Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 118, 119	
12:50 – 14:20	Лекция	
	1 неделя	2 неделя
14:35 - 17:55	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью

- C++11 threads (pthreads)
- OpenMP

Системы с распределенной памятью

- MPI = message passing interface

Гетерогенные системы (графические процессоры)

- CUDA

Лекция 12. MPI. Прием/передача сообщений

Стандарты MPI http://www.mpi-forum.org

Обмен данными с использованием MPI.

- http://habrahabr.ru/company/intel/blog/251357/

Лекции и семинары

- http://www.slideshare.net/Aleximos/mpi-9793227

Технологии

- http://parallel.ru/tech/tech_dev/mpi.html
- http://parallel.ru/tech/tech_dev/MPI/examples/ (примеры)

Примеры из учебника "Технологии параллельного программирования MPI и OpenMP"

- http://parallel.ru/tech/tech_dev/MPI%26OpenMP/examples/

Составляющие сообщения

1. Блок данных сообщения – void *
2. Информация о данных сообщения.
   (a) Тип данных – MPI_Datatype;
   (b) Количество данных.
3. Информация о получателе и отправителе сообщения.
   (a) Коммуникатор – идентификатор группы процессов типа MPI_Comm, коммуникатор верхнего уровня – MPI_COMM_WORLD;
   (b) Ранг получателя – номер процесса получателя в указанном коммуникаторе;
   (c) Ранг отправителя – номер процесса отправителя в указанном коммуникаторе
   Можно принимать сообщения от всех отправителей в данном коммуникаторе MPI_ANY_SOURCE.
4. Тег сообщения.
   Произвольное число типа int. Можно принимать сообщения с определенным тегом, можно с любым – MPI_ANY_TAG.

Соответствие типов данных MPI и C

Тип MPI	Тип C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char
MPI_PACKED	

Богачёв К. Ю.

Основы параллельного программирования / К. Ю. Богачёв. — М.: БИНОМ. Лаборатория знаний, 2003. — 342 с., илл.

