Технология программирования MPI

Антонов Александр Сергеевич,
к.ф.-м.н., в.н.с. лаборатории
Параллельных информационных
технологий НИВЦ МГУ,
asa@parallel.ru

MPI

•MPI - *Message Passing Interface*, интерфейс передачи сообщений.

•Стандарт MPI 3.0.

•Более 250 процедур.

•SPMD-модель параллельного программирования.

MPI

•Префикс **`MPI_`** .

`#include <mpi.h>`

(**`mpif.h`** для языка Фортран)

•*Процессы*, посылка сообщений.

•*Сообщение* – массив однотипных данных, расположенных в последовательных ячейках памяти.

•*Группа* – упорядоченное множество процессов.

MPI

Коммуникатор – контекст обмена группы.

В операциях обмена используются только коммуникаторы!

Коммуникаторы имеют в языке Си предопределённый тип **`MPI_Comm`** (в языке Фортран – тип **`INTEGER`**).

MPI

`MPI_COMM_WORLD` – коммуникатор для всех процессов приложения.

`MPI_COMM_SELF` – коммуникатор, включающий только текущий процесс.

`MPI_COMM_NULL` – коммуникатор, не содержащий ни одного процесса.

MPI

Каждый процесс может одновременно входить в разные коммуникаторы.

Два основных атрибута процесса: коммуникатор (группа) и номер процесса в коммуникаторе (группе).

Если коммуникатор содержит **`n`** процессов, то номера процессов в нём лежат в пределах от **`0`** до **`n - 1`**.

MPI

•*Сообщение* — набор данных некоторого типа.

•Атрибуты сообщения: номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения, коммуниктор.

•Идентификатор сообщения - целое неотрицательное число в диапазоне от **0** до **MPI_TAG_UB** (не меньше **32767**).

•Для работы с атрибутами сообщений введена структура **MPI_Status**.

MPI

Возвращаемым значением (в Фортране – последним аргументом) для большинства процедур MPI является информация об успешности завершения.

В случае успешного выполнения процедура вернёт значение **MPI_SUCCESS**, иначе - код ошибки.

Предопределённые значения, соответствующие различным ошибочным ситуациям, перечислены в файле **mpi.h**

MPI

int MPI_Init(int *argc, char ***argv)

Инициализация параллельной части программы. Почти все другие процедуры MPI могут быть вызваны только после вызова **MPI_Init**. Инициализация параллельной части для каждого приложения должна выполняться только один раз.

MPI

int MPI_Finalize(void)

Завершение параллельной части приложения. Все последующие обращения к большинству процедур 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.

MPI

Простейшая параллельная программа:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   printf("Before MPI_INIT\n");
   MPI_Init(&argc, &argv);
   printf("Parallel section\n");
   MPI_Finalize();
   printf("After MPI_FINALIZE\n");
}
```

MPI

int MPI_Initialized(int *flag)

В аргументе **flag** возвращает **1**, если вызвана после процедуры **MPI_Init**, и **0** в противном случае.

int MPI_Finalized(int *flag)

В аргументе **flag** возвращает **1**, если вызвана после процедуры **MPI_Finalize**, и **0** в противном случае.

Процедуры можно вызвать до **MPI_Init** и после **MPI_Finalize.**

MPI

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

В аргументе **`size`** возвращает число параллельных процессов в коммуникаторе **`comm`**.

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

В аргументе **`rank`** возвращает номер процесса в коммуникаторе **`comm`** в диапазоне от **`0`** до **`size-1`**.

MPI

```
#include <stdio.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char **argv)
{
   int rank, size, n, i, ibeg, iend;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   n=(MAX-1)/size+1;
   ibeg=rank*n+1; iend=(rank+1)*n;
   for(i=ibeg; i<=((iend>MAX)?MAX:iend); i++)
      printf ("process %d, %d^2=%d\n", rank, i, i*i);
   MPI_Finalize();
}
```

MPI

`double MPI_Wtime(void)`

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

`double MPI_Wtick(void)`

Возвращает разрешение таймера в секундах.

MPI

`int MPI_Get_processor_name(char *name, int *len)`

Возвращает в строке **`name`** имя узла, на котором запущен вызвавший процесс. В переменной **`len`** возвращается количество символов в имени, не превышающее константы **`MPI_MAX_PROCESSOR_NAME`**.

MPI

Определение характеристик системного таймера:

