- 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)
```

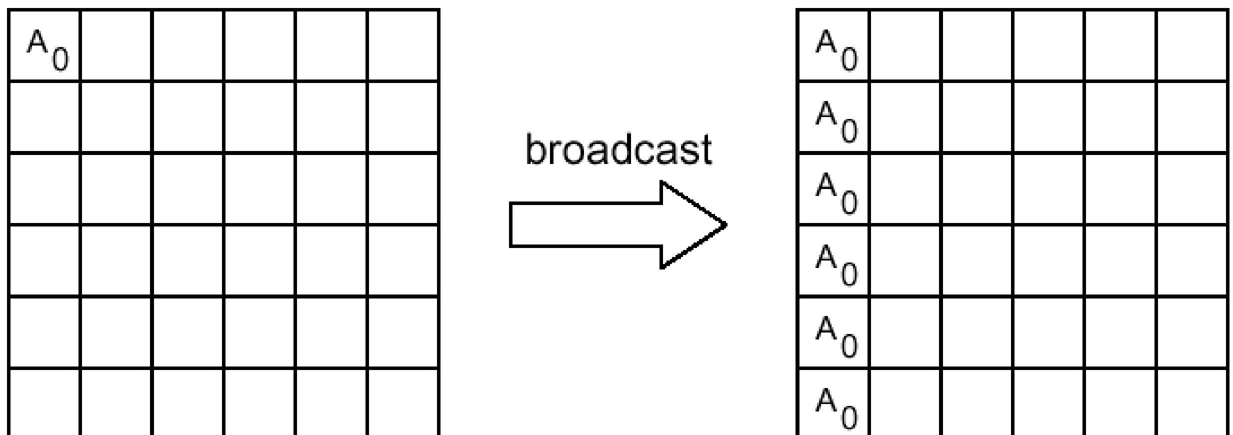
```
            call MPI_WAIT(req(1), statuses, ierr)
         end do
      end if
      time_finish = MPI_WTIME(ierr)-time_start
      print *, 'rank = ', rank, ' all time = ',
     &         (time_finish)/NTIMES

      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:

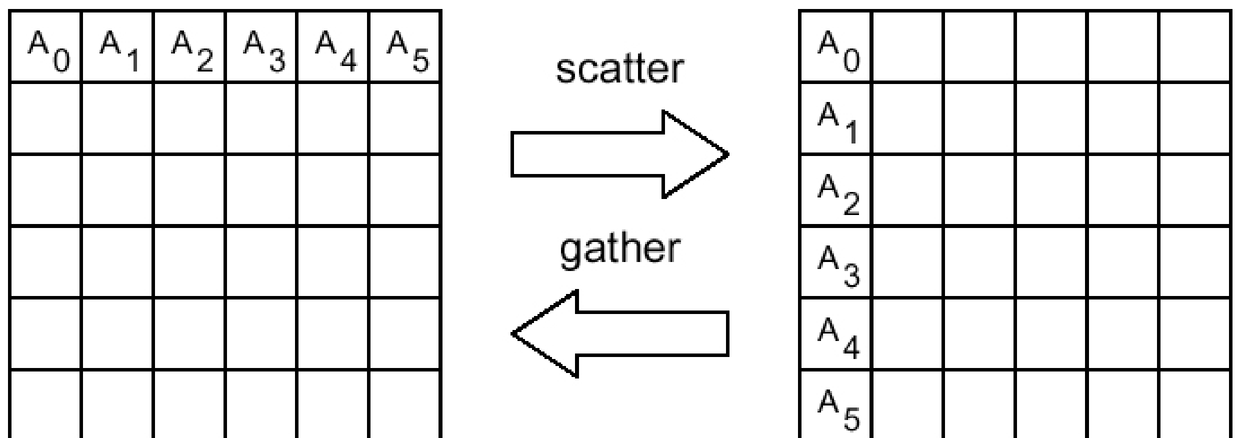
```
      call MPI_BCAST(buf, 100, MPI_INTEGER,
     &               2, MPI_COMM_WORLD, ierr)
```

```
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**).

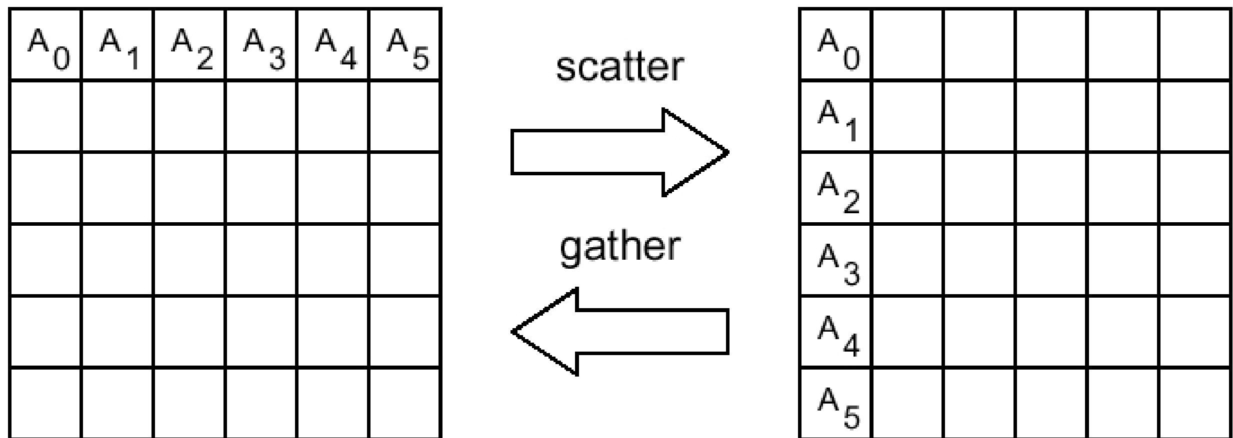
Данные, посланные процессом **J-1**, размещаются в **J**-ом блоке буфера **RBUF** на процессе **ROOT**, который начинается со смещением в **DISPLS(J)** элементов типа **RTYPE** с начала буфера.

