

Введение в параллельные вычисления

Лекция 9. MPI. Часть 3 Коллективные операции (продолжение) MPI-3.0

КС-40, КС-44
РХТУ

Преподаватель
Митричев Иван Игоревич, к.т.н.,
ассистент кафедры ИКТ

2017

Коллективный сбор

int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

sendbuf – адрес буфера с данными;

count – число элементов типа *datatype* в буфере;

op – идентификатор операции (типа *MPI_Op*), которую нужно осуществить над пересланными данными для получения результата в буфере *recvbuf*;

root – ранг получателя в коммуникаторе **comm**;

выходной параметр:

recvbuf – указатель на буфер, где требуется получить результат.

Функция должна быть вызвана во всех процессах группы *comm* с одинаковыми значениями аргументов *root*, *comm*, *count*, *datatype*, *op*.

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Отличие от Reduce: рассылка результата операции *op* осуществляется всем процессам

Массовая рассылка одних данных всем: MPI_Bcast

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

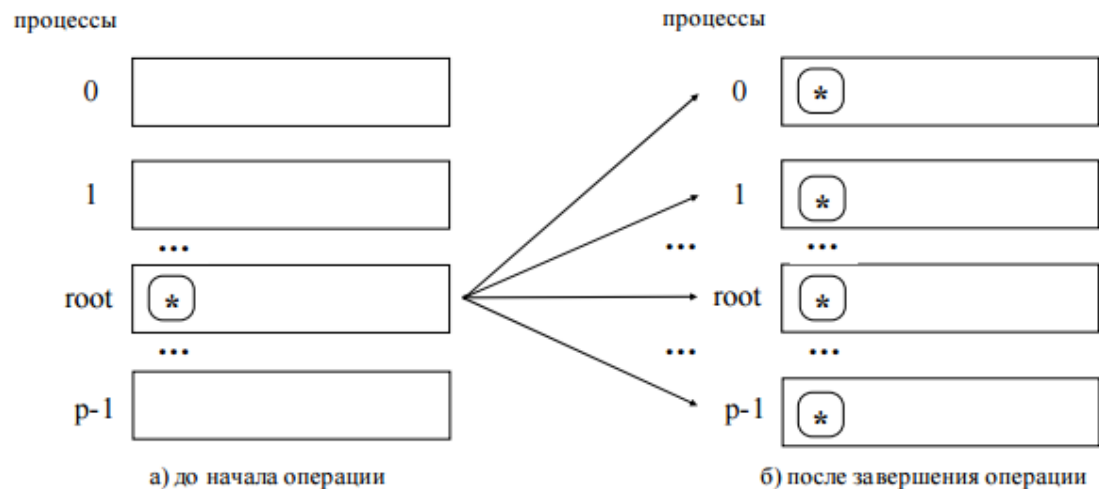
buf – адрес начала буфера послылки сообщения;

count – число передаваемых элементов в сообщении;

source – номер рассылающего процесса.

**Bcast приблизительно
обратно Reduce!**

Рассылка сообщения от процесса *source* всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера *buf* процесса *source* будет скопировано в локальный буфер процесса. Значения параметров *count*, *datatype* и *source* должны быть одинаковыми у всех процессов.



Коллективное взаимодействие процессов

Пример: вычисление суммы

```
#include "mpi.h"
#define N 100
```

```
using namespace std;
```

```
int main(int argc, char **argv)
```

```
{
    double x[100], sum_x, sum_p;
```

```
    int myrank, p;
```

```
    MPI_Status status; MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD,
```

```
&myrank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
    // let's fill array only for process 0
```

```
    if (myrank == 0)
```

```
    for (int i = 0; i < N; i++)
```

```
        x[i] = 1.0;
```

```
    MPI_Bcast (x, N, MPI_DOUBLE, 0,
```

```
MPI_COMM_WORLD);
```

```
    int k = N / p;
```

```
    int i1 = k * myrank;
```

```
    int i2 = k * (myrank + 1);
```

```
    if (myrank == p - 1)
```

```
        i2 = N;
```

```
    for ( int i = i1; i < i2; i++ )
```

```
        sum_p += x[i];
```

