPI_GROUP_RANK(group1, rank1, ierr)
      call MPI_GROUP_RANK(group2, rank2, ierr)
      if (rank1 .eq. MPI_UNDEFINED) then
         if(rank2 .lt. size1) then
            call MPI_GROUP_TRANSLATE_RANKS(group1, 1, rank2,
     &                                     group, rank3, ierr)
         else
            rank3 = MPI_UNDEFINED
         end if
```

```
 else
    call MPI_GROUP_TRANSLATE_RANKS(group2, 1, rank1,
&                                   group, rank3, ierr)
 end if
 a(1) = rank
 a(2) = rank1
 a(3) = rank2
 a(4) = rank3
 if (rank3 .ne. MPI_UNDEFINED) then
    call MPI_SENDRECV(a, 4, MPI_INTEGER, rank3, 1,
&                     b, 4, MPI_INTEGER, rank3, 1,
&                     MPI_COMM_WORLD, status, ierr)
 end if
 call MPI_GROUP_FREE(group, ierr)
 call MPI_GROUP_FREE(group1, ierr)
 call MPI_GROUP_FREE(group2, ierr)
 print *, 'process ', rank, ' a=', a, ' b=', b
 call MPI_FINALIZE(ierr)
 end
```

Операции с коммуникаторами

Коммуникатор предоставляет отдельный контекст обмена процессов некоторой группы. Контекст обеспечивает возможность независимых обменов данными. Каждой группе процессов может соответствовать несколько коммуникаторов, но каждый коммуникатор в любой момент времени однозначно соответствует только одной группе.

Следующие коммуникаторы создаются сразу после вызова процедуры **MPI_INIT**:

- **MPI_COMM_WORLD** – коммуникатор, объединяющий все процессы приложения;
- **MPI_COMM_NULL** – значение, используемое для ошибочного коммуникатора;
- **MPI_COMM_SELF** – коммуникатор, включающий только вызвавший процесс.

Создание коммуникатора является коллективной операцией и требует операции межпроцессного обмена, поэтому такие процедуры должны вызываться всеми процессами некоторого существующего коммуникатора.

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

Создание нового коммуникатора **NEWCOMM** с той же группой процессов и атрибутами, что и у коммуникатора **COMM**.

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERR)
INTEGER COMM, GROUP, NEWCOMM, IERR
```

Создание нового коммуникатора **NEWCOMM** из коммуникатора **COMM** для группы процессов **GROUP**, которая должна являться подмножеством группы, связанной с коммуникатором **COMM**. Вызов должен встретиться во всех процессах коммуникатора **COMM**. На процессах, не принадлежащих группе **GROUP**, будет возвращено значение **MPI_COMM_NULL**.

В следующем примере создается две новых группы, одна из которых содержит первую половину процессов, а вторая – вторую половину. При нечетном числе процессов во вторую группу войдет на один процесс больше. Каждая группа создается только на тех процессах, которые в нее входят. Для каждой новой группы создается соответствующий ей коммуникатор **new_comm**, и операция **MPI_ALLREDUCE** выполняется по отдельности для процессов, входящих в разные группы.

```
 call MPI_COMM_GROUP(MPI_COMM_WORLD, group, ierr)
 do i = 1, size/2
    ranks(i) = i-1
 end do
 if (rank .lt. size/2) then
    call MPI_GROUP_INCL(group, size/2, ranks,
&                       new_group, ierr)
 else
    call MPI_GROUP_EXCL(group, size/2, ranks,
&                       new_group, ierr)
 end if
 call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group,
&                     new_comm, ierr)
 call MPI_ALLREDUCE(sbuf, rbuf, 1, MPI_INTEGER,
&                   MPI_SUM, new_comm, ierr)
 call MPI_GROUP_RANK(new_group, new_rank, ierr)
 print *, 'rank= ', rank, ' newrank= ',
&         new_rank, ' rbuf= ', rbuf
```

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERR
```
Разбиение коммуникатора **COMM** на несколько новых коммуникаторов по числу значений параметра **COLOR**. В один коммуникатор попадают процессы с одним значением **COLOR**. Процессы с б**о**льшим значением параметра **KEY** получат больший ранг в новой группе, при одинаковом значении параметра **KEY** порядок нумерации процессов выбирается системой.

Процессы, которые не должны войти в новые коммуникаторы, указывают в качестве параметра **COLOR** константу **MPI_UNDEFINED**. Им в параметре **NEWCOMM** вернется значение **MPI_COMM_NULL**.

В следующем примере коммуникатор **MPI_COMM_WORLD** разбивается на три части. В первую войдут процессы с номерами **0**, **3**, **6** и т.д., во вторую – **1**, **4**, **7** и т.д., а в третью – **2**, **5**, **8** и т.д. Задание в качестве параметра **KEY** переменной **rank** гарантирует, что порядок нумерации процессов в создаваемых группах соответствует порядку нумерации в исходной группе, то есть, порядку перечисления выше.