```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 100
int main(int argc, char **argv)
{
   double time_start, time_finish, tick;
   int rank, i;
   int len;
   char *name;
   name =
(char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Get_processor_name(name, &len);
```

MPI

```
   tick = MPI_Wtick();
   time_start = MPI_Wtime();
   for (i = 0; i<NTIMES; i++)
      time_finish = MPI_Wtime();
   printf ("processor %s, process %d: tick= %lf, time=
%lf\n", name, rank, tick, (time_finish-
time_start)/NTIMES);
   MPI_Finalize();
}
```

MPI

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

Блокирующая посылка массива **`buf`** с идентификатором **`msgtag`**, состоящего из **`count`** элементов типа **`datatype`**, процессу с номером **`dest`** в коммуникаторе **`comm`**.

MPI

Типы данных:

- **`MPI_INT – int`**
- **`MPI_SHORT – short`**
- **`MPI_LONG - long`**
- **`MPI_FLOAT – float`**
- **`MPI_DOUBLE – double`**
- **`MPI_CHAR – char`**
- **`MPI_BYTE –`** 8 бит
- **`MPI_PACKED –`** тип для упакованных данных.

Все типы данных перечислены в файле **`mpi.h`**.

MPI

Модификации процедуры **`MPI_Send`**:

- **`MPI_Bsend`** — передача сообщения с буферизацией.
- **`MPI_Ssend`** — передача сообщения с синхронизацией.
- **`MPI_Rsend`** — передача сообщения по готовности.

MPI

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

Назначение массива **`buf`** размера **`size`** для использования при посылке сообщений с буферизацией. В каждом процессе может быть только один такой буфер.

Размер массива, выделяемого для буферизации, должен превосходить общий размер сообщения как минимум на величину, определяемую константой **`MPI_BSEND_OVERHEAD`**.

MPI

`int MPI_Buffer_detach(buf, size)`

Освобождение массива **`buf`** для других целей. Процесс блокируется до того момента, когда все сообщения уйдут из данного буфера.

MPI

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

Блокирующий приём в буфер **`buf`** не более **`count`** элементов сообщения типа **`datatype`** с идентификатором **`msgtag`** от процесса с номером **`source`** в коммуникаторе **`comm`** с заполнением структуры атрибутов приходящего сообщения **`status`**.

MPI

Вместо аргументов **source** и **msgtag** можно использовать константы:

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

MPI

Параметры принятого сообщения всегда можно определить по соответствующим элементам структуры **status**:

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

MPI

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.

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

MPI

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

По значению параметра **status** определяет число **count** уже принятых (после обращения к **MPI_Recv**) или принимаемых (после обращения к **MPI_Probe** или **MPI_Iprobe**) элементов сообщения типа **datatype**.

MPI

```
int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)
```

Получение в параметре **status** информации о структуре ожидаемого сообщения с блокировкой. Возврата не произойдёт, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения.

MPI

Тупиковые ситуации (deadlock):

процесс 0:	процесс 1:
Recv(1)	**Recv(0)**
Send(1)	**Send(0)**

процесс 0:	процесс 1:
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

MPI

Разрешение тупиковых ситуаций:

1.

процесс 0: процесс 1:

Send(1) Recv(0)

Recv(1) Send(0)

2. Использование неблокирующих операций

3. Использование функции **MPI_Sendrecv**

MPI

Обмен сообщениями чётных и нечётных процессов:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
   int size, rank, a, b;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   a = rank;
   b = -1;
```

MPI

```
   if((rank%2) == 0){
      if(rank<size-1)
         MPI_Send(&a, 1, MPI_INT, rank+1, 5,
MPI_COMM_WORLD);
   }
   else
      MPI_Recv(&b, 1, MPI_INT, rank-1, 5, MPI_COMM_WORLD,
&status);
   printf("process %d a = %d, b = %d\n", rank, a, b);
   MPI_Finalize();
}
```

MPI

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

Неблокирующая посылка сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **request** идентифицирует пересылку.

MPI

Модификации функции **MPI_Isend**:

- **MPI_Ibsend** — передача сообщения с буферизацией;
- **MPI_Issend** — передача сообщения с синхронизацией;
- **MPI_Irsend** — передача сообщения по готовности.

MPI

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

Неблокирующий приём сообщения. Возврат из функции происходит сразу после инициализации передачи. Переменная **request** идентифицирует пересылку.