$$S = \sum_{i=1}^n x_i$$

```
    cout << " My x " << sum_p << endl;
```

```
    if (myrank == 0 )
```

```
    {
```

```
        sum_x = sum_p;
```

```
        for ( int i = 1; i < p; i++ )
```

```
        {
```

```
            MPI_Recv(&sum_p, 1,
```

```
MPI_DOUBLE, MPI_ANY_SOURCE, 0,
```

```
MPI_COMM_WORLD, &status);
```

```
            sum_x += sum_p;
```

```
        }
```

```
    }
```

```
    else
```

```
        MPI_Send(&sum_p, 1,
```

```
MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

```
        if ( myrank == 0 ) cout << " Summa x " <<
```

```
sum_x << endl;
```

```
        MPI_Finalize();
```

```
        return 0;
```

```
    }
```

109_01.cpp

Массовый сбор от всех с объединением в общий массив

```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int  
recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Параметры:

- sendbuf – адрес буфера с посылаемыми данными;
- sendcount – число посылаемых элементов;
- sendtype – тип данных;
- recvbuf – адрес буфера для получения сообщения (используется только для root);
- recvcount – максимальное число элементов в приемном буфере;
- recvtype – тип данных;
- root – ранг получателя;
- comm – коммуникатор;

109_02.cpp



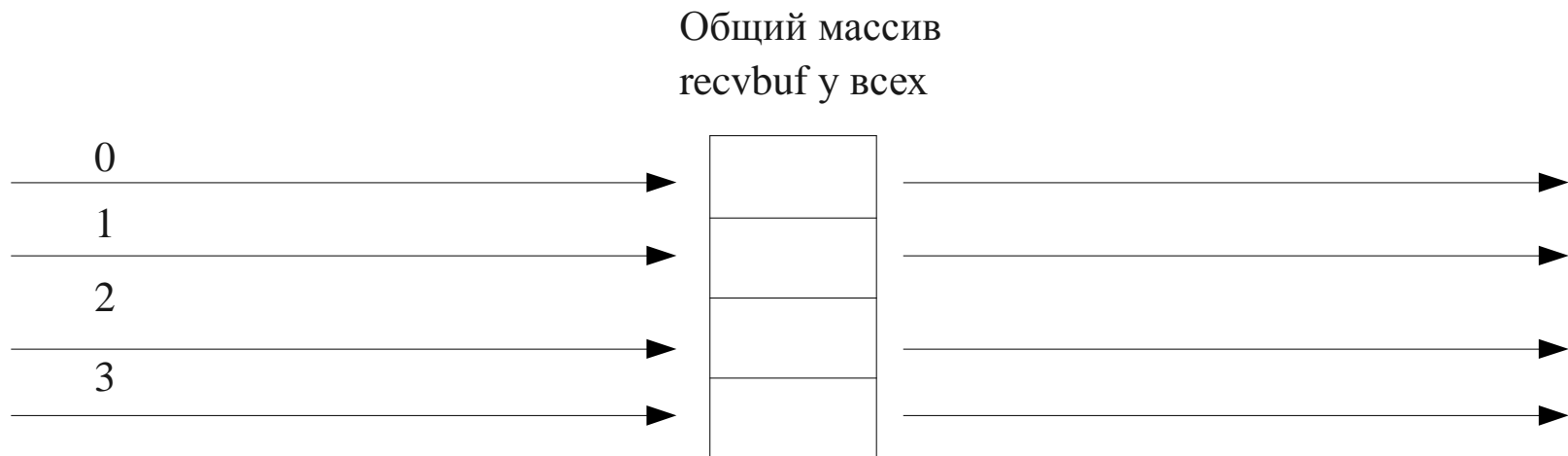
Общий массив
recvbuf у root

Массовый сбор с пересылкой всем = Результирующий массив доступен всем процессам

```
int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,  
int recvcount, MPI_Datatype recvtype, MPI_Comm comm);
```

