Технологии параллельного программирования.

Введение в MPI.

Лектор: доц. Н.Н.Попова

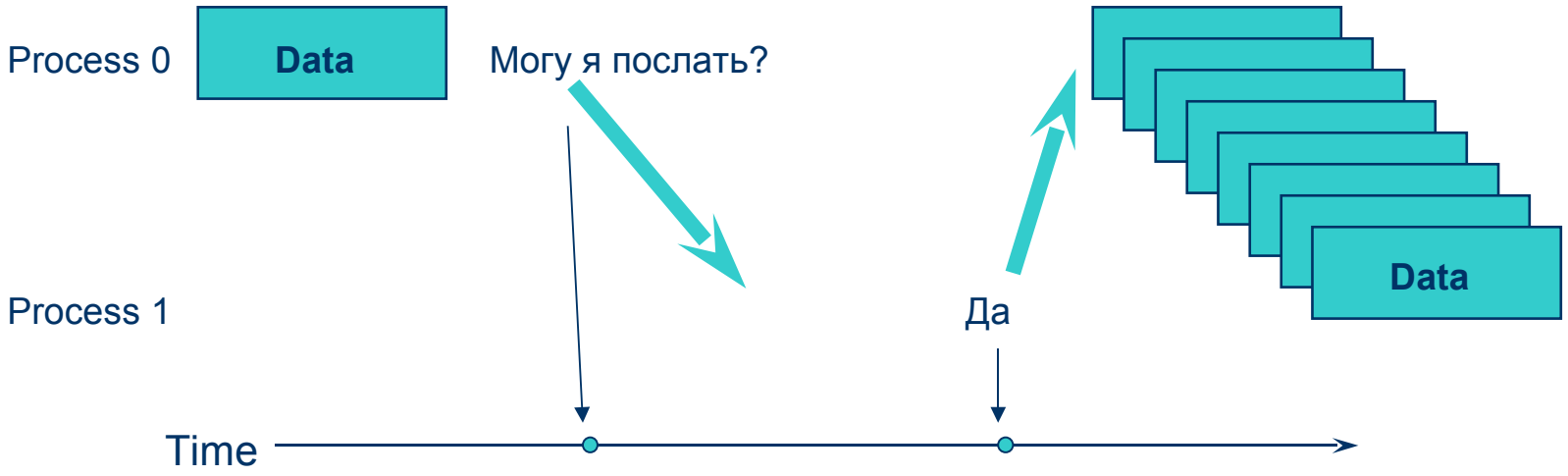
План

- Краткая история MPI
- Структуры данных MPI
- Взаимодействие «точка-точка»
- Коллективные операции