```
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, mod(rank, 3),
     &                    rank, new_comm, ierr)
```

```
MPI_COMM_FREE(COMM, IERR)
INTEGER COMM, IERR
```
Удаление коммуникатора **COMM**. После выполнения процедуры переменной **COMM** присваивается значение **MPI_COMM_NULL**. Если с этим коммуникатором к моменту вызова процедуры уже выполняется какая-то операция, то она будет завершена.

В следующем примере создается один новый коммуникатор **comm_revs**, в который входят все процессы приложения, пронумерованные в обратном порядке. Когда коммуникатор становится ненужным, он удаляется при помощи вызова процедуры **MPI_COMM_FREE**. Так можно использовать процедуру **MPI_COMM_SPLIT** для перенумерации процессов.

```
      program example17
      include 'mpif.h'
      integer ierr, rank, size
      integer comm_revs, rank1
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, 1, size-rank,
     &                    comm_revs, ierr)
      call MPI_COMM_RANK(comm_revs, rank1, ierr)
      print *, 'rank = ', rank,' rank1 = ', rank1
      call MPI_COMM_FREE(comm_revs, ierr)
```

```
call MPI_FINALIZE(ierr)
end
```

Задания

- Какие группы процессов существуют при запуске приложения?
- Могут ли группы процессов иметь непустое пересечение, не совпадающее ни с одной из них полностью?
- В чем отличие между группой процессов и коммуникатором?
- Могут ли обмениваться данными процессы, принадлежащие разным коммуникаторам?
- Может ли в какой-то группе не быть процесса с номером **0**?
- Может ли в какую-либо группу не войти процесс с номером **0** в коммуникаторе **MPI_COMM_WORLD**?
- Может ли только один процесс в некоторой группе вызвать процедуру **MPI_GROUP_INCL**?
- Как создать новую группу из процессов **3**, **4** и **7** коммуникатора **MPI_COMM_WORLD**?
- Разбить все процессы приложения на три произвольных группы и напечатать ранги в **MPI_COMM_WORLD** тех процессов, что попали в первые две группы, но не попали в третью.
- Какие коммуникаторы существуют при запуске приложения?
- Можно ли в процессе выполнения программы изменить число процессов в коммуникаторе **MPI_COMM_WORLD**?
- Может ли только один процесс в некотором коммуникаторе вызвать процедуру **MPI_COMM_CREATE**?
- Можно ли при помощи процедуры **MPI_COMM_SPLIT** создать ровно один новый коммуникатор?
- Можно ли при помощи процедуры **MPI_COMM_SPLIT** создать столько новых коммуникаторов, сколько процессов входит в базовый коммуникатор?
- Реализовать разбиение процессов на две группы, в одной из которых осуществляется обмен данными по кольцу, а в другой – коммуникации по схеме master-slave.

Виртуальные топологии

Топология – это механизм сопоставления процессам некоторого коммуникатора альтернативной схемы адресации. В MPI топологии виртуальны, то есть они не связаны с физической топологией коммуникационной сети. Топология используется программистом для более удобного обозначения процессов,

и таким образом, приближения параллельной программы к структуре математического алгоритма. Кроме того, топология может использоваться системой для оптимизации распределения процессов по физическим процессорам используемого параллельного компьютера при помощи изменения порядка нумерации процессов внутри коммуникатора.

В MPI предусмотрены два типа топологий:

- *декартова топология* (прямоугольная решетка произвольной размерности);
- *топология графа*.

```
MPI_TOPO_TEST(COMM, TYPE, IERR)
INTEGER COMM, TYPE, IERR
```

Процедура определения типа топологии, связанной с коммуникатором **COMM**. Возможные возвращаемые значения параметра **TYPE**:

- **MPI_GRAPH** для топологии графа;
- **MPI_CART** для декартовой топологии;
- **MPI_UNDEFINED** – с коммуникатором **COMM** не связана никакая топология.

Декартова топология

```
MPI_CART_CREATE(COMM, NDIMS, DIMS, PERIODS, REORDER, COMM_CART,
IERR)
INTEGER COMM, NDIMS, DIMS(*), COMM_CART, IERR
LOGICAL PERIODS(*), REORDER
```

Создание коммуникатора **COMM_CART**, обладающего декартовой топологией, из процессов коммуникатора **COMM**. Параметр **NDIMS** задает размерность получаемой декартовой решетки, **DIMS(I)** – число элементов в измерении **I**, **1≤I≤NDIMS**. **PERIODS** – логический массив из **NDIMS** элементов, определяющий, является ли решетка периодической (значение **.TRUE.**) вдоль каждого измерения. **REORDER** – логический параметр, определяющий, что при значении **.TRUE.** системе разрешено менять порядок нумерации процессов для оптимизации распределения процессов по физическим процессорам используемого параллельного компьютера.

Процедура является коллективной, а значит, должна быть вызвана всеми процессами коммуникатора **COMM**. Если количество процессов в задаваемой топологии **COMM_CART** меньше числа процессов в исходном коммуникаторе **COMM**, то некоторым процессам может вернуться значение **MPI_COMM_NULL**, а значит, они не будут принимать участия в создаваемой топологии. Если ко-

личество процессов в задаваемой топологии больше числа процессов в исходном коммуникаторе, то вызов будет ошибочным.

В следующем примере создается трехмерная топология **4×3×2**, каждое измерение которой является периодическим, кроме того, разрешается переупорядочение процессов. Данный фрагмент должен выполняться не менее чем на **24** процессах.