Параметры:

- sendbuf – адрес буфера с посылаемыми данными;
- sendcount – число посылаемых элементов;
- sendtype – тип данных;
- recvbuf – адрес буфера для получения сообщения;
- recvcount – максимальное число элементов в приемном буфере;
- recvtype – тип данных;
- comm – коммунникатор;



Массовая рассылка массива: по части каждому процессу

```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype  
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,  
int root, MPI_Comm comm);
```

Параметры:

- sendbuf – адрес буфера с посылаемыми данными (используется только у root);
- sendcount – число посылаемых элементов (используется только у root);
- sendtype – тип данных;
- recvbuf – адрес буфера для получения сообщения;
- recvcount – максимальное число элементов в приемном буфере;
- recvtype – тип данных;
- root – ранг отправителя;
- comm – коммунникатор;

109_03.cpp

Scatter обратно Gather!



Общий массив sendbuf у
root делится на части

Еще о сообщениях

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

source – номер процесса-отправителя или *MPI_ANY_SOURCE*

msgtag – идентификатор ожидаемого сообщения или *MPI_ANY_TAG*

comm – идентификатор группы

OUT *status* – параметры обнаруженного сообщения

Получение информации о структуре ожидаемого сообщения с блокировкой.

Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить с помощью параметра *status*.

Подпрограмма определяет факт прихода сообщения, но реально его не принимает.

Проверка состояния

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

request – идентификатор асинхронного приема или передачи;

OUT *flag* – признак завершенности операции обмена;

OUT *status* – параметры сообщения.

Проверка завершенности асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В параметре *flag* – 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить с помощью параметра *status*.

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

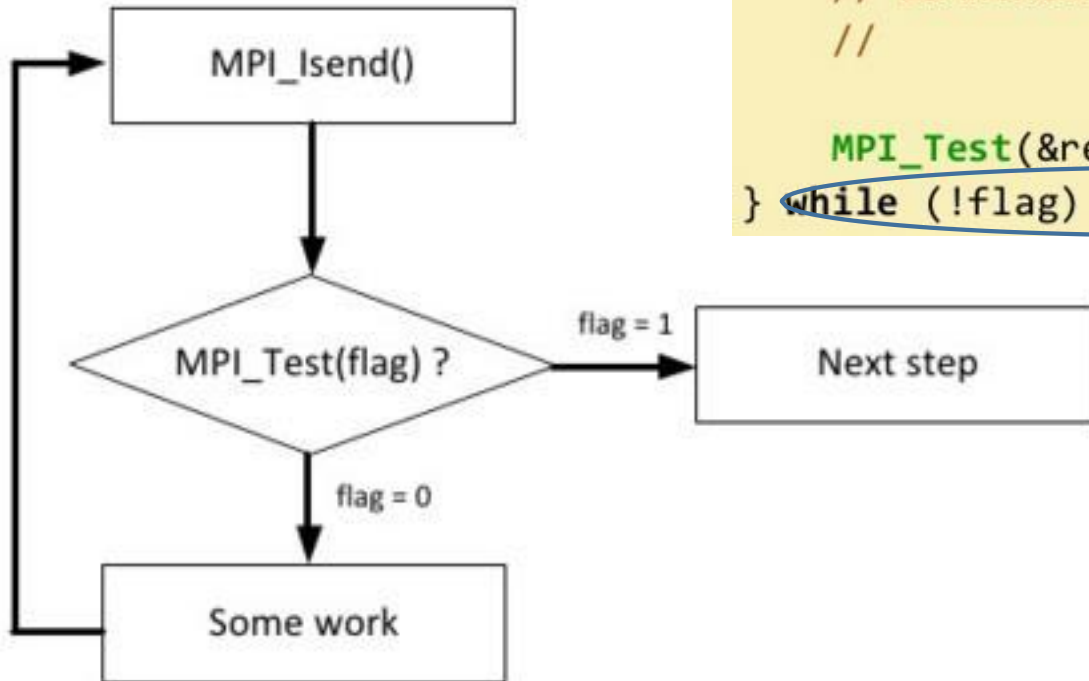
source – номер процесса-отправителя или *MPI_ANY_SOURCE*;

msgtag – идентификатор ожидаемого сообщения или *MPI_ANY_TAG*;

OUT *status* – параметры обнаруженного сообщения.