Модели параллельных программ

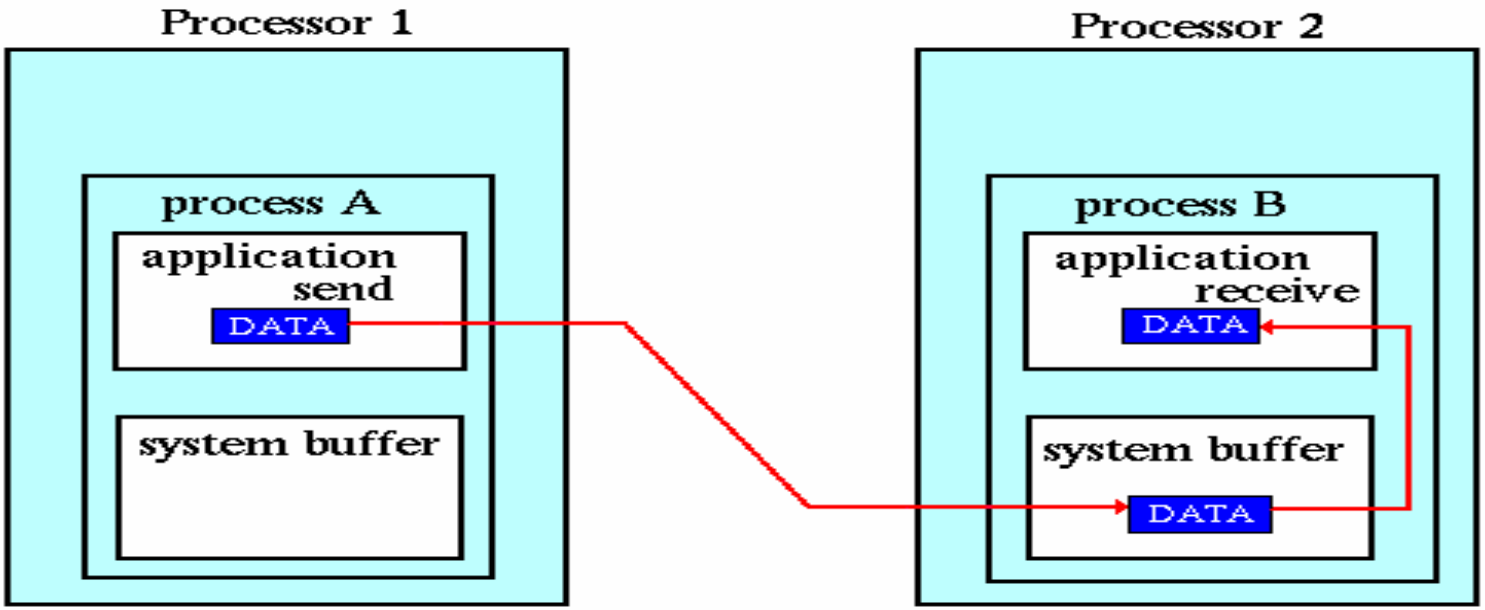
- Системы с распределенной памятью
  - Явное задание коммуникаций между процессами - “**Message Passing”**
  - Message passing библиотеки:
    - MPI (“Message Passing Interface”)
    - PVM (“Parallel Virtual Machine”)
    - Shmem, MPT (Cray)
- Shared memory системы
  - Программирование, основанное на “Thread”
  - Директивы компиляторов (OpenMP, …)
  - Возможность использования передачи сообщений

Message passing = передача данных + синхронизация



- Требуется взаимодействие между отправителем и получателем сообщений

Схема передачи



Path of a message buffered at the receiving process

C: Hello, MPI world!

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    printf("Hello, MPI world\n");
    MPI_Finalize();
    return 0; }
```

C++: Hello, MPI world!

```
#include <iostream.h>
#include "mpi++.h"
int main(int argc, char **argv) {
MPI::Init (argc, argv);
cout << "Hello world" << endl;
MPI::Finalize();
return 0; }
```

Fortran: Hello, MPI world!