Общие процедуры и коммуникаторы

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

получить количество процессов в коммуникаторе

коммуникатор

указатель на результат

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

получить количество всех процессов

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

получить ранг (номер) процесса в группе (коммуникаторе)

```
int my_rank;
Int MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Блокирующая посылка сообщения

Операции точка-точка с синхронизацией (один – отправитель, другой – получатель)

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

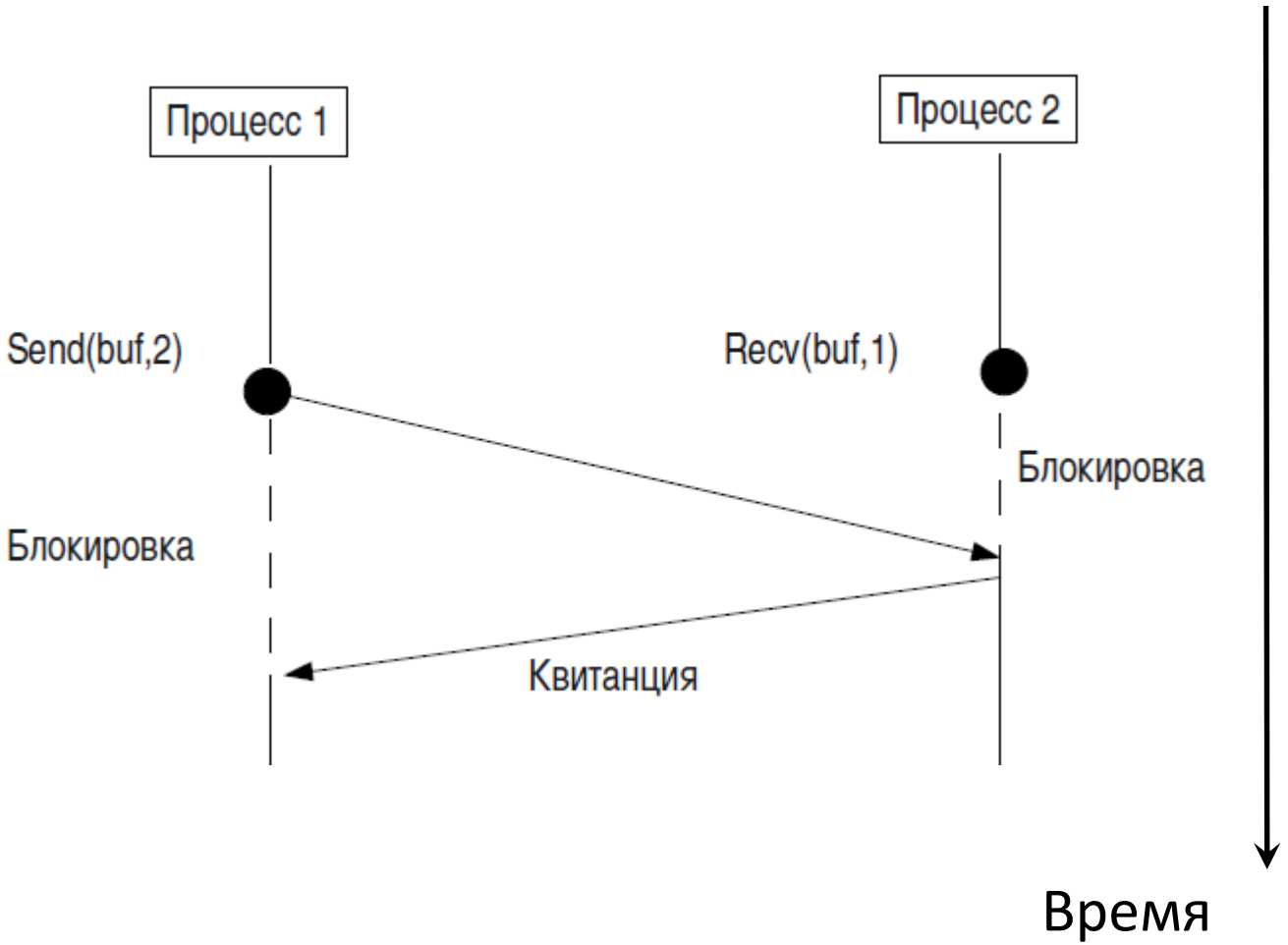
buf – адрес начала буфера посылки сообщения;
count – число передаваемых элементов в сообщении;
datatype – тип передаваемых элементов;
dest – номер процесса-получателя;
msgtag – идентификатор сообщения;
comm – идентификатор группы.

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены подряд в буфере *buf*. Значение *count* может быть нулем. Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант типа **MPI_Datatype**. Разрешается передавать сообщение самому себе, что может привести к тупиковым ситуациям.

Блокирующая посылка сообщения

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы посредством: копирования в промежуточный буфер или непосредственной передачи процессу *dest*.