Получение информации в массиве *status* о поступлении и структуре ожидаемого сообщения без блокировки. В параметре *flag* - значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично *MPI_Probe*), и значение 0, если сообщения с указанными атрибутами еще нет.

Совмещение обменов и вычислений



```
MPI_Isend(buf, count, MPI_INT, 1, 0,  
          MPI_COMM_WORLD, &req);  
  
do {  
    //  
    // Вычисления ... (не использовать buf)  
    //  
  
    MPI_Test(&req, &flag, &status);  
} while (!flag)
```

Пример MPI_Probe

Процедура MPI_Probe использована для определения структуры входящего сообщения.

Процесс 0 ждет сообщения от любого из процессов 1 и 2 с одинаковым тегом.

Посылаемые данные имеют разный тип.

Чтобы определить в какую переменную помещать входящее сообщение, процесс определяет от кого оно поступило.

Следующий после MPI_Probe вызов MPI_Recv гарантировано примет нужное сообщение, далее принимается сообщение от другого процесса.

```
int main(int argc, char **argv)
{ int rank, size, ibuf; float rbuf;
  MPI_Status status;  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Пример MPI_Probe (продолжение)

Выбираем
порядок приема
в зависимости
от результата
Probe

```
    ibuf = rank;  
    rbuf = 1.0 * rank;  
    if(rank == 1) MPI_Send(&ibuf, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);  
    if(rank == 2) MPI_Send(&rbuf, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);  
    if(rank == 0){  
        MPI_Probe(MPI_ANY_SOURCE, 5, MPI_COMM_WORLD, &status);  
        if (status.MPI_SOURCE == 1) {  
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);  
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);  
        }  
        else if (status.MPI_SOURCE == 2) {  
            MPI_Recv(&rbuf, 1, MPI_FLOAT, 2, 5, MPI_COMM_WORLD, &status);  
            MPI_Recv(&ibuf, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);  
        } cout.setf(ios::fixed);  
        cout << " Process 0 recv " << ibuf << " from process 1, " << rbuf <<  
        " from process 2" << endl;  
    }  
    MPI_Finalize();  
}
```

109_04.cpp

```
Process 0 recv 1 from process 1, 2.000000 from process 2
```

Завершение асинхронного обмена

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request – идентификатор асинхронного приема или передачи;

OUT *status* – параметры сообщения.

Ожидание завершения асинхронных процедур *MPI_Isend* или *MPI_Irecv*, ассоциированных с идентификатором *request*. В случае приема, атрибуты и длину полученного сообщения можно определить с помощью параметра *status*.

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

requests – массив идентификаторов асинхронного приема или передачи;

OUT *statuses* – параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива *statuses* будет установлено в соответствующее значение.

Неблокирующий кольцевой обмен

```
#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);
```

```
    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();
}
```

Процессы обмениваются сообщениями с соседями в соответствии с топологией кольца при помощи неблокирующих операций.

Если операции блокирующие – deadlock.

109_05.cpp