MPI

Сообщение, отправленное любой из процедур **MPI_Send**, **MPI_Isend** и любой из трёх их модификаций, может быть принято любой из процедур **MPI_Recv** и **MPI_Irecv**.

До завершения неблокирующей операции не следует записывать в используемый массив данных!

MPI

int MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status)

Получение информации о структуре ожидаемого сообщения без блокировки. В аргументе **flag** возвращается значение **1**, если сообщение с подходящими атрибутами уже может быть принято.

MPI

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Ожидание завершения асинхронной операции, ассоциированной с идентификатором **request**. Для неблокирующего приёма определяется параметр **status**. **request** устанавливается в значение **MPI_REQUEST_NULL.**

MPI

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

Ожидание завершения **count** асинхронных операций, ассоциированных с идентификаторами **requests**. Для неблокирующих приёмов определяются параметры в массиве **statuses**.

MPI

Обмен по кольцевой топологии при помощи неблокирующих операций:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, prev, next;
   int buf[2];
   MPI_Request reqs[4];
   MPI_Status stats[4];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

MPI

```
   prev = rank - 1;
   next = rank + 1;
   if(rank==0) prev = size - 1;
   if(rank==size - 1) next = 0;
   MPI_Irecv(&buf[0], 1, MPI_INT, prev, 5, MPI_COMM_WORLD,
&reqs[0]);
   MPI_Irecv(&buf[1], 1, MPI_INT, next, 6, MPI_COMM_WORLD,
&reqs[1]);
   MPI_Isend(&rank, 1, MPI_INT, prev, 6, MPI_COMM_WORLD,
&reqs[2]);
   MPI_Isend(&rank, 1, MPI_INT, next, 5, MPI_COMM_WORLD,
&reqs[3]);
   MPI_Waitall(4, reqs, stats);
   printf("process %d prev = %d next=%d\n", rank, buf[0],
buf[1]);
   MPI_Finalize();
}
```

MPI

```
int MPI_Waitany(int count,
MPI_Request *requests, int
*index, MPI_Status *status)
```

Ожидание завершения одной из **count** асинхронных операций, ассоциированных с идентификаторами **requests**. Для неблокирующего приёма определяется параметр **status**.

MPI

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 завершённой операции.

MPI

```
int MPI_Waitsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Ожидание завершения хотя бы одной из **incount** асинхронных операций, ассоциированных с идентификаторами **requests**.

MPI

Параметр **outcount** содержит число завершённых операций, а первые **outcount** элементов массива **indexes** содержат номера элементов массива **requests** с их идентификаторами.

Первые **outcount** элементов массива **statuses** содержат параметры завершённых операций (для неблокирующих приёмов).

MPI

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

Проверка завершённости асинхронной операции, ассоциированной с идентификатором **request**. В параметре **flag** возвращается значение **1**, если операция завершена.

MPI

```
int MPI_Testall(int count,
MPI_Request *requests, int
*flag, MPI_Status *statuses)
```

Проверка завершённости **count** асинхронных операций, ассоциированных с идентификаторами **requests**.

MPI

```
int MPI_Testany(int count,
MPI_Request *requests, int
*index, int *flag, MPI_Status
*status)
```

В параметре **flag** возвращается значение **1**, если хотя бы одна из операций асинхронного обмена завершена.

MPI

```
int MPI_Testsome(int incount,
MPI_Request *requests, int
*outcount, int *indexes,
MPI_Status *statuses)
```

Аналог **MPI_Waitsome**, но возврат происходит немедленно. Если ни одна из операций не завершилась, то значение **outcount** будет равно нулю.

MPI

```
int MPI_Send_init(void *buf, int
count, MPI_Datatype datatype,
int dest, int msgtag, MPI_Comm
comm, MPI_Request *request)
```

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

MPI

Модификации функции **MPI_Send_init**:

- **MPI_Bsend_init** — формирование запроса на передачу сообщения с буферизацией;
- **MPI_Ssend_init** — формирование запроса на передачу сообщения с синхронизацией;
- **MPI_Rsend_init** — формирование запроса на передачу сообщения по готовности.

MPI

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

Формирование отложенного запроса на приём сообщения. Сама операция приёма не начинается!

MPI

