```
 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