```
process 2 prev = 1 next=3
process 0 prev = 3 next=1
process 1 prev = 0 next=2
process 3 prev = 2 next=0
```

Пример из книги: Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.-(Серия "Суперкомпьютерное образование"). **14**

Завершение асинхронного обмена

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

OUT *index* – номер завершенной операции обмена;

OUT *status* – параметры сообщений.

Выполнение процесса блокируется, пока какая-либо одна операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если операций несколько, то случайно выбирается одна из них. Параметр *index* содержит номер элемента в массиве *requests*, содержащего идентификатор завершенной операции.

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

incount – число идентификаторов;

OUT *outcount* – число идентификаторов завершившихся операций обмена;

OUT *indexes* – массив номеров завершившихся операции обмена;

OUT *statuses* – параметры завершившихся сообщений.

Выполнение процесса блокируется до тех пор, пока, по крайней мере, хотя бы одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. 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.

Совмещенные прием/передача сообщений

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

sbuf – адрес начала буфера послыки сообщения;

scount – число передаваемых элементов в сообщении; *stype* – тип передаваемых элементов;

dest – номер процесса-получателя;

stag – идентификатор посылаемого сообщения;

OUT *rbuf* – адрес начала буфера приема сообщения; *rcount* – число принимаемых элементов сообщения; *rtype* – тип принимаемых элементов;

source – номер процесса-отправителя;

rtag – идентификатор принимаемого сообщения;

comm – идентификатор группы;

OUT *status* – параметры принятого сообщения.

Буферы приема и послыки обязательно должны быть различными.

MPI_Sendrecv

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией *MPI_Sendrecv*, может быть принято *MPI_Recv*, и также операция *MPI_Sendrecv* может принять сообщение, отправленное обычной операцией *MPI_Send*.

MPI_Sendrecv содержит 12 параметров: первые 5 параметров такие же, как у *MPI_Send*, остальные 7 параметров такие же как у *MPI_Recv*.

Один вызов – те же действия, что и блок IF-ELSE с четырьмя вызовами.

Следует учесть:

- и прием, и передача используют один и тот же коммутатор;

- порядок приема и передачи данных *MPI_Sendrecv* выбирает автоматически;

- гарантируется, что автоматический выбор не приведет к deadlock;

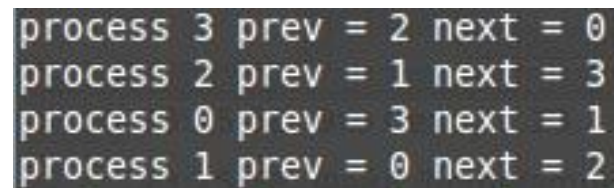
- MPI_Sendrecv* совместима с *MPI_Send* и *MPI_Recv*, т.е. может "общаться" с ними.

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

MPI_Sendrecv – отправить и принять

Операции двунаправленного обмена с соседними процессами в кольцевой топологии

```
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);
    cout << "process " << rank << " prev = " << buf[0] << " next = " << buf[1] << endl;
    MPI_Finalize();
}
```



```
process 3 prev = 2 next = 0
process 2 prev = 1 next = 3
process 0 prev = 3 next = 1
process 1 prev = 0 next = 2
```

109_06.cpp

Пример из книги: Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.-(Серия "Суперкомпьютерное образование"). **18**

MPI-3.0 Порождение процессов «на лету»

int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[]);

command - имя создаваемой программы (строка, значимая только в корневом каталоге)

argv - аргументы команды (массив строк, значимый только в корне)

maxprocs - максимальное количество запущенных процессов (целое число, значимое только в корневом каталоге)

info - набор пар ключ-значение, сообщаящий системе времени выполнения, где и как запускать процессы (обрабатывать, значимые только в корневом каталоге)

root - ранг процесса, в котором рассматриваются предыдущие аргументы (integer)

comm - intracommunicator, содержащий группу нерестовых процессов (дескриптор)

OUT intercomm - intercommunicator между исходной группой и вновь созданной группой (дескриптором)

OUT array_of_errcodes - один код для каждого процесса (массив целых чисел)

Новые процессы, которые должны вызывать MPI_Init, имеют свой собственный MPI_COMM_WORLD, состоящий из всех процессов, созданных вызовом MPI_Comm_spawn. Функция MPI_Comm_get_parent, вызываемая дочерними элементами, возвращает интеркоммуникатор (intercommunicator) с дочерними элементами в локальной группе и родителями в удаленной группе.

Коллективная функция MPI_Intercomm_merge может быть вызвана родителями и детьми для создания нормального (интра)коммуникатора, содержащего все процессы, как старые, так и новые, но для многих типовых случаев коммуникации это необязательно.

MPI_Comm_spawn

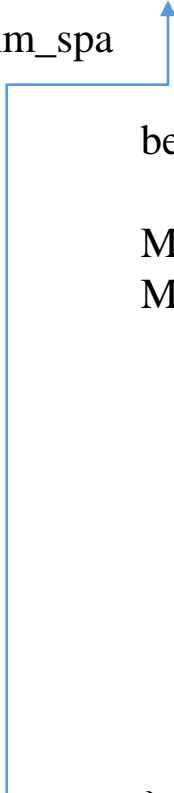
```
/* Пример с веб-страницы
http://mpi.deino.net/mpi\_functions/MPI\_Comm\_spawn.html */
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define NUM_SPAWNS 2