Возврат из подпрограммы *MPI_Send* не означает ни того, что сообщение уже передано процессу *dest*, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший *MPI_Send*.



Модификации MPI_Send

В MPI_Send нет гарантии, что сообщение получено процессом dest.

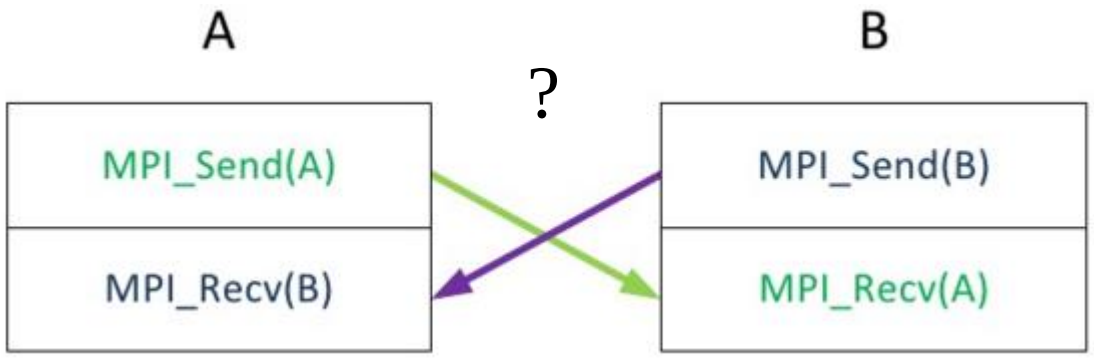
MPI_Bsend – передача сообщения *с буферизацией* (сообщение записывается в буфер, не зависит от приема сообщения, код ошибки, если места в буфере недостаточно).

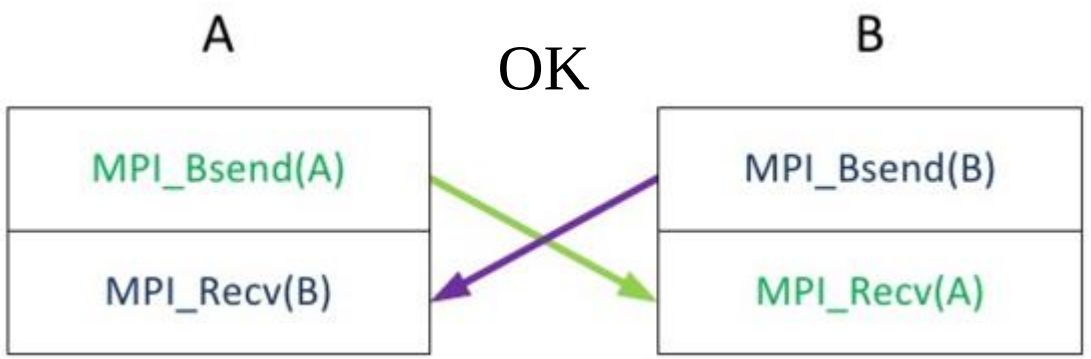
MPI_Ssend – передача сообщения *с синхронизацией* (выход из процедуры, если прием посылаемого сообщения инициализирован получателем; может замедлить выполнение кода).

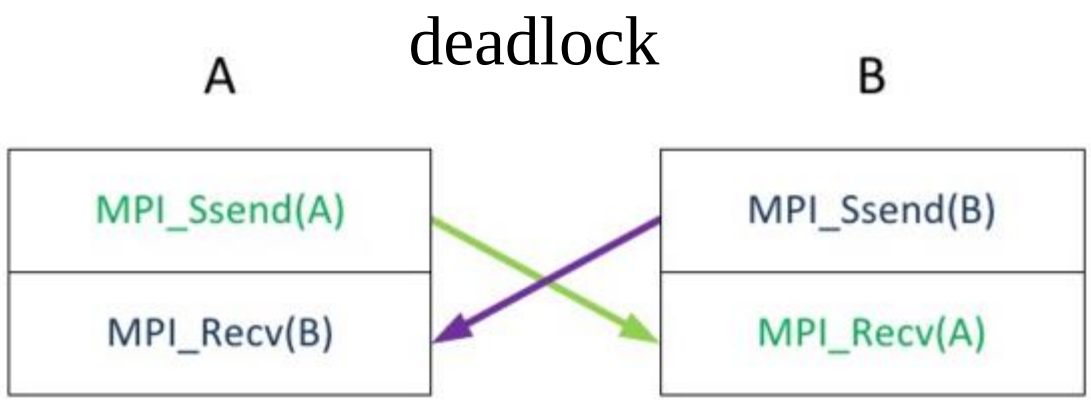
MPI_Rsend – передача сообщения *по готовности* (используется если получатель уже инициализировал прием, например, использовав MPI_Barrier, может сократить протокол взаимодействия между отправителем и получателем).

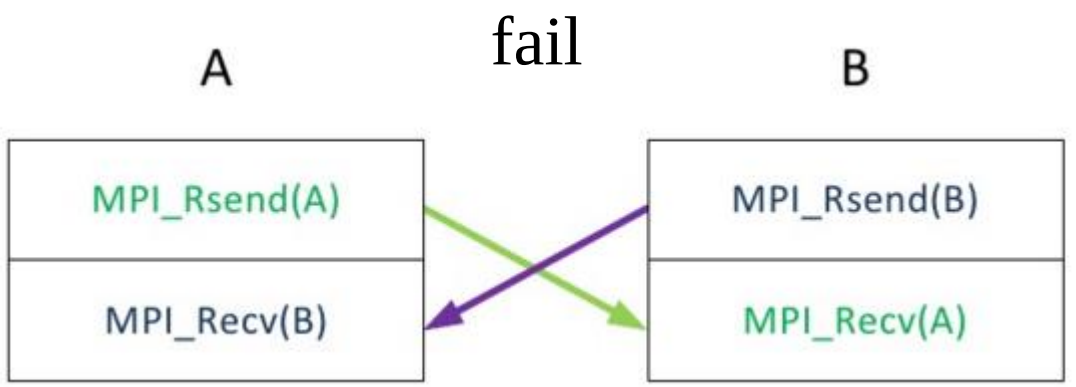
Обычно в MPI выделяется некоторый объём памяти для буферизации сообщений, рекомендуется явно выделять в программе *достаточный* буфер для всех пересылок с буферизацией.

Deadlock


A
B
?
MPI_Send(A)
MPI_Recv(B)
MPI_Send(B)
MPI_Recv(A)


A
B
OK
MPI_Bsend(A)
MPI_Recv(B)
MPI_Bsend(B)