```
int MPI_Start(MPI_Request
*request)
```

Инициализация отложенного запроса на выполнение операции обмена, соответствующей значению параметра **request**. Операция запускается как неблокирующая.

MPI

`int MPI_Startall(int count, MPI_Request *requests)`

Инициализация **`count`** отложенных запросов на выполнение операций обмена, соответствующих значениям первых **`count`** элементов массива **`requests`**. Операции запускаются как неблокирующие.

MPI

Сообщение, отправленное при помощи отложенного запроса, может быть принято любой из процедур **`MPI_Recv`** и **`MPI_Irecv`**, и наоборот.

По завершении отложенного запроса значение параметра **`request (requests)`** сохраняется и может использоваться в дальнейшем!

MPI

`int MPI_Request_free(MPI_Request *request)`

Удаляет структуры данных, связанные с параметром **`request`**. **`request`** устанавливается в значение **`MPI_REQUEST_NULL`**. Если операция, связанная с этим запросом, уже выполняется, то она завершится.

MPI

Схема итерационного метода с обменом по кольцевой топологии при помощи отложенных запросов:

```
prev = rank - 1;
next = rank + 1;
if (rank == 0) prev = size - 1;
if (rank == (size - 1)) next = 0;
MPI_Recv_init(&rbuf[0], 1, MPI_FLOAT, prev, tag1,
MPI_COMM_WORLD, &reqs[0]);
MPI_Recv_init(&rbuf[1], 1, MPI_FLOAT, next, tag2,
MPI_COMM_WORLD, &reqs[1]);
MPI_Send_init(&sbuf[0], 1, MPI_FLOAT, prev, tag2,
MPI_COMM_WORLD, &reqs[2]);
MPI_Send_init(&sbuf[1], 1, MPI_FLOAT, next, tag1,
MPI_COMM_WORLD, &reqs[3]);
```

MPI

```
for(…){
 sbuf[0]=…;
 sbuf[1]=…;
 MPI_Startall(4, reqs);
 ...
 MPI_Waitall(4, reqs, stats);
 ...
}
MPI_Request_free(&reqs[0]);
MPI_Request_free(&reqs[1]);
MPI_Request_free(&reqs[2]);
MPI_Request_free(&reqs[3]);
```

MPI

`int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype, int source, int rtag, MPI_Comm comm, MPI_Status *status)`

MPI

Совмещённые приём и передача сообщений с блокировкой. Буферы передачи и приёма не должны пересекаться. 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

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int stag, int source, int rtag, MPI_comm comm, MPI_Status *status)
```

Совмещённые приём и передача сообщений с блокировкой через общий буфер **`buf`**.

MPI

Обмен по кольцевой топологии при помощи процедуры **`MPI_Sendrecv`**:

```
prev = rank – 1;
next = rank + 1;
if (rank == 0) prev = size – 1;
if (rank == (size – 1)) next = 0;
MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev, tag2, &rbuf[0], 1, MPI_FLOAT, next, tag2, MPI_COMM_WORLD, &status1);
MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next, tag1, &rbuf[1], 1, MPI_FLOAT, prev, tag1, MPI_COMM_WORLD, &status2);
```

MPI

Специальное значение **`MPI_PROC_NULL`** для несуществующего процесса. Операции с таким процессом завершаются немедленно с кодом завершения **`MPI_SUCCESS`**.

```
next = rank + 1;
if(rank == (size - 1)) next=MPI_PROC_NULL;
MPI_Send (&buf, 1, MPI_FLOAT, next, tag, MPI_COMM_WORLD);
```

MPI

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора*!

Как и для блокирующих процедур, возврат означает то, что разрешён свободный доступ к буферу приёма или посылки.

Сообщения, вызванные коллективными операциями, не пересекутся с другими сообщениями.

MPI

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).

MPI

`int MPI_Barrier(MPI_Comm comm)`

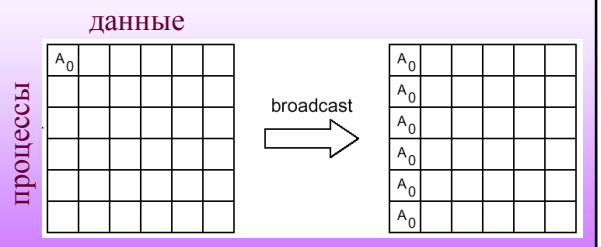
Работа процессов блокируется до тех пор, пока все оставшиеся процессы коммуникатора **`comm`** не выполнят эту процедуру. Все процессы должны вызвать **`MPI_Barrier`**, хотя реально исполненные различными процессами коммуникатора вызовы могут быть расположены в разных местах программы.