int main( int argc, char *argv[] )
{
    int np = NUM_SPAWNS;
    int errcodes[NUM_SPAWNS];
    MPI_Comm parentcomm, intercomm;

    MPI_Init( &argc, &argv );
    MPI_Comm_get_parent( &parentcomm );

    if (parentcomm == MPI_COMM_NULL)
    {
        /* Create 2 more processes - this example must
        be called spawn_example.exe for this to work. */
        MPI_Comm_spawn( "a.out",
            MPI_ARGV_NULL, np, MPI_INFO_NULL, 0,
            MPI_COMM_WORLD, &intercomm, errcodes );
        printf("I'm the parent.\n");
    }
    else
    {
        printf("I'm the spawned.\n");
    }
    fflush(stdout);
    MPI_Finalize();
    return 0;
}
```



109_07.cpp

Порождение процессов с аргументами

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define NUM_SPAWNS 2

int main( int argc, char *argv[] )
{
    int np = NUM_SPAWNS;
    int errcodes[NUM_SPAWNS];
    MPI_Init( &argc, &argv );
    MPI_Comm parentcomm, intercomm;

    MPI_Comm_get_parent( &parentcomm );
    if (parentcomm == MPI_COMM_NULL)
    {
        char command[] = "a.out";
        char* my_argv[] = {"-a", "abc",
NULL};

        /* Create 2 more processes - this
example must be called spawn_example.exe
for this to work. */
        MPI_Comm_spawn( command,
my_argv, np, MPI_INFO_NULL, 0,
MPI_COMM_WORLD, &intercomm,
errcodes);
    }
    else
    {
        int index;
        for(index = 0; index < argc; index++)
            printf("The %d is
%s\n",index,argv[index]);
        fflush(stdout);
        MPI_Finalize();
        return 0;
    }
}
```

109_08.cpp

Не рекомендуется часто порождать дочерние процессы, так как это снижает производительность

Алгоритм запуска mpi-программы на нескольких узлах

1. Настроить доступ по паре публичный-приватный ключ для каждого компьютера
`ssh-keygen -t rsa`
`ssh-copy-id ip-other-computer....`

2. Создать одинаковые папки (и пользователей) на всех компьютерах, где разместить исполняемые файлы и все другие требуемые файлы программы или настроить NFS, как показано здесь <http://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/>

3. Компилировать программу на каждом узле (или скопировать, если оборудование и набор библиотек в системе полностью идентичны)
`mpic++ 109_03.cpp && ./a.out`
`0 prints 0`

`scp ./a.out ip-other-computer:/home/user/`

4. Создать файл с именами узлов. Пример для 4 процессов
содержимое file.txt:

ip_компьютера_1
ip_компьютера_1
ip_компьютера_2
ip_компьютера_2

`mpirun -n 4 --hostfile file.txt ./a.out`
`1 prints 1`
`0 prints 0`
`2 prints 2`
`3 prints 3`

Второй вариант:

`mpirun -n 4 -hosts ip_компьютера_1,ip_компьютера_1,ip_компьютера_2,ip_компьютера_2 ./a.out`

Можно использовать имена узлов вместо ip, если имена прописаны в /etc/hosts на каждом компьютере.