```
program main
include 'mpif.h'
integer ierr
call MPI_INIT(ierr)
print *, 'Hello, MPI world'
call MPI_FINALIZE(ierr)
end
```

MPI forum

- Первый стандарт для систем передачи сообщений
- MPI 1.1 Standard разрабатывался 92-94
- MPI 2.0 Standard - 95-97
- Стандарты
  - http://www.mcs.anl.gov/mpi
  - http://www.mpi-forum.org/docs/docs.html

Цель MPI

- Основная цель:
  - Обеспечение переносимости исходных кодов
  - Эффективная реализация
- Кроме того:
  - Большая функциональность
  - Поддержка неоднородных параллельных архитектур

Управляющие конструкции MPI

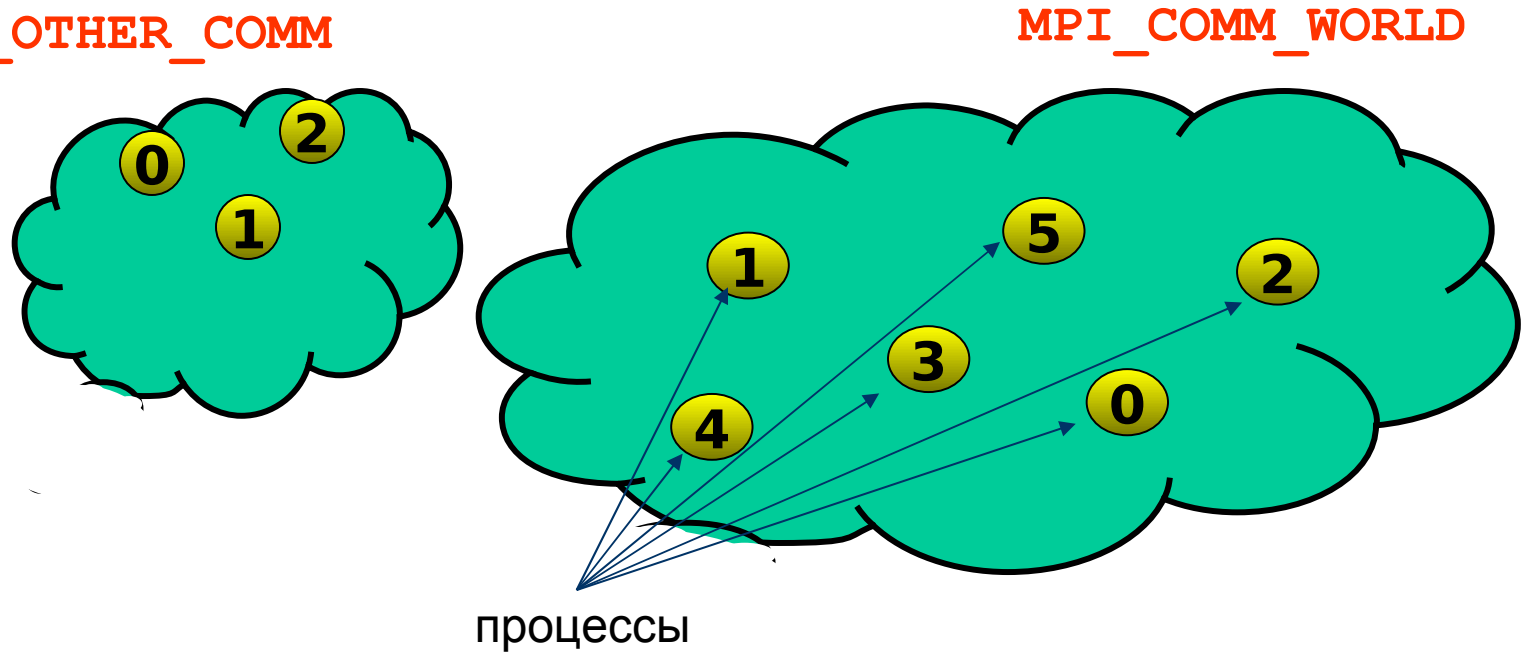
- Внутренние структуры данных MPI (поддерживают неоднородность)
- Возможность обращения к ним в программе
- Реализуются в C посредством `typedef`

Понятие коммуникатора MPI

- Управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом
- **Все** обращения к MPI функциям содержат коммуникатор, как параметр

- Наиболее часто используемый коммуникатор `MPI_COMM_WORLD`
  - Определяется при вызове `MPI_Init`
  - Содержит ВСЕ процессы программы

MPI-коммуникаторы



Заголовочный файл

- MPI константы, макросы, типы, …

C:

```
#include <mpi.h>
```

Формат MPI-функций

C (case sensitive):

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

C++ (case sensitive):

```
error = MPI::Xxxxx(parameter,...);
MPI::Xxxxx(parameter,...);
```

Fortran:

```
call MPI_XXXXX(parameter,...,IERR);
```

Инициализация MPI

- `MPI_Init` должна первым вызовом, вызывается только один раз

C:

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

C++:

```
void MPI::Init (int& argc, char**& argv)
```

Fortran:

```
INTEGER IERR
MPI_INIT (IERR)
```

Количество процессов в коммуникаторе

- Размер коммуникатора

C:

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

Номер процесса (process rank)

- Process ID в коммуникаторе
  - Начинается с 0 до (*n*-1), где *n* – число процессов
- Используется для определения номера процесса-отправителя и получателя

C:

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

Завершение MPI-процессов

- Никаких вызовов MPI функций после

C:

MPI_Finalize()

*MPI_Abort (*MPI_Comm_size(MPI_Comm c*omm, int*errorcode)*

Если какой-либо из процессов не выполняет MPI_Finalize, программа зависает.

C: Hello, MPI world!

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
   int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello, MPI world! I am %d of %d\n“,rank,size);
    MPI_Finalize();
    return 0; }
```