MPI_Recv(A)


deadlock
A
B
MPI_Ssend(A)
MPI_Recv(B)
MPI_Ssend(B)
MPI_Recv(A)


fail
A
B
MPI_Rsend(A)
MPI_Recv(B)
MPI_Rsend(B)
MPI_Recv(A)

Типичные взаимные блокировки

Процесс 1 блокирует ресурс А.
Процесс 2 блокирует ресурс Б.
Процесс 1 пытается получить доступ к ресурсу Б.
Процесс 2 пытается получить доступ к ресурсу А.

В итоге один из процессов должен быть прерван, чтобы другой мог продолжить выполнение.

Прием сообщения

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

OUT *buf* – адрес начала буфера приема сообщения;

count – максимальное число элементов в принимаемом сообщении;

datatype – тип элементов принимаемого сообщения;

source – номер процесса-отправителя;

msgtag – идентификатор принимаемого сообщения;

OUT *status* – параметры принятого сообщения.

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Чтобы узнать точное число элементов в сообщении – подпрограмма *MPI_Probe*.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере *buf*.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову *MPI_Recv*, то первым будет принято то сообщение, которое было отправлено раньше.

Обмен сообщениями для двух процессов

Нулевой процесс посылает сообщение процессу с номером 1 и ждет от него ответа.

Программа запущена на 3-х процессах, выполняют пересылки 0 и 1.