```
      dims(1) = 4
      dims(2) = 3
      dims(3) = 2
      periods(1) = .TRUE.
      periods(2) = .TRUE.
      periods(3) = .TRUE.
      call MPI_CART_CREATE(MPI_COMM_WORLD, 3, dims, periods,
     &                     .TRUE., comm_cart, ierr)
```

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

Процедура помогает определить размеры **DIMS(I)** для каждой из **NDIMS** размерностей при создании декартовой топологии для **NNODES** процессов. Предпочтительным считается создание топологии, в которой число процессов по разным размерностям примерно одно и то же. Пользователь может управлять числом процессов в некоторых размерностях следующим образом. Значение **DIMS(I)** рассчитывается данной процедурой, если перед вызовом оно равно **0**, иначе оставляется без изменений. Отрицательные значения элементов массива **DIMS** являются ошибочными. Перед вызовом процедуры значение **NNODES** должно быть кратно произведению ненулевых значений массива **DIMS**. Выходные значения массива **DIMS**, переопределенные данной процедурой, будут упорядочены в порядке убывания. Процедура является локальной и не требует межпроцессного взаимодействия.

В следующей таблице приведены четыре примера использования процедуры **MPI_DIMS_CREATE** для создания трехмерных топологий. В первом примере **6** процессов образуют решетку **3×2×1**, причем размеры упорядочены в порядке убывания. Во втором примере делается попытка распределить **7** процессов по трем измерениям, единственный возможный вариант – решетка **7×1×1**. В третьем примере для второй размерности изначально задано значение **3**, две оставшиеся размерности определяют решетку **2×3×1**. Четвертый вызов ошибочен, так как общее число процессов (**7**) не делится нацело на заданный размер во второй размерности (**3**).

dims перед вызовом	вызов процедуры	**dims** после вызова
(0, 0, 0)	**MPI_DIMS_CREATE(6, 3, dims, ierr)**	**(3, 2, 1)**
(0, 0, 0)	**MPI_DIMS_CREATE(7, 3, dims, ierr)**	**(7, 1, 1)**
(0, 3, 0)	**MPI_DIMS_CREATE(6, 3, dims, ierr)**	**(2, 3, 1)**
(0, 3, 0)	**MPI_DIMS_CREATE(7, 3, dims, ierr)**	**ошибка**

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

Определение декартовых координат процесса по его рангу **RANK** в коммуникаторе **COMM**. Координаты возвращаются в массиве **COORDS** с числом элементов **MAXDIMS**. Отсчет координат по каждому измерению начинается с нуля.

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

Определение ранга **RANK** процесса в коммуникаторе **COMM** по его декартовым координатам **COORDS**. Для периодических решеток координаты вне допустимых интервалов пересчитываются, для непериодических решеток они являются ошибочными.

```
MPI_CART_SUB(COMM, DIMS, NEWCOMM, IERR)
INTEGER COMM, NEWCOMM, IERR
LOGICAL DIMS(*)
```

Расщепление коммуникатора **COMM**, с которым связана декартова топология при помощи процедуры **MPI_CART_CREATE**, на подгруппы, соответствующие декартовым подрешеткам меньшей размерности. **I**–ый элемент логического массива **DIMS** устанавливается равным значению **.TRUE.**, если **I**–ое измерение должно остаться в формируемой подрешетке, связанной с коммуникатором **NEWCOMM**.

Возьмем трехмерную топологию, созданную в предыдущем примере. Ниже показано, как расщепить топологию **4×3×2** на **3** двумерных подрешетки **4×2** по **8** процессов в каждой.