C++: Hello, MPI world!

```
#include <iostream.h>
#include "mpi++.h"
int main(int argc, char **argv) {
    MPI::Init (argc, argv);
    int rank = MPI::COMM_WORLD.Get_rank();
    int size = MPI::COMM_WORLD.Get_size();
    cout << "Hello world! I am " << rank << " of "
    << size << endl;
    MPI::Finalize();
    return 0;}
```

Fortran: Hello, MPI world!

```
program main
include ’mpif.h’
integer rank, size, ierr
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
print *, ’Hello world! I am ’, rank, ’ of ’, size
call MPI_FINALIZE(ierr)
end
```

Bones.c

```
#include <mpi.h>
int main(int argc, char *argv[]) {
   int rank, size;
/* … Non-parallel part of code   */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

/* … your code here … */

   MPI_Finalize ();
}
```

Сообщения

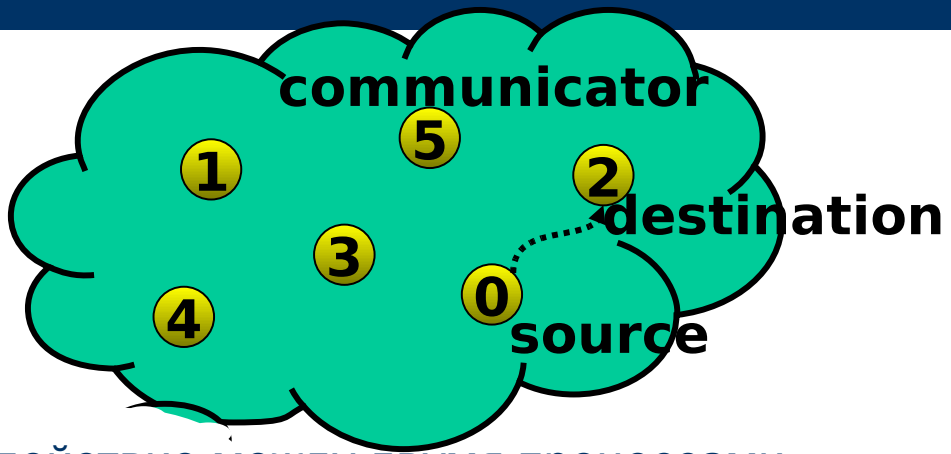
- Сообщение – массив элементов некоторого, заданного типа данных MPI
- Типы данных MPI:
  - Базовые типы
  - производные типы

  Производные типы строятся с использованием базовых

Базовые типы MPI - C

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

Передача сообщений типа «точка-точка»



- Взаимодействие между двумя процессами
- Процесс-отправитель(Source process) ***посылает*** сообщение процессу-получателю (Destination process)
- Процесс-получатель ***принимает*** сообщение
- Передача сообщения происходит в рамках заданного коммуникатора
- Процесс-получатель определяется рангом в коммуникаторе

Завершение

- “Завершение” передачи означает, что буфер в памяти, занятый для передачи, может быть безопасно использован для доступа, т.е.
  - Send: переменная, задействованная в передаче сообщения, может быть доступна для дальнейшей работы
  - Receive: переменная, получающая значение в результате передачи, может быть использована

Режимы (моды) операций передачи сообщений

- Режимы MPI-коммуникаций определяют, при каких условиях операции передачи завершаются
- Режимы могут быть блокирующими (blocking) или неблокирующими (non-blocking)
  - Blocking: возврат из функций передачи сообщений только по завершению коммуникаций
  - Non-blocking: немедленный возврат из функций, пользователь должен контролировать завершение передач