MPI

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

Рассылка **`count`** элементов данных типа **`datatype`** из массива **`buf`** от процесса **`root`** всем процессам данного коммуникатора **`comm`**, включая сам рассылающий процесс. Значения параметров **`count`**, **`datatype`**, **`root`** и **`comm`** должны быть одинаковыми у всех процессов.

MPI



MPI

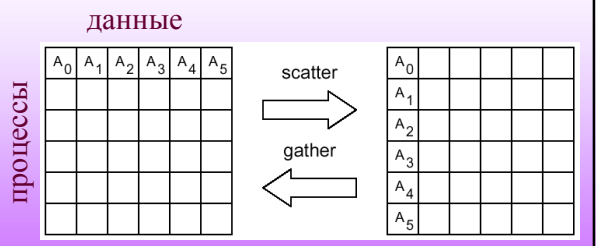
`int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)`

Сборка **`scount`** элементов данных типа **`stype`** из массивов **`sbuf`** со всех процессов коммуникатора **`comm`** в буфер **`rbuf`** процесса **`root`**. Данные сохраняются в порядке возрастания номеров процессов.

MPI

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 от каждого процесса.

MPI



MPI

```
int MPI_Gatherv(void *sbuf, int
scount, MPI_Datatype stype, void
*rbuf, int *rcounts, int
*displs, MPI_Datatype rtype, int
root, MPI_Comm comm)
```

Сборка различного количества данных из массивов **sbuf**. Порядок расположения задает массив **displs**.

MPI

rcounts – целочисленный массив, содержащий количество элементов, передаваемых от каждого процесса (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

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

MPI

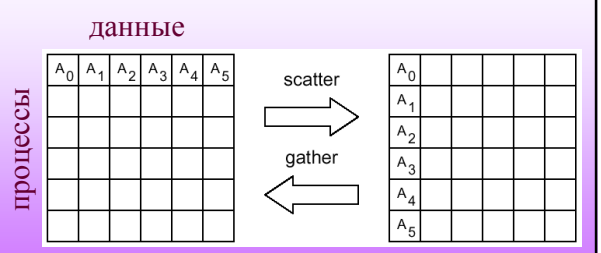
```
int MPI_Scatter(void *sbuf, int
scount, MPI_Datatype stype, void
*rbuf, int rcount, MPI_Datatype
rtype, int root, MPI_Comm comm)
```

Рассылка по **scount** элементов данных типа **stype** из массива **sbuf** процесса **root** в массивы **rbuf** всех процессов коммуникатора **comm**, включая сам процесс **root**. Данные рассылаются в порядке возрастания номеров процессов.

MPI

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.

MPI



MPI

```
float sbuf[SIZE][SIZE], rbuf[SIZE];
if(rank == 0)
  for(i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      sbuf[i][j]=…;
if (numtasks == SIZE)
  MPI_Scatter(sbuf, SIZE, MPI_FLOAT, rbuf,
SIZE, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

MPI

```
int MPI_Scatterv(void *sbuf, int
*scounts, int *displs,
MPI_Datatype stype, void *rbuf,
int rcount, MPI_Datatype rtype,
int root, MPI_Comm comm)
```

Рассылка различного количества данных из массива **sbuf**. Начало рассылаемых порций задает массив **displs**.

MPI

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

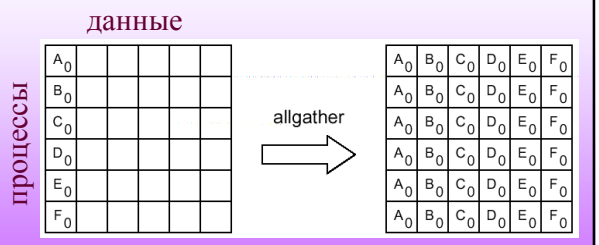
displs – целочисленный массив, содержащий смещения относительно начала массива **sbuf** (индекс равен рангу адресата, длина равна числу процессов в коммуникаторе).

MPI

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

Сборка данных из массивов **sbuf** со всех процессов коммуникатора **comm** в буфере **rbuf** каждого процесса. Данные сохраняются в порядке возрастания номеров процессов.

MPI



MPI