```
process 2 a = 0 b = 0
process 0 a = 2 b = 1
process 1 a = 2 b = 1
```

```
int main(int argc, char **argv) {
  int rank;  float a, b;        MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  a = 0.0;              b = 0.0;
  if ( rank  ==  0 ) {
    b = 1.0;
    MPI_Send(&b, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
  }
  if ( rank  ==  1 ) {
    a = 2.0;
    MPI_Recv(&b, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD, &status);
    MPI_Send(&a, 1, MPI_FLOAT, 0, 5, MPI_COMM_WORLD);
  }
  cout << " process " << rank <<  " a = " << a << " b = " << b << endl;
  MPI_Finalize();
}
```

Пример MPI_Send/MPI_Recv

Каждый процесс с четным номером посылает сообщение соседу с номером на 1 большим.

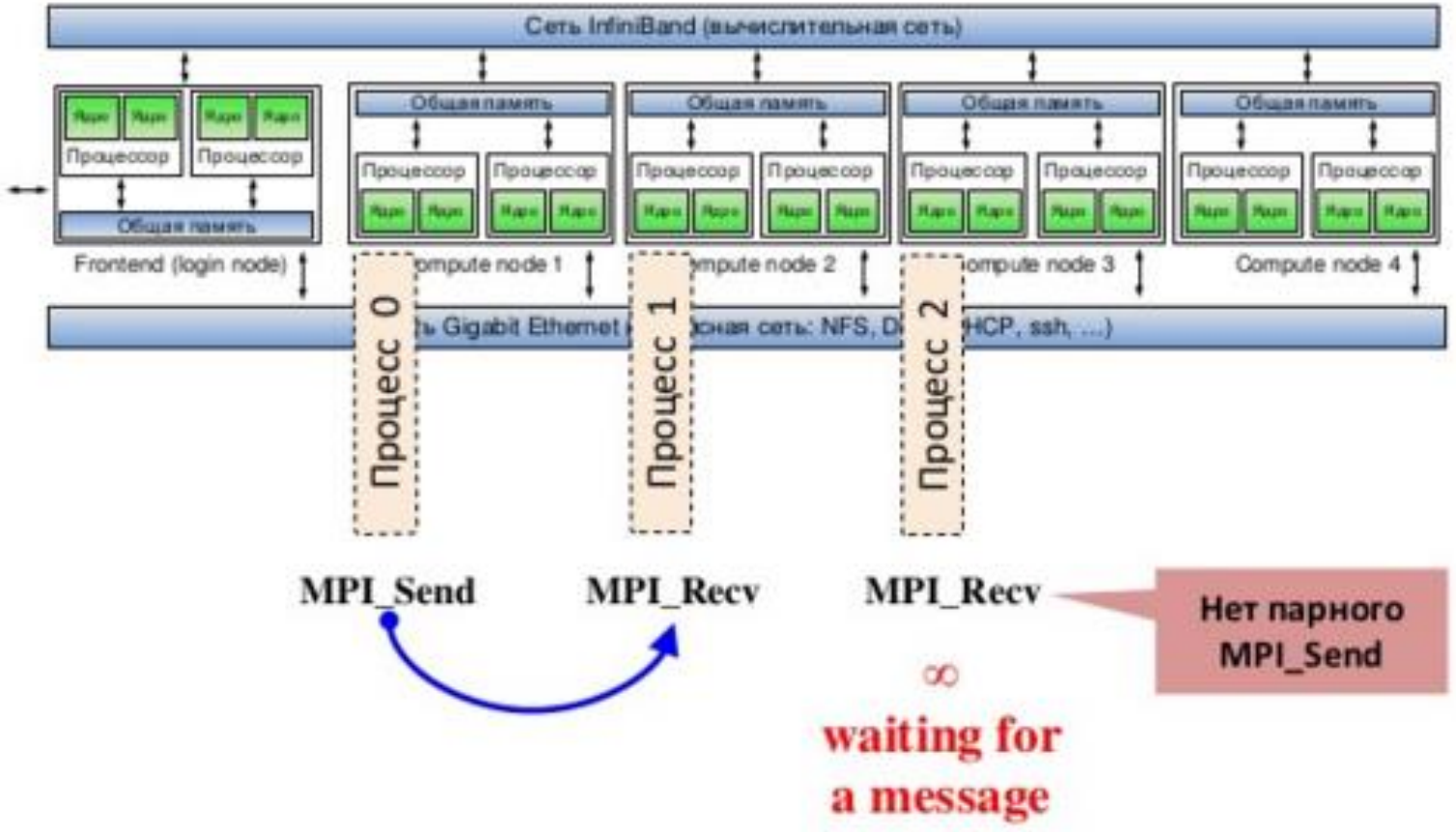
Поставлена проверка для процесса с максимальным номером он не посылает сообщение несуществующему процессу.

Значение *b* изменяется только на процессах с нечетными номерами.

```
process 1 a = 1 b = 0
process 0 a = 0 b = -1
process 2 a = 2 b = -1
```

```
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;
  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);
  cout << " process " << rank <<  " a = " << a << " b = " << b << endl;
  MPI_Finalize();
}
```

Бесконечное ожидание