```
dims(0) = .TRUE.
dims(1) = .FALSE.
dims(2) = .TRUE.
call MPI_CART_SUB(comm_cart, dims, newcomm, ierr)
```

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

Определение размерности **NDIMS** декартовой топологии, связанной с коммуникатором **COMM**.

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

LOGICAL PERIODS(*)
Получение информации о декартовой топологии коммуникатора **COMM** и координатах в ней вызвавшего процесса. **MAXDIMS** задает размерность декартовой топологии. В параметре **DIMS** возвращается количество процессов для каждого измерения, в параметре **PERIODS** – периодичность по каждому измерению, в параметре **COORDS** – координаты вызвавшего процесса в декартовой топологии.

MPI_CART_SHIFT(COMM, DIRECTION, DISP, SOURCE, DEST, IERR)
INTEGER COMM, DIRECTION, DISP, SOURCE, DEST, IERR
Получение номеров посылающего (**SOURCE**) и принимающего (**DEST**) процессов в декартовой топологии коммуникатора **COMM** для осуществления сдвига вдоль измерения **DIRECTION** на величину **DISP**.

Для периодических измерений осуществляется циклический сдвиг, для непериодических – линейный сдвиг. В случае линейного сдвига на некоторых процессах в качестве номеров посылающего или принимающего процессов может быть получено значение **MPI_PROC_NULL**, означающее выход за границы диапазона. В случае циклического сдвига последний процесс по данному измерению осуществляет обмены с нулевым процессом. Для **n**-мерной декартовой решетки значение **DIRECTION** должно быть в пределах от **0** до **n-1**.

Значения **SOURCE** и **DEST** можно использовать, например, для обмена с помощью процедуры **MPI_SENDRECV**.

В следующем примере создается двумерная декартова решетка, периодическая по обоим измерениям, определяются координаты процесса в данной решетке. Потом при помощи процедуры **MPI_CART_SHIFT** вычисляются координаты процессов, с которыми нужно совершить обмен данными для осуществления циклического сдвига с шагом **2** по измерению **1**. В конце фрагмента полученные значения номеров процессов используются для обмена данными при помощи процедуры **MPI_SENDRECV_REPLACE**.

```
      periods(1) = .TRUE.
      periods(2) = .TRUE.
      call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims,
     &                     periods, .TRUE., comm, ierr)
      call MPI_COMM_RANK(comm, rank, ierr)
      call MPI_CART_COORDS(comm, rank, 2, coords, ierr)
      shift = 2
      dest = 1
      call MPI_CART_SHIFT(comm, 0, shift, source, dest, ierr)
      call MPI_SENDRECV_REPLACE(a, 1, MPI_REAL, dest, 0,
     &                          source, 0, comm, status, ierr)
```

Топология графа

```
MPI_GRAPH_CREATE(COMM, NNODES, INDEX, EDGES, REORDER,
COMM_GRAPH, IERR)
INTEGER COMM, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERR
LOGICAL REORDER
```

Создание на основе коммуникатора **COMM** нового коммуникатора **COMM_GRAPH** с топологией графа. Параметр **NNODES** задает число вершин графа, **INDEX(I)** содержит суммарное количество соседей для первых **I** вершин. Массив **EDGES** содержит упорядоченный список номеров процессов-соседей всех вершин. Параметр **REORDER** при значении **.TRUE.** означает, что системе разрешено менять порядок нумерации процессов.

Процедура является коллективной, а значит, должна быть вызвана всеми процессами исходного коммуникатора. Если **NNODES** меньше числа процессов коммуникатора **COMM**, то некоторым процессам вернется значение **MPI_COMM_NULL**, а значит, они не будут принимать участия в создаваемой топологии. Если **NNODES** больше числа процессов коммуникатора **COMM**, то вызов процедуры является ошибочным.

В следующей табличке приведен пример описания графа через задание всех соседей каждой вершины.

Процесс	Соседи
0	**1, 3**
1	**0**
2	**3**
3	**0, 2**

Для описания такого графа нужно заполнить следующие структуры данных:

```
INDEX=2, 3, 4, 6
EDGES=1, 3, 0, 3, 0, 2
```

После этого можно создать топологию графа, например, с помощью следующего вызова (вызов будет корректным при выполнении на не менее чем на **4** процессах):

```
      call MPI_GRAPH_CREATE(MPI_COMM_WORLD, 4, INDEX, EDGES,
     &                      .TRUE., comm_graph, ierr)
```

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERR)
INTEGER COMM, RANK, NNEIGHBORS, IERR
```
Определение количества **NNEIGHBORS** непосредственных соседей процесса с рангом **RANK** в графовой топологии, связанной с коммуникатором **COMM**.

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAX, NEIGHBORS, IERR)
INTEGER COMM, RANK, MAX, NEIGHBORS(*), IERR
```
Определение рангов непосредственных соседей процесса с рангом **RANK** в графовой топологии, связанной с коммуникатором **COMM**. Ранги соседей возвращаются в массиве **NEIGHBORS**, **MAX** задает ограничение на количество соседей (может быть получено, например, вызовом процедуры **MPI_GRAPH_NEIGHBORS_COUNT**).

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERR)
INTEGER COMM, NNODES, NEDGES, IERR
```
Определение числа вершин **NNODES** и числа ребер **NEDGES** графовой топологии, связанной с коммуникатором **COMM**.