```
int MPI_Allgatherv(void *sbuf,
int scount, MPI_Datatype stype,
void *rbuf, int *rcounts, int
*displs, MPI_Datatype rtype,
MPI_Comm comm)
```

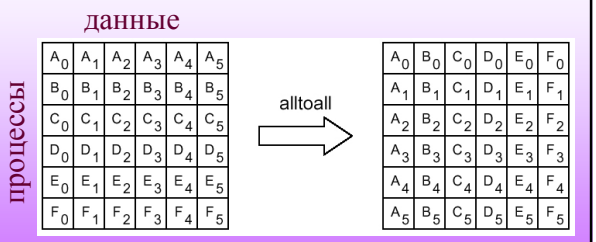
Сборка на всех процессах различного количества данных из **sbuf**. Порядок расположения задаёт массив **displs**.

MPI

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

Рассылка каждым процессом коммуникатора **comm** различных порций данных всем другим процессам. **j**-й блок массива **sbuf** процесса **i** попадает в **i**-й блок массива **rbuf** процесса **j**.

MPI



MPI

`int MPI_Alltoallv(void* sbuf, int *scounts, int *sdispls, MPI_Datatype stype, void* rbuf, int *rcounts, int *rdispls, MPI_Datatype rtype, MPI_Comm comm)`

Рассылка со всех процессов коммуникатора **`comm`** различного количества данных всем другим процессам. Размещение данных в буфере **`sbuf`** отсылающего процесса определяется массивом **`sdispls`**, а в буфере **`rbuf`** принимающего процесса – массивом **`rdispls`**.

MPI

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

Выполнение **`count`** независимых глобальных операций **`op`** над соответствующими элементами массивов **`sbuf`**. Результат операции над **`i`**-ми элементами массивов **`sbuf`** получается в **`i`**-ом элементе массива **`rbuf`** процесса **`root`**.

MPI

Типы предопределённых глобальных операций:

`MPI_MAX, MPI_MIN` – максимальное и минимальное значения;

`MPI_SUM, MPI_PROD` – глобальная сумма и глобальное произведение;

`MPI_LAND, MPI_LOR, MPI_LXOR` – логические “И”, “ИЛИ”, искл. “ИЛИ”;

`MPI_BAND, MPI_BOR, MPI_BXOR` – побитовые “И”, “ИЛИ”, искл. “ИЛИ”.

MPI

`int MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

Выполнение **`count`** независимых глобальных операций **`op`** над соответствующими элементами массивов **`sbuf`**. Результат получается в массиве **`rbuf`** каждого процесса.

MPI

```
for(i=0; i<n; i++) s[i]=0.0;
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    s[i]=s[i]+a[i][j];
MPI_Allreduce(s, r, n, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение $\sum_i$ **rcounts(i)** независимых глобальных операций **op** над соответствующими элементами массивов **sbuf**.

MPI

Сначала выполняются глобальные операции, затем результат рассылается по процессам.

i-й процесс получает **rcounts(i)** значений результата и помещает в массив **rbuf.**

MPI

```
int MPI_Scan(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Выполнение **count** независимых частичных глобальных операций **op** над соответствующими элементами массивов **sbuf**.

MPI

i-й процесс выполняет глобальную операцию над соответствующими элементами массива **sbuf** процессов **0…i** и помещает результат в массив **rbuf.**

Окончательный результат глобальной операции получается в массиве **rbuf** последнего процесса.

MPI

```
int MPI_Op_create (MPI_User_function *func, int commute, MPI_Op *op)
```

Создание пользовательской глобальной операции. Если **commute=1**, то операция должна быть коммутативной и ассоциативной. Иначе порядок фиксируется по увеличению номеров процессов.

MPI

```
typedef void MPI_User_function (void *invec, void *inoutvec, int *len, MPI_Datatype type)
```

Интерфейс пользовательской функции.