```
#include "mpi.h"
#include <stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
   int rank; int size, a;
   MPI_Status status; a = 3;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if(rank == 0)
      MPI_Send(&a, 1, MPI_DOUBLE, 1, 1, MPI_COMM_WORLD);
   else //if (rank == 1)
      MPI_Recv(&a, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status);
   cout << rank << endl;
   MPI_Finalize();
}
```

if...else – работает только для 2-х процессов.

Выборочный прием сообщения

При приеме сообщений вместо аргументов SOURSE и MSGTAG можно использовать предопределенные константы:

- MPI_ANY_SOURSE – признак, что подходит сообщение от любого процесса;
- MPI_ANY_TAG – признак , что подходит сообщение с любым идентификатором.

При одновременном использовании будет принято сообщение с любым идентификатором от любого процесса.

О статусе сообщения

Атрибуты принятого сообщения можно определить по элементам массива status.

Параметр status – структура предопределенного типа MPI_Status с полями MPI_SOURSE (реальный ранг сообщения), MPI_TAG (реальный тег) и MPI_ERROR (код ошибки).

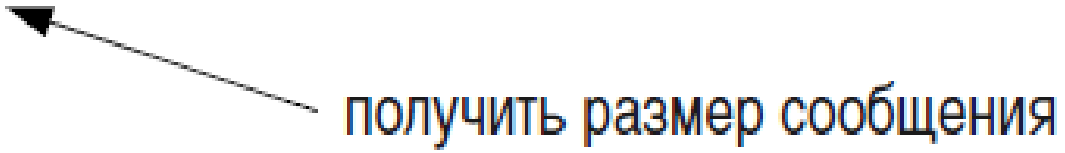
Номер принимающего процесса требуется указать явно.

Если один процесс посылает два сообщения, соответствующие одному MPI_Recv, другому процессу, то первым принимается первое сообщение.

Если сообщение отправлено разными процессами, то порядок получения не определен.

Размер сообщения

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



- status – информация о полученном сообщении;
- datatype – тип данных, в единицах которого нужно получить размер сообщения;

Выходной параметр:

count – количество полученных элементов в единицах datatype или константа MPI_UNDEFINED, если длина данных сообщения не делится нацело на размер типа datatype.

При коллективных обменах получатель информацию о сообщении не получает.

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.

Пример вычисление интеграла 1

a и b – границы интервала;
INTEGRAL – результат;
p – число процессов;
proc_integral – функция, работающая в процессе с номером myrank;
a_p, b_p, n_p, h_p – параметры подинтервалов для текущего процесса;

```
using namespace std;
#define N 10000000
#define NTIMES 100
static double a = 0.;          static double b = 1.;
static double INTEGRAL = 0.;
void proc_integral(double (*fun)(double), const int myrank, int p)
{
        double h_p = (b - a) / p;          int n_p = N / p; int i;
        double a_p = a + myrank * h_p;     double b_p = a_p + h_p;
        double integ = 0.;                 double x_p = a_p;
        while (x_p < b_p) {
                integ += fun(x_p) * h_p / n_p;
                x_p += h_p / n_p;
        }
        MPI_Reduce(&integ, &INTEGRAL, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
```

integ – значение интеграла в текущем процессе;
MPI_Reduce – сложить все ответы и передать процессу 0.

Функция MPI_Reduce

Коллективный обмен сообщениями можно считать избыточным (Богачев К.Ю.),
любая программа м.б. написана без их использования.

Коллективный обмен м.б. заменен на цикл парных обменов, но при этом скорость ниже и
не используется специфика вычислительной установки.

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 получатель в коммуникатора 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.

Результат – в процессе с номером root в группе comm.

Операции над данными в MPI

Операция MPI	Значение
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое «и»
MPI_BAND	побитовое «и»
MPI_LOR	логическое «или»
MPI_BOR	побитовое «или»
MPI_LXOR	логическое «исключающее или»
MPI_BXOR	побитовое «исключающее или»
MPI_MAXLOC	максимум и его позиция
MPI_MINLOC	минимум и его позиция

Пример вычисление интеграла 2

```
double f(double x) { return x*x; }
int main(int argc, char **argv)
{
  double time_start, time_finish;      int myrank; int p;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
          time_start = MPI_Wtime();
  proc_integral(f, myrank, p);
          time_finish = MPI_Wtime();
  cout  << " Process " << myrank << " time = " << (time_finish-time_start)/NTIMES << endl;
  if (myrank == 0)
          cout << " Integral " << INTEGRAL << endl;
  MPI_Finalize();
  return 0;
}
```

proc_integral(f, myrank, p); – вычисление интеграла в каждом из процессов;

– печать ответа в процессе 0.