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERR)
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERR
```
Определение информации о топологии графа, связанной с коммуникатором **COMM**. В массивах **INDEX** и **EDGES** возвращается описание графовой топологии в том виде, как она задается при создании топологии с помощью процедуры **MPI_GRAPH_CREATE**. Параметры **MAXINDEX** и **MAXEDGES** задают ограничения на размеры соответствующих массивов (могут быть получены, например, вызовом процедуры **MPI_GRAPHDIMS_GET**).

В следующем примере создается графовая топология **comm_graph** для общения процессов по коммуникационной схеме master-slave. Все процессы в рамках данной топологии могут общаться только с нулевым процессом. После создания топологии с помощью вызова процедуры **MPI_GRAPH_CREATE** каждый процесс определяет количество своих непосредственных соседей в рамках данной топологии (с помощью вызова процедуры **MPI_GRAPH_NEIGHBORS_COUNT**) и ранги процессов-соседей (с помощью вызова процедуры **MPI_GRAPH_NEIGHBORS**). После этого каждый процесс может в рамках данной топологии обмениваться данными со своими непосредственными соседями, например, при помощи вызова процедуры **MPI_SENDRECV**.

```
 program example18
 include 'mpif.h'
 integer ierr, rank, rank1, i, size, MAXPROC, MAXEDGES
 parameter (MAXPROC = 128, MAXEDGES = 512)
 integer a, b
 integer status(MPI_STATUS_SIZE)
 integer comm_graph, index(MAXPROC), edges(MAXEDGES)
 integer num, neighbors(MAXPROC)
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 do i = 1, size
    index(i) = size+i-2
 end do
 do i = 1, size-1
    edges(i) = i
    edges(size+i-1) = 0
 end do
 call MPI_GRAPH_CREATE(MPI_COMM_WORLD, size, index, edges,
&                      .TRUE., comm_graph, ierr)
 call MPI_GRAPH_NEIGHBORS_COUNT(comm_graph, rank, num,
&                               ierr)
 call MPI_GRAPH_NEIGHBORS(comm_graph, rank, num, neighbors,
&                         ierr)
 do i = 1, num
    call MPI_SENDRECV(rank, 1, MPI_INTEGER, neighbors(i),
&                     1, rank1, 1, MPI_INTEGER,
&                     neighbors(i), 1, comm_graph,
&                     status, ierr)
    print *, 'procecc ', rank, ' communicate with process',
&            rank1
 end do
 call MPI_FINALIZE(ierr)
 end
```

Задания

- Обязана ли виртуальная топология повторять физическую топологию целевого компьютера?
- Любой ли коммуникатор может обладать виртуальной топологией?
- Может ли процесс входить одновременно в декартову топологию и в топологию графа?
- Можно ли вызвать процедуру **MPI_CART_CREATE** только на половине процессов коммуникатора?
- Необходимо ли использовать процедуру **MPI_DIMS_CREATE** перед вызовом процедуры **MPI_CART_CREATE**?
- Можно ли использовать координаты процесса в декартовой топологии в процедурах обмена данными?

- Какие пересылки данных осуществляются процедурой **`MPI_CART_SHIFT`**?
- Как определить, с какими процессами в топологии графа связан данный процесс?
- Как создать топологию графа, в которой каждый процесс связан с каждым?
- Реализовать разбиение процессов на две группы, в одной из которых осуществляется обмен по кольцу при помощи сдвига в одномерной декартовой топологии, а в другой – коммуникации по схеме master-slave, реализованной при помощи топологии графа.
- Использовать двумерную декартову топологию процессов при реализации параллельной программы перемножения матриц.

Пересылка разнотипных данных

Под *сообщением* в MPI понимается массив однотипных данных, расположенных в последовательных ячейках памяти. Часто в программах требуются пересылки более сложных объектов данных, состоящих из разнотипных элементов или расположенных не в последовательных ячейках памяти. В этом случае можно либо посылать данные небольшими порциями расположенных подряд элементов одного типа, либо использовать копирование данных перед отсылкой в некоторый промежуточный буфер. Оба варианта являются достаточно неудобными и требуют дополнительных затрат как времени, так и оперативной памяти.

Для пересылки разнотипных данных в MPI предусмотрены два специальных способа:

- *Производные типы данных*;
- *Упаковка данных*.

Производные типы данных

Производные типы данных создаются во время выполнения программы с помощью процедур-конструкторов на основе существующих к моменту вызова конструктора типов данных.

Создание типа данных состоит из двух этапов:

1. Конструирование типа.
2. Регистрация типа.

После регистрации производный тип данных можно использовать наряду с предопределенными типами в операциях пересылки, в том числе и в коллективных операциях. После завершения работы с производным типом данных его рекомендуется аннулировать. При этом все произведенные на его основе новые типы данных остаются и могут использоваться дальше.

Производный тип данных характеризуется последовательностью базовых типов данных и набором целочисленных значений смещения элементов типа относительно начала буфера обмена. Смещения могут быть как положительными, так и отрицательными, не обязаны различаться, не требуется их упорядоченность. Таким образом, последовательность элементов данных в производном типе может отличаться от последовательности исходного типа, а один элемент данных может встречаться в конструируемом типе многократно.

MPI_TYPE_CONTIGUOUS(COUNT, TYPE, NEWTYPE, IERR)
INTEGER COUNT, TYPE, NEWTYPE, IERR

Создание нового типа данных **NEWTYPE**, состоящего из **COUNT** последовательно расположенных элементов базового типа данных **TYPE**. Фактически создаваемый тип данных представляет массив данных базового типа как отдельный объект.

В следующем примере создается новый тип данных **newtype**, который в дальнейшем (после регистрации типа) может использоваться для пересылки пяти расположенных подряд целых чисел.