```
int MPI_Op_free(MPI_Op *op)
```

Уничтожение пользовательской глобальной операции.

MPI

```
#include <stdio.h>
#include "mpi.h"
#define n 1000
void smod5(void *in, void *inout, int *l, MPI_Datatype
*type){
   int i;
   for(i=0; i<*l; i++) ((int*)inout)[i] = (((int*)in)[i]
+ ((int*)inout)[i])%5;
}
int main(int argc, char **argv)
{
   int rank, size, i;
   int a[n];
   int b[n];
```

MPI

```
   MPI_Op op;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for(i=0; i<n; i++) a[i] = i + rank +1;
   printf("process %d a[0] = %d\n", rank, a[0]);
   MPI_Op_create(&smod5, 1, &op);
   MPI_Reduce(a, b, n, MPI_INT, op, 0, MPI_COMM_WORLD);
   MPI_Op_free(&op);
   if(rank==0) printf("b[0] = %d\n", b[0]);
   MPI_Finalize();
}
```

MPI

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

Разбиение коммуникатора **comm** на несколько по числу значений параметра **color**. В одну подгруппу попадают процессы с одним значением **color**. Процессы с бóльшим значением **key** получат больший ранг.

MPI

Процессы, которые не должны войти в новые группы, указывают в качестве **color** константу **MPI_UNDEFINED**. Им в параметре **newcomm** вернется значение **MPI_COMM_NULL**.

```
MPI_Comm_split(MPI_COMM_WORLD, rank%3, rank,
new_comm)
```

MPI

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

Удаление коммуникатора **comm**. Переменной **comm** присваивается значение **MPI_COMM_NULL**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int rank, size, rank1;
   MPI_Comm comm_revs;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_split(MPI_COMM_WORLD, 1, size-rank,
&comm_revs);
   MPI_Comm_rank(comm_revs, &rank1);
   printf("rank = %d rank1 = %d\n", rank, rank1);
   MPI_Comm_free(&comm_revs);
   MPI_Finalize();
}
```

MPI

Сообщение – массив однотипных данных, расположенных в последовательных ячейках памяти.

Для пересылки разнотипных данных можно использовать:

•Производные типы данных

•Упаковку данных

MPI

Производные типы данных создаются во время выполнения программы с помощью подпрограмм-конструкторов.

Создание типа:

•Конструирование типа

•Регистрация типа

MPI

Производный тип данных характеризуется последовательностью базовых типов и набором значений смещения относительно начала буфера обмена.

Смещения могут быть как положительными, так и отрицательными, не требуется их упорядоченность.

MPI

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

Создание нового типа данных **newtype**, состоящего из **count** последовательно расположенных элементов базового типа данных **type**.

```
MPI_Type_contiguous(5, MPI_INT, &newtype);
```

MPI

```
int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype type, MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Следующий блок начинается через **stride** элементов после начала предыдущего.

MPI

```
count=2;
blocklen=3;
stride=5;
MPI_Type_vector(count, blocklen, stride, MPI_DOUBLE, &newtype);
```

Создание нового типа данных (тип элемента, количество элементов от начала буфера):

{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 1), (**MPI_DOUBLE**, 2),

(**MPI_DOUBLE**, 5), (**MPI_DOUBLE**, 6), (**MPI_DOUBLE**, 7)}

MPI

```
int MPI_Type_hvector(int count,
int blocklen, MPI_Aint stride,
MPI_Datatype type, MPI_Datatype
*newtype)
```

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.

MPI

```
int
MPI_Type_create_indexed_block(in
t count, int blocklen, int
displs[], MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklen** элементов базового типа данных **type**. Смещения блоков с начала буфера посылки в количестве элементов базового типа данных **type** задаются в массиве **displs**.

MPI

```
int MPI_Type_indexed(int count,
int *blocklens, int *displs,
MPI_Datatype type, MPI_Datatype
*newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** элементов с начала буфера.

MPI

```
for(i=0; i<n; i++){
   blocklens[i]=n-i;
   displs[i]=(n+1)*i;
}
MPI_Type_indexed(n, blocklens, displs,
MPI_DOUBLE, &newtype)
```

Создание нового типа данных для описания верхнетреугольной матрицы.

MPI

```
int MPI_Type_hindexed(int count,
int *blocklens, MPI_Aint
*displs, MPI_Datatype type,
MPI_Datatype *newtype)
```

Создание нового типа данных **newtype**, состоящего из **count** блоков по **blocklens[i]** элементов базового типа данных. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
int MPI_Type_struct(int count,
int *blocklens, MPI_Aint
*displs, MPI_Datatype *types,
MPI_Datatype *newtype)
```

Создание структурного типа данных из **count** блоков по **blocklens[i]** элементов типа **types[i]**. **i**-ый блок начинается через **displs[i]** байт с начала буфера.

