

Асинхронные процедуры MPI.

В отличие от блокирующих процедур, возврат из процедур данной группы происходит сразу после вызова без остановки работы процессов. Для получения информации от асинхронных процедур требуется вызов дополнительной процедуры, которая проверит завершилась ли процедура или дождется её завершения.

Асинхронные процедуры MPI.

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

Неблокирующая посылка сообщения.

Переменная request идентифицирует пересылку, позволяет узнать статус отправки в дальнейшем.

Асинхронные процедуры MPI.

С помощью процедур семейства MPI_Wait и MPI_Test можно определить момент времени, когда возможно использовать повторно буфер buf, без опасения испортить передаваемое сообщение.

Асинхронные процедуры MPI.

Модификации функции `MPI_Isend`:

- `MPI_Ibsend` — передача сообщения с буферизацией;
- `MPI_Issend` — передача сообщения с синхронизацией;
- `MPI_Irsend` — передача сообщения по ГОТОВНОСТИ.

Асинхронные процедуры 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.

До завершения неблокирующего приема не следует записывать в массив `buf` данные.

Асинхронные процедуры MPI.

Передача с помощью процедур MPI_Send, MPI_Isend и любой из трех модификаций может быть принята любой из процедур MPI_Recv и MPI_Irecv.

Асинхронные процедуры MPI.

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

Позволяет получить информацию о структуре ожидаемого сообщения без блокировки. В аргументе `flag` принимает значение 1, если сообщение с подходящими атрибутами уже может быть принято, и значение 0, если сообщения с указанными атрибутами еще нет.

Асинхронные процедуры MPI.

```
int MPI_Wait(MPI_Request *request,  
MPI_Status *status)
```

Процедура ожидает завершения асинхронной операции, ассоциированной с идентификатором request. Для неблокирующего приёма определяется параметром status. После вызова параметр request устанавливается в значение MPI_REQUEST_NULL.

Асинхронные процедуры MPI.

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

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

Асинхронные процедуры MPI.

Если во время выполнения асинхронных операций обмена возникают ошибки, то будут заполнены поля ошибок структуры `status`. После выполнения асинхронных процедур обмена структура `request` примет значение `MPI_REQUEST_NULL`.

Асинхронные процедуры MPI.

Обмен по кольцевой топологии при помощи неблокирующих операций:

```
int main(int argc, char **argv)
{
    int rank, size, prev, next, 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);
    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();
}
```

Асинхронные процедуры MPI.

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

Процедура ожидает завершения одной из `count` асинхронных операций, ассоциированных с идентификаторами `requests`. Для неблокирующего приёма определяется параметр `status`.

Асинхронные процедуры MPI.

Если к моменту вызова завершились несколько из ожидаемых асинхронных операций, то случайным образом будет выбрана одна из них. Параметр `index` содержит номер элемента в массиве `requests`, содержащего идентификатор завершённой операции.

Соответствующий элемент `requests` устанавливается в `MPI_REQUEST_NULL`.

Асинхронные процедуры MPI.

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

Процедура ожидает завершения хотя бы одной из `incount` асинхронных операций, ассоциированных с идентификаторами `requests`.

Асинхронные процедуры MPI.

Аргумент `outcount` есть число завершённых операций, а первые `outcount` элементов `indexes` содержат номера элементов `requests` с их идентификаторами.

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

Соответствующий элемент `requests` устанавливается в `MPI_REQUEST_NULL`.

Асинхронные процедуры 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, и значение 0 – в противном случае.

Асинхронные процедуры MPI.

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`.

Асинхронные процедуры MPI.

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

Проверка завершённости count асинхронных операций, ассоциированных с идентификаторами requests. В параметре flag возвращает 1, если все указанные операции завершены. Параметры принимаемых сообщений будут в массиве statuses.

Асинхронные процедуры MPI.

Если какая-либо из операций не завершилась, то возвращается 0, и определенность элементов массива `statuses` не гарантируется. После выполнения процедуры соответствующие элементы параметра `requests` устанавливаются в значение `MPI_REQUEST_NULL`.

Асинхронные процедуры 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, при этом `index` содержит номер соответствующего элемента в массиве `requests`, а `status` – параметры принимаемого сообщения.

Асинхронные процедуры MPI.

Если ни одна из операций не завершилась, то в параметре `flag` будет возвращено значение 0. Если к моменту вызова завершились несколько ожидаемых операций, то случайным образом будет выбрана одна из них. После выполнения процедуры соответствующий элемент параметра `requests` устанавливается в значение `MPI_REQUEST_NULL`.

Отложенные запросы на взаимодействие.

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

Позволяет получить информацию об асинхронной операции, ассоциированной с идентификатором `request`, без освобождения соответствующих структур данных. В параметре `flag` возвращается значение 1, если операция завершена, и значение 0 – в противном случае.