```
call MPI_TYPE_CONTIGUOUS(5, MPI_INTEGER, newtype, ierr)
```

MPI_TYPE_VECTOR(COUNT, BLOCKLEN, STRIDE, TYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLEN, STRIDE, TYPE, NEWTYPE, IERR

Создание нового типа данных **NEWTYPE**, состоящего из **COUNT** блоков по **BLOCKLEN** элементов базового типа данных **TYPE**. Следующий блок начинается через **STRIDE** элементов базового типа данных после начала предыдущего блока.

В следующем примере создается новый тип данных **newtype**, который после регистрации может быть использован для пересылки как единого целого шести элементов данных, которые можно представить следующим образом (тип элемента данных, количество элементов данных от начала буфера посылки):

{(**MPI_REAL**, **0**), (**MPI_REAL**, **1**), (**MPI_REAL**, **2**),
(**MPI_REAL**, **5**), (**MPI_REAL**, **6**), (**MPI_REAL**, **7**)}

```
      count = 2
      blocklen = 3
      stride = 5
      call MPI_TYPE_VECTOR(count, blocklen, stride,
     &                     MPI_REAL, newtype, ierr)
```

MPI_TYPE_HVECTOR(COUNT, BLOCKLEN, STRIDE, TYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLEN, STRIDE, TYPE, NEWTYPE, IERR
Создание нового типа данных **NEWTYPE**, состоящего из **COUNT** блоков по **BLOCKLEN** элементов базового типа данных **TYPE**. Следующий блок начинается через **STRIDE** байт после начала предыдущего блока.

MPI_TYPE_INDEXED(COUNT, BLOCKLENS, DISPLS, TYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENS(*), DISPLS(*), TYPE, NEWTYPE, IERR
Создание нового типа данных **NEWTYPE**, состоящего из **COUNT** блоков по **BLOCKLENS(I)** элементов базового типа данных. **I**-й блок начинается через **DISPLS(I)** элементов базового типа данных с начала буфера посылки. Полученный тип данных можно считать обобщением векторного типа.

В следующем примере задается тип данных **newtype** для описания нижнетреугольной матрицы типа **double precision** (при этом учитывается, что в языке Фортран массивы хранятся по столбцам).

```
      do i = 1, n
         blocklens(i) = n-i+1
         displs(i) = n*(i-1)+i-1
      end do
      call MPI_TYPE_INDEXED(n, blocklens, displs,
                            MPI_DOUBLE_PRECISION, newtype, ierr)
```

MPI_TYPE_HINDEXED(COUNT, BLOCKLENS, DISPLS, TYPE, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENS(*), DISPLS(*), TYPE, NEWTYPE, IERR
Создание нового типа данных **NEWTYPE**, состоящего из **COUNT** блоков по **BLOCKLENS(I)** элементов базового типа данных. **I**-й блок начинается через **DISPLS(I)** байт с начала буфера посылки.

MPI_TYPE_STRUCT(COUNT, BLOCKLENS, DISPLS, TYPES, NEWTYPE, IERR)
INTEGER COUNT, BLOCKLENS(*), DISPLS(*), TYPES(*), NEWTYPE, IERR
Создание структурного типа данных **NEWTYPE** из **COUNT** блоков по **BLOCKLENS(I)** элементов типа **TYPES(I)**. **I**-й блок начинается через **DISPLS(I)** байт с начала буфера посылки.

В следующем примере создается новый тип данных **newtype**, который после регистрации может быть использован для пересылки как единого целого пяти элементов данных, которые можно представить следующим образом (тип элемента данных, количество байт от начала буфера посылки):
{(**MPI_DOUBLE_PRECISION**, **0**), (**MPI_DOUBLE_PRECISION**, **8**),

(**MPI_DOUBLE_PRECISION**, **16**), (**MPI_CHARACTER**, **24**), (**MPI_CHARACTER**, **25**)}

```
      blocklens(1) = 3
      blocklens(2) = 2
      types(1) = MPI_DOUBLE_PRECISION
      types(2) = MPI_CHARACTER
      displs(1) = 0
      displs(2) = 24
      call MPI_TYPE_STRUCT(2, blocklens, displs, types,
     &                     newtype, ierr)
```

```
MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER DATATYPE, IERR
```

Регистрация созданного производного типа данных **DATATYPE**. После регистрации этот тип данных можно использовать в операциях обмена наравне с предопределенными типами данных. Предопределенные типы данных регистрировать не нужно.