```
Process 1 time = 0.000887368
Process 3 time = 0.00129306
Process 2 time = 0.00146127
Process 0 time = 0.0015192
Integral 0.333333
```

Вычисление суммы способ 2

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

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

buf – адрес начала буфера посылки сообщения;

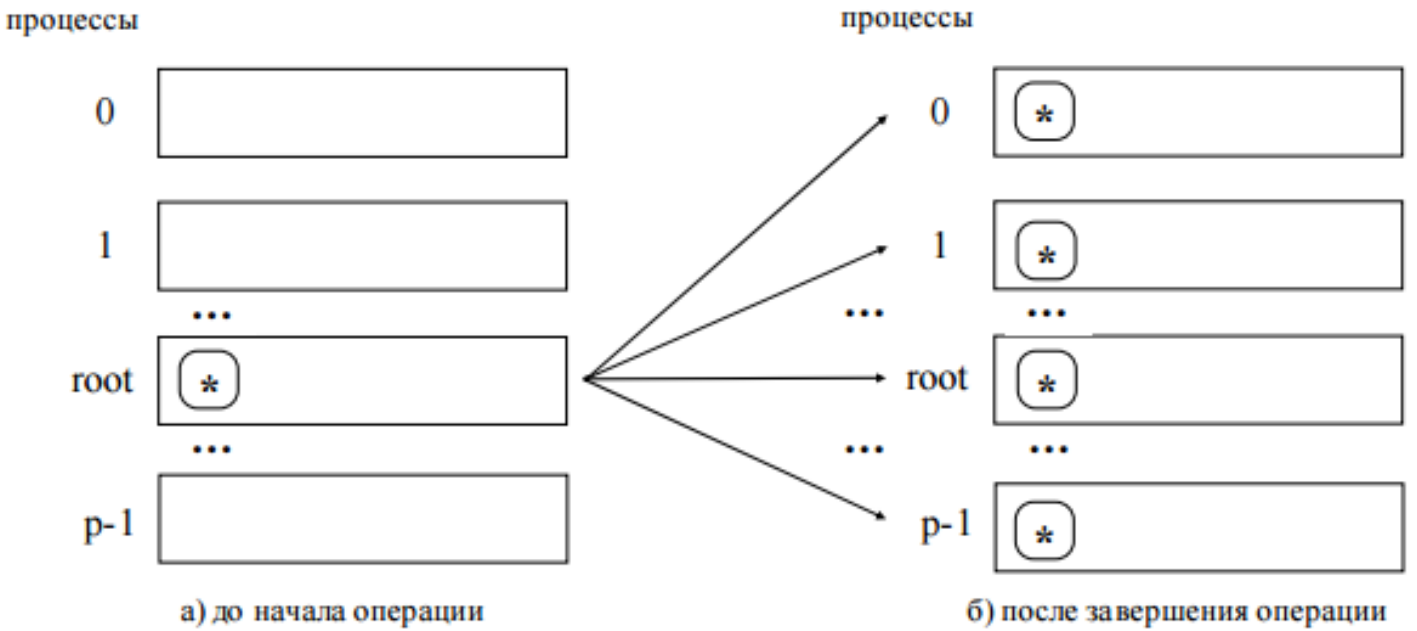
count – число передаваемых элементов в сообщении;

source – номер рассылающего процесса.

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.



Пример вычисление суммы