MPI

```
blocklens[0]=3;
blocklens[1]=2;
types[0]=MPI_DOUBLE;
types[1]=MPI_CHAR;
displs[0]=0;
displs[1]=24;
MPI_Type_struct(2, blocklens, displs, types,
&newtype);
```

Создание нового типа данных (тип элемента, количество байт от начала буфера):
{(**MPI_DOUBLE**, 0), (**MPI_DOUBLE**, 8), (**MPI_DOUBLE**, 16),
(**MPI_CHAR**, 24), (**MPI_CHAR**, 25)}

MPI

```
int MPI_Type_commit(MPI_Datatype
*datatype)
```

Регистрация созданного производного типа данных **datatype**. После регистрации этот тип данных можно использовать в операциях обмена.

MPI

```
int MPI_Type_free(MPI_Datatype
*datatype)
```

Аннулирование производного типа данных **datatype**. **datatype** устанавливается в значение **MPI_DATATYPE_NULL**. Производные от **datatype** типы данных остаются. Предопределённые типы данных не могут быть аннулированы.

MPI

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

Определение абсолютного байт-адреса **address** размещения массива **location** в оперативной памяти компьютера. Адрес отсчитывается от некоторого базового адреса, значение которого содержится в системной константе **MPI_BOTTOM**.

MPI

```
blocklens[0] = 1;
blocklens[1] = 1;
types[0] = MPI_DOUBLE;
types[1] = MPI_CHAR;
MPI_Address(dat1, &displs[0]);
MPI_Address(dat2, &displs[1]);
MPI_Type_struct(2, blocklens, displs, types,
&newtype);
MPI_Type_commit(&newtype);
MPI_Send(MPI_BOTTOM, 1, newtype, dest, tag,
MPI_COMM_WORLD);
```

MPI

```
int MPI_Type_size(MPI_Datatype
datatype, int *size)
```

Определение размера типа **datatype** в байтах (объёма памяти, занимаемого одним элементом данного типа).

MPI

```
int
MPI_Type_get_extent(MPI_Datatype
datatype, MPI_Aint *lb, MPI_Aint
*extent)
```

Для элемента типа данных **datatype** определяет смещение от начала буфера данных нижней границы **lb** и диапазон **extent** (разницу между верхней и нижней границами) в байтах.

MPI

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

Упаковка **incount** элементов типа **datatype** из массива **inbuf** в массив **outbuf** со сдвигом **position** байт. **outbuf** должен содержать хотя бы **outsize** байт.

MPI

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

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

Распаковка из массива **inbuf** со сдвигом **position** байт в массив **outbuf** **outcount** элементов типа **datatype**.

MPI

```
int MPI_Pack_size(int incount,
MPI_Datatype datatype, MPI_Comm
comm, int *size)
```

Определение необходимого объёма памяти (в байтах) для упаковки **incount** элементов типа **datatype**.

MPI

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
   int size, rank, position, i;
   float a[10];
   char b[10], buf[100];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for(i = 0; i<10; i++){
      a[i] = rank + 1.0;
      if(rank==0) b[i]='a';
      else b[i] = 'b';
   }
   position=0;
```

MPI

```
   if(rank==0){
      MPI_Pack(a, 10, MPI_FLOAT, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Pack(b, 10, MPI_CHAR, buf, 100, &position,
MPI_COMM_WORLD);
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
   } else{
      MPI_Bcast(buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, a, 10, MPI_FLOAT,
MPI_COMM_WORLD);
      MPI_Unpack(buf, 100, &position, b, 10, MPI_CHAR,
MPI_COMM_WORLD);
   }
   for(i = 0; i<10; i++) printf("process %d a=%f b=%c\n",
rank, a[i], b[i]);
   MPI_Finalize();}
```

MPI

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. Технологии параллельного программирования MPI и OpenMP: Учебник М.: Издательство Московского университета, 2012. 334 с. (Серия Суперкомпьютерное образование). ISBN 978-5-211-06343-3
4. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие.-М.: Изд-во МГУ, 2004.
5. Антонов А.С. Введение в параллельные вычисления (методическое пособие). М.: Изд-во Физического факультета МГУ, 2002.

MPI

Литература

6. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных вычислительных систем. Ростов-на-Дону: Издательство ООО "ЦВВР", 2003.
7. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие. Минск: БГУ, 2002.
8. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. СПб.: БХВ-Петербург, 2002.
9. Корнеев В.Д. Параллельное программирование в MPI. Новосибирск: Изд-во СО РАН, 2000.