```
MPI_TYPE_FREE(DATATYPE, IERR)
INTEGER DATATYPE, IERR
```

Аннулирование производного типа данных **DATATYPE**. Параметр **DATATYPE** устанавливается в значение **MPI_DATATYPE_NULL**. Гарантируется, что любой начатый обмен, использующий данные аннулируемого типа, будет нормально завершен. При этом производные от **DATATYPE** типы данных остаются и могут использоваться дальше. Предопределенные типы данных не могут быть аннулированы.

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

Определение размера **SIZE** типа данных **DATATYPE** в байтах (объема памяти, занимаемого одним элементом данного типа).

```
MPI_ADDRESS(LOCATION, ADDRESS, IERR)
<type> LOCATION(*)
INTEGER ADDRESS, IERR
```

Определение абсолютного байт-адреса **ADDRESS** размещения массива **LOCATION** в оперативной памяти компьютера. Адрес отсчитывается от базового адреса, значение которого содержится в системной константе **MPI_BOTTOM**. Процедура позволяет определять абсолютные адреса объектов как в языке Фортран, так и в Си, хотя в Си для этого предусмотрены иные средства. В языке Си параметр **ADDRESS** имеет тип **MPI_Aint**.

В следующем примере описывается новый тип данных **newtype**, который после регистрации используется для пересылки как единого целого двух элементов данных типов **double precision** и **character(1)**. В качестве адреса буфера посылки используется базовый адрес **MPI_BOTTOM**, а для определения смещений элементов данных **dat1** и **dat2** используются вызовы процедуры

MPI_ADDRESS. Перед пересылкой новый тип регистрируется при помощи вызова процедуры **MPI_TYPE_COMMIT**. Заметим, что пересылается один элемент данных производного типа, хотя он и состоит из двух разнотипных элементов.

```
      blocklens(1) = 1
      blocklens(2) = 1
      types(1) = MPI_DOUBLE_PRECISION
      types(2) = MPI_CHARACTER
      call MPI_ADDRESS(dat1, address(1), ierr)
      displs(1) = address(1)
      call MPI_ADDRESS(dat2, address(2), ierr)
      displs(2) = address(2)
      call MPI_TYPE_STRUCT(2, blocklens, displs, types,
     &                     newtype, ierr)
      call MPI_TYPE_COMMIT(newtype, ierr)
      call MPI_SEND(MPI_BOTTOM, 1, newtype, dest, tag,
     &              MPI_COMM_WORLD, ierr)
```

```
MPI_TYPE_LB(DATATYPE, DISPL, IERR)
INTEGER DATATYPE, DISPL, IERR
```

Определение смещения **DISPL** в байтах нижней границы элемента типа данных **DATATYPE** от начала буфера данных.

```
MPI_TYPE_UB(DATATYPE, DISPL, IERR)
INTEGER DATATYPE, DISPL, IERR
```

Определение смещения **DISPL** в байтах верхней границы элемента типа данных **DATATYPE** от начала буфера данных.

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

Определение диапазона **EXTENT** (разницы между верхней и нижней границами элемента данного типа) типа данных **DATATYPE** в байтах.

В следующем примере производный тип данных используется для перестановки столбцов матрицы в обратном порядке. Тип данных **matr_rev**, создаваемый процедурой **MPI_TYPE_VECTOR**, описывает локальную часть матрицы данного процесса в переставленными в обратном порядке столбцами. После регистрации этот тип данных может использоваться при пересылке. Программа работает правильно, если размер матрицы **N** делится нацело на число процессов приложения.

```
 program example19
 include 'mpif.h'
 integer ierr, rank, size, N, nl
 parameter (N = 8)
 double precision a(N, N), b(N, N)
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
 nl = (N-1)/size+1
 call work(a, b, N, nl, size, rank)
 call MPI_FINALIZE(ierr)
 end

 subroutine work(a, b, n, nl, size, rank)
 include 'mpif.h'
 integer ierr, rank, size, n, nl, ii, matr_rev
 double precision a(n, nl), b(n, nl)
 integer i, j, status(MPI_STATUS_SIZE)
 do j = 1, nl
    do i = 1, n
       b(i,j) = 0.d0
       ii = j+rank*nl
       a(i,j) = 100*ii+i
    enddo
 enddo
 call MPI_TYPE_VECTOR(nl, n, -n, MPI_DOUBLE_PRECISION,
&                     matr_rev, ierr)
 call MPI_TYPE_COMMIT(matr_rev, ierr)
 call MPI_SENDRECV(a(1, nl), 1, MATR_REV, size-rank-1, 1,
&                  b, nl*n, MPI_DOUBLE_PRECISION,
&                  size-rank-1, 1, MPI_COMM_WORLD,
&                  status, ierr)
 do j = 1, nl
    do i = 1, n
       print *, 'process ', rank, ': ',
&              j+rank*nl, ' ', i, a(i,j), ' ', b(i,j)
    enddo
 enddo
 end
```

Упаковка данных

Для пересылок разнородных данных наряду с созданием производных типов можно использовать операции упаковки и распаковки данных. Разнородные или расположенные не в последовательных ячейках памяти данные помещаются в один непрерывный буфер, пересылается, а потом полученное сообщение снова распределяется по нужным ячейкам памяти.

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION,
COMM, IERR)
<type> INBUF(*), OUTBUF(*)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERR
```