```
#define N 100
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);
        MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        int k = N / p; for ( int i = 0; i < N; i++ ) x [i] = 1;
        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]; cout << " Summa 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;
}
```

Операции ввода-вывода в MPI

Ввод-вывод на экран зависит от реализации:

- всем работающим процессам разрешено осуществлять ввод-вывод – перемешанный вывод от всех процессов;
- ввод-вывод разрешен только процессу с номером 0 в группе MPI_COM_WORLD, все процессы посылают запрос на свой ввод-вывод в качестве сообщений;
- всем работающим процессам запрещено осуществлять ввод-вывод на экран, используется на распределенных вычислительных установках, ввод-вывод осуществляется через файлы.

Для переносимой программы ввод-вывод должен осуществляться через файл.

При одновременной записи в файл может быть перемешивание и потеря данных:

- файловый ввод-вывод только процессу с номером 0 в группе MPI_COM_WORLD;
- каждый процесс осуществляет ввод-вывод в собственный файл.

Вычисление экспоненты

Задача легко поддается распараллеливанию, так как искомое число является суммой отдельных слагаемых и каждый отдельный процесс может заняться вычислением отдельных слагаемых. Количество слагаемых рассчитываться в каждом отдельно взятом процессе.
Для нахождения факториала используется рекурсивная функция, его раздельное вычисление приводит к значительной потере производительности и скорости вычисления.
Алгоритм выполнения кода

1. В программу передается значение числа n, которое отправляется по всем процессам.
2. При инициализации первого главного процесса, запускается таймер.
3. Каждый процесс выполняет цикл, где значением приращения является количество процессов в системе. В каждой итерации цикла вычисляется слагаемое и сумма таких слагаемых сохраняется в переменную Sum.
4. После завершения цикла каждый процесс суммирует свое значение Sum к переменной Result, используя для этого функцию приведения MPI_Reduce.
5. После завершения расчетов на всех процессорах, первый главный процессор останавливает таймер и отправляет в поток вывода получившееся значение переменной Result.
6. В поток вывода отправляется также и отмеренное таймером значение времени в милисекундах.

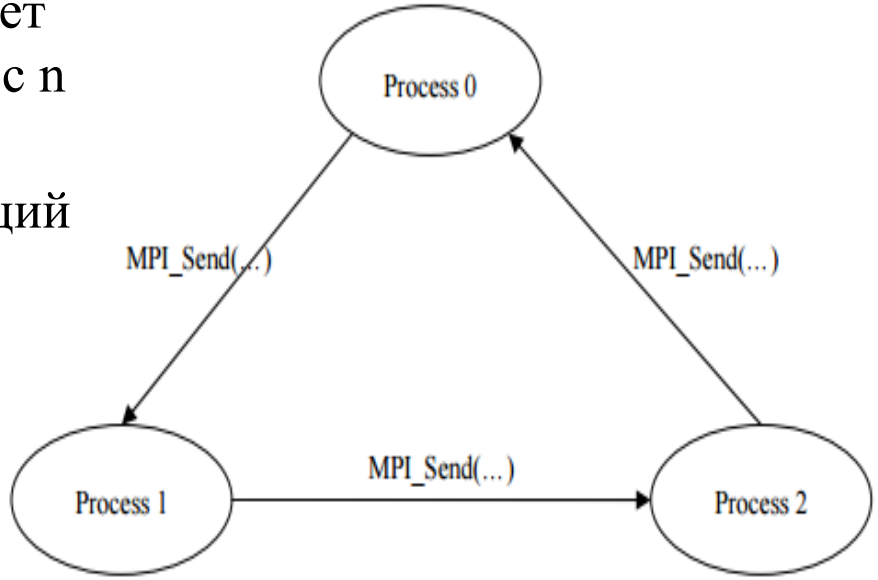
Обзор типичных ошибок MPI программ

- **Взаимная блокировка при пересылке сообщений.**

n процессов передают информацию по цепочке, процесс i передает информацию процессу i+1 (для индексов i = 0,…, n – 2), процесс с n – 1 передает информацию процессу с индексом 0.

Передача осуществляется с использованием блокирующих функций MPI_Send и MPI_Recv, каждый процесс сначала вызывает MPI_Send, а затем MPI_Recv. MPI_Send возвратит управление только если передача завершена. Но передача не может завершиться, пока принимающий процесс не вызовет MPI_Recv.

А MPI_Recv будет вызвана, только после того, как процесс завершит отправку своего сообщения. Цепочка передачи замкнута.



Избежать блокировки можно, например, вызывая на процессах с четными индексами сначала MPI_Send, а затем MPI_Recv, а на процессах с нечетными индексами – в обратной порядке.

- **Освобождение или изменение буферов, используемых при неблокирующей пересылке** (самые сложные для отладки).
- **Несоответствие вызовов функций передачи и приема сообщений.**