Асинхронные процедуры MPI.

`int MPI_Cancel(MPI_Request *request)` — запускает процесс отмены асинхронной операции, ассоциированной с идентификатором `request`. Возврат не означает, что соответствующая операция отменена. После вызова необходимо вызвать одну из функций `MPI_Request_free`, `MPI_Wait`, `MPI_Test` или их производных.

Асинхронные процедуры MPI.

```
int MPI_Test_cancelled(  
MPI_Status *status, int *flag)
```

Возвращает в параметре `flag` значение 1, если операция, ассоциированная с идентификатором `request`, была успешно отменена, и 0 – в противном случае. Если операция неблокирующего приема могла быть отменена, то прежде чем обращаться к полям `status`, нужно проверить отмену при помощи `MPI_Test_cancelled`.

Асинхронные процедуры 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.

Отложенные запросы на взаимодействие.

Как правило, эти процедуры тоже не блокирующие. Отличие в том, что при использовании отложенных запросов операция не начинается пока не будет вызвана процедура запуска.

Отложенные запросы на взаимодействие.

```
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 данных buf. Операция не начинается.

Отложенные запросы на взаимодействие.

Возможно использование любых модификаций `MPI_Send`:

- `MPI_Bsend_init` — формирование запроса на посылку с буфером.
- `MPI_Ssend_init` — формирование посылку с синхронизацией.
- `MPI_Rsend_init` — формирование запроса на посылку по ГОТОВНОСТИ.

Отложенные запросы на взаимодействие.

```
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 прием данных в buf. Операция не начинается.

Отложенные запросы на взаимодействие.

Для начала отправки/приема используется процедура

`MPI_Start(MPI_Request* request)` — операция запускается как не блокирующая.

Процедура

`MPI_Startall(int count, MPI_Request* requests)` — используется для старта сразу нескольких операций.

Отложенные запросы на взаимодействие.

Когда операции выполнены и `request` больше не требуется, необходимо его освободить с помощью процедуры

`int MPI_Request_free(MPI_Request* request)` — `request` установится в значение `MPI_REQUEST_NULL`. Если операция, связанная с этим запросом еще выполняется, то MPI завершит ее.

Отложенные запросы на взаимодействие.

Преимущества:

Отложенные пересылкам экономят часть времени, требующегося на подготовку процедур отправки/приема.

Недостатки:

Вызов большого количества приемов/передач может вызвать «взрывную» нагрузку на сеть и привести к низкой производительности системы.

Отложенные запросы на взаимодействие.

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.

Отложенные запросы на взаимодействие.

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
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, 5, MPI_COMM_WORLD, reqs[0]);
MPI_Recv_init(rbuf[1], 1, MPI_FLOAT, next, 6, MPI_COMM_WORLD, reqs[1]);
MPI_Send_init(sbuf[0], 1, MPI_FLOAT, prev, 6, MPI_COMM_WORLD, reqs[2]);
MPI_Send_init(sbuf[1], 1, MPI_FLOAT, next, 5, MPI_COMM_WORLD, reqs[3]);
for(i=...){
    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]);
```

Совмещенные функции.

Совмещенные функции весьма полезны для выполнения сдвига по цепи процессов. Если для такого сдвига были использованы блокирующие приемы и передачи, тогда нужно корректно упорядочить эти приемы и передачи (например, четные процессы передают, затем принимают, нечетные процессы сначала принимают, затем передают) так, чтобы предупредить циклические зависимости, которые могут привести к deadlock-у.

Когда используется процедура `MPI_Sendrecv`, коммуникационная система решает эти проблемы.

Совмещенные функции.

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

служит для блокирующего приема и отправки одновременно. Стандарт MPI гарантирует, что при использовании этой функции не случится deadlock.

Совмещенные функции.

Процедура `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)
```

Совмещенный приём и передача сообщения с блокировкой через общий буфер `buf`. MPI гарантирует, что сообщение сначала уйдёт из буфера `buf`, а затем будет принято входящее сообщение.

Совмещенные функции.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
    int rank, size, prev, next;
    int buf[2];
    MPI_Status status1;
    MPI_Status status2;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    prev = rank - 1;
    next = rank + 1;
    if(rank==0) prev = size - 1;
    if(rank==size-1) next = 0;
    MPI_Sendrecv(&rank, 1, MPI_INT, prev, 6, &buf[1], 1, MPI_INT, next, 6,
MPI_COMM_WORLD, &status2);
    MPI_Sendrecv(&rank, 1, MPI_INT, next, 5, &buf[0], 1, MPI_INT, prev, 5,
MPI_COMM_WORLD, &status1);
    printf("process %d prev=%d next=%d\n", rank, buf[0], buf[1]);
    MPI_Finalize();
}
```