Упаковка **INCOUNT** элементов типа **DATATYPE** из массива **INBUF** в массив **OUTBUF** со сдвигом **POSITION** байт от начала массива. Буфер **OUTBUF** должен содержать по крайней мере **OUTSIZE** байт. После выполнения процедуры параметр **POSITION** увеличивается на число байт, равное размеру записи. Параметр **COMM** указывает на коммуникатор, в котором в дальнейшем будет пересылаться сообщение. Для пересылки упакованных данных используется тип данных **MPI_PACKED**.

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

Распаковка **OUTCOUNT** элементов типа **DATATYPE** из массива **INBUF** со сдвигом **POSITION** байт от начала массива в массив **OUTBUF**. Массив **INBUF** имеет размер не менее **INSIZE** байт.

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

Определение необходимого объема памяти **SIZE** (в байтах) для упаковки **INCOUNT** элементов типа **DATATYPE**. Необходимый для упаковки размер может превышать сумму размеров пакуемых элементов данных.

В следующем примере массив **buf** используется в качестве буфера для упаковки **10** элементов массива **a** типа **real** и **10** элементов массива **b** типа **character**. Полученное сообщение пересылается процедурой **MPI_BCAST** от процесса **0** всем остальным процессам, полученное сообщение распаковывается при помощи вызовов процедур **MPI_UNPACK**.

```
program example20
include 'mpif.h'
integer ierr, rank, position
real a(10)
character b(10), buf(100)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
do i = 1, 10
   a(i) = rank + 1.0
   if(rank .eq. 0) then
      b(i) = 'a'
   else
      b(i) = 'b'
   end if
end do
position=0
```

```
 if(rank .eq. 0) then
    call MPI_PACK(a, 10, MPI_REAL, buf, 100, position,
&                 MPI_COMM_WORLD, ierr)
    call MPI_PACK(b, 10, MPI_CHARACTER, buf, 100, position,
&                 MPI_COMM_WORLD, ierr)
    call MPI_BCAST(buf, 100, MPI_PACKED, 0,
&                  MPI_COMM_WORLD, ierr)
 else
    call MPI_BCAST(buf, 100, MPI_PACKED, 0,
&                  MPI_COMM_WORLD, ierr)
    position=0
    call MPI_UNPACK(buf, 100, position, a, 10, MPI_REAL,
&                   MPI_COMM_WORLD, ierr)
    call MPI_UNPACK(buf, 100, position, b, 10,
&                   MPI_CHARACTER, MPI_COMM_WORLD, ierr)
 end if
 print *, 'procecc ', rank, ' a=', a, ' b=', b
 call MPI_FINALIZE(ierr)
 end
```

Задания

- Создать производный тип данных для пересылок диагональной матрицы.
- Как соотносятся значения, возвращаемые процедурами **MPI_TYPE_SIZE** и **MPI_TYPE_EXTENT**?
- Можно ли использовать производные типы данных без вызова процедуры **MPI_TYPE_COMMIT**?
- Переслать нулевому процессу от всех процессов приложения структуру, состоящую из ранга процесса и названия узла, на котором данный процесс запущен (полученного с помощью процедуры **MPI_GET_PROCESSOR_NAME**).
- Прямоугольная матрица распределена по процессам по строкам. Переставить строки матрицы в обратном порядке, используя для пересылок производный тип данных.
- Сделать предыдущую задачу с использованием пересылок упакованных данных.
- В чем преимущества и недостатки использования пересылок упакованных данных по сравнению с пересылками данных производных типов?
- Написать программу транспонирования матрицы с использованием производных типов данных.

Литература

1. MPI: A Message-Passing Interface Standard (Version 1.1) (http://parallel.ru/docs/Parallel/mpi1.1/mpi-report.html)
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.
3. Антонов А.С. Введение в параллельные вычисления (методическое пособие). М.: Изд-во Физического факультета МГУ, 2002.
4. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону: Издательство ООО "ЦВВР", 2003.
5. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. Минск: БГУ, 2002.
6. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.
7. Корнеев В.Д. Параллельное программирование в MPI. Новосибирск: Изд-во СО РАН, 2000.

Учебное издание

Антонов Александр Сергеевич

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ MPI

Подписано в печать 21.01.2004. Формат 60x84/16.
Бумага офсетная №1. Печать ризо. Усл. печ. л. 4,5.
Уч.-изд. л. 4,3. Тираж 100 экз. Заказ № 1.

Ордена "Знак Почета" Издательство Московского университета.
125009, Москва, ул. Б. Никитская, 5/7.

Участок оперативной печати НИВЦ МГУ.
119992, ГСП-2, Москва, НИВЦ МГУ.