Режимы (modes) передачи

Режим (Mode)	Условие завершения (Completion Condition)
Synchronous send	Завершается только при условии инициации приема
Buffered send	Всегда завершается (за исключением ошибочных передач), независимо от приема
Standard send	Сообщение отослано (состояние приема неизвестно)
Ready send	Всегда завершается (за исключением ошибочных передач), независимо от приема
Receive	Завершается по приему сообщения

Фунции передачи сообщений «точка-точка» (блокирующие)

Режим (MODE)	MPI функции
Standard send	MPI_Send
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Ready send	MPI_Rsend
Receive	MPI_Recv

Отсылка сообщения

C:

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

Параметры MPI_Send

buf	адрес буфера
count	число пересылаемых элементов
datatype	MPI datatype
dest	rank процесса-получателя
tag	определяемый пользователем параметр, который может быть использован для идентификации сообщений
comm	MPI-коммуникатор

Пример:

```
MPI_Send(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD)
```

Синхронный send (`MPI_Ssend`)

- Критерий завершения: принимающий процесс посылает подтверждение (“handshake”), которое должно быть получено отправителем прежде, чем send может считаться завершенным
- Используется в случаях, когда надо точно знать, что сообщение получено
- Посылающий и принимающий процессы синхронизируются
  - Независимо от того, кто работает быстрее
  - Idle time (простой) процесса возможен
- Самый безопасный режим работы

Buffered send (`MPI_Bsend`)

- Критерий завершения: завершение передачи, когда сообщение скопируется в буфер
- Преимущество: гарантировано немедленное завершение передачи (предсказуемость)
- Недостатки: надо явно выделять буфер под сообщения
- Функции MPI для контроля буферного пространства

```
MPI_Buffer_attach
MPI_Buffer_detach
```

Standard send (`MPI_Send`)

- Критерий завершения: Не предопределен
- Завершается, когда сообщение отослано

- Можно предполагать, что сообщение достигло адресата

- Зависит от реализации

Ready send (`MPI_Rsend`)

- Критерий завершения: завершается немедленно, но успешно только в том случае, если процесс-получатель выставил receive
- Преимущество: немедленное завершение
- Недостатки: необходимость синхронизации
- Потенциально хорошая производительность

Прием сообщения

C:

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

Условия успешного взаимодействия «точка-точка»

- Отправитель должен указать правильный rank получателя
- Получатель должен указать верный rank отправителя
- Одинаковый коммуникатор
- Тэги должны соответствовать друг другу
- Буфер у процесса-получателя должен быть достаточного объема

Wildcarding (джокеры)

- Получатель может использовать джокер

Для получения от ЛЮБОГО процесса

`MPI_ANY_SOURCE`

- Для получения сообщения с ЛЮБЫМ тэгом

`MPI_ANY_TAG`

- Реальные номер процесса-отправителя и тэг возвращаются через параметр ***`status`***

Информация о завершившемся приеме сообщения

- Возвращается функцией **MPI_Recv** через параметр status

- Содержит:
  - Source: ***status.MPI_SOURCE***
  - Tag: ***status.MPI_TAG***
  - Count: ***MPI_Get_count***