```
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**.

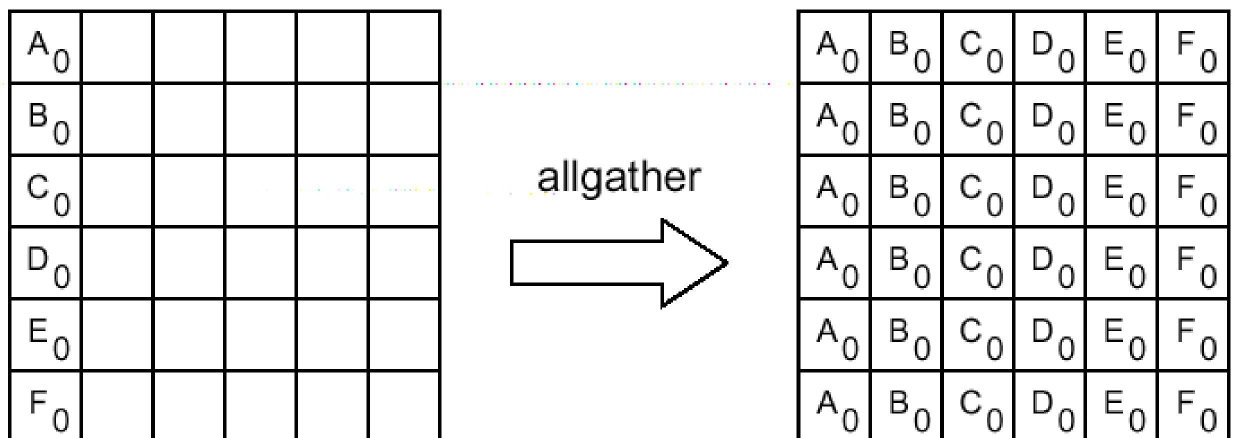
SCOUNTS – целочисленный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе **COMM**).

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**.

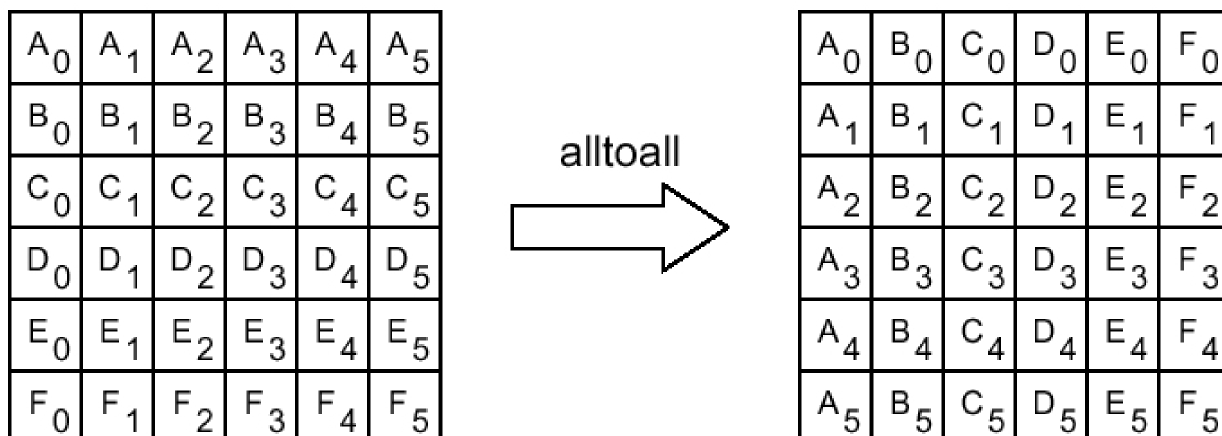
Следующая схема иллюстрирует действие процедуры **MPI_ALLGATHER**.



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 $\sum_I$ **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,