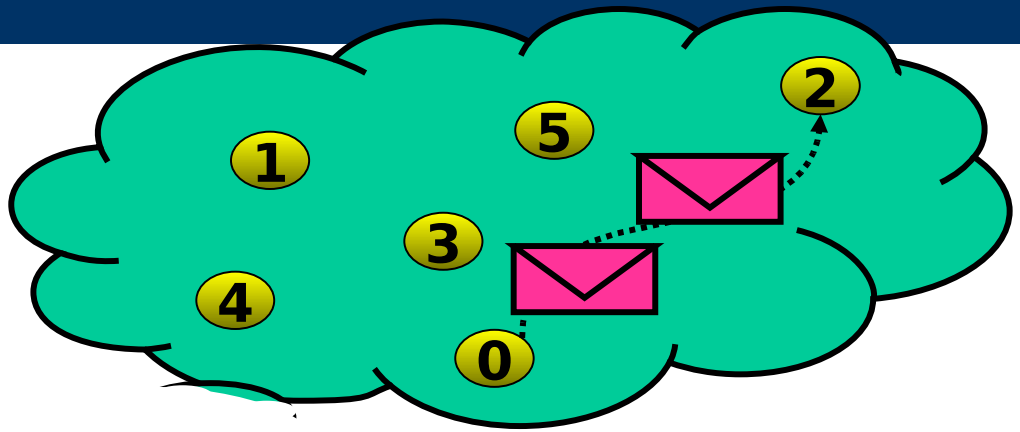
Полученное сообщение

- Может быть меньшего размера, чем указано в функции MPI_Recv
- `count` – число реально полученных элементов

C:

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

Порядок приема сообщений



- Сообщения принимаются в том порядке, в котором они отсылались
- Пример: Процесс 0 посылает 2 сообщения
  Процесс 2 выставляет 2 receive, которые соответствуют send
  Порядок сохраняется

Пример

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
 int main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==1) {
       for(i=0;i<100;++i) data[i]=i;
       MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {
```

Пример (продолжение)

```
 MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,
        MPI_COMM_WORLD,&status);
 printf("P:%d Got data from processor %d \n",rank,
           status.MPI_SOURCE);
 MPI_Get_count(&status,MPI_FLOAT,&count);
 printf("P:%d Got %d elements \n",rank,count);
 printf("P:%d value[5]=%f \n",rank,value[5]);
 }
MPI_Finalize();
}
```

Пример (продолжение)

```
 MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,
        MPI_COMM_WORLD,&status);
 printf("P:%d Got data from processor %d \n",rank,
           status.MPI_SOURCE);
 MPI_Get_count(&status,MPI_FLOAT,&count);
 printf("P:%d Got %d elements \n",rank,count);
 printf("P:%d value[5]=%f \n",rank,value[5]);
 }
MPI_Finalize();
}
```

Program Output

P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000

Замер времени MPI_Wtime

- Время замеряется в секундах
- Выделяется интервал в программе

C:

```
double MPI_Wtime(void);
```

Пример:

```
int count, *buf, source;
MPI_Probe(MPI_ANY_SOURCE, 0, comm,
    &status);
source = status.MPI_SOURCE;
MPI_Get_count (status, MPI_INT, &count);
buf = malloc (count * sizeof (int));
MPI_Recv (buf, count, MPI_INT, source, 0,
    comm, &status);
```

Коллективные операции

- ***int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)***
- *buf* - адрес начала буфера посылки сообщения (выходной параметр)
- *count* - число передаваемых элементов в сообщении
- *datatype* - тип передаваемых элементов
- *source* - номер рассылающего процесса
- *comm* - идентификатор группы.

- ***int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)***
- *sbuf* – адрес начала буфера для аргументов операции op;
- *rbuf* – адрес начала буфера для результата операции op (выходной параметр);
- *count* – число аргументов у каждого процесса;
- *datatype* – тип аргументов;
- *op* – идентификатор глобальной операции;
- *root* – процесс-получатель результата;
- *comm* – идентификатор коммуникатора.
- **MPI_Reduce** аналогична предыдущей функции, но результат

Таблица операций

- MPI_MAX Maximum
- MPI_MIN Minimum
- MPI_PROD Product
- MPI_SUM Sum
- MPI_LAND Logical and
- MPI_LOR Logical or
- MPI_LXOR Logical exclusive or (xor)
- MPI_BAND Bitwise and
- MPI_BOR Bitwise or
- MPI_BXOR Bitwise xor
- MPI_MAXLOC Maximum value and location
- MPI_MINLOC Minimum value and location

Барьерная синхронизация

- int MPI_Barrier(MPI_Comm comm)

Коллективные функции

- MPI_ALLGATHER
- MPI_ALLGATHERV
- MPI_ALLREDUCE
- MPI_ALLTOALL
- MPI_ALLTOALLV
- MPI_BCAST
- MPI_GATHER
- MPI_GATHERV
- MPI_REDUCE
- MPI_SCAN
- MPI_SCATTER
- MPI_SCATTERV