ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В СТАНДАРТЕ MPI

Если у вас есть яблоко и у меня есть яблоко, и мы обмениваемся этими яблоками, то у вас и у меня остается по одному яблоку. А если у вас есть идея и у меня есть идея, и мы обмениваемся идеями, то у каждого из нас будет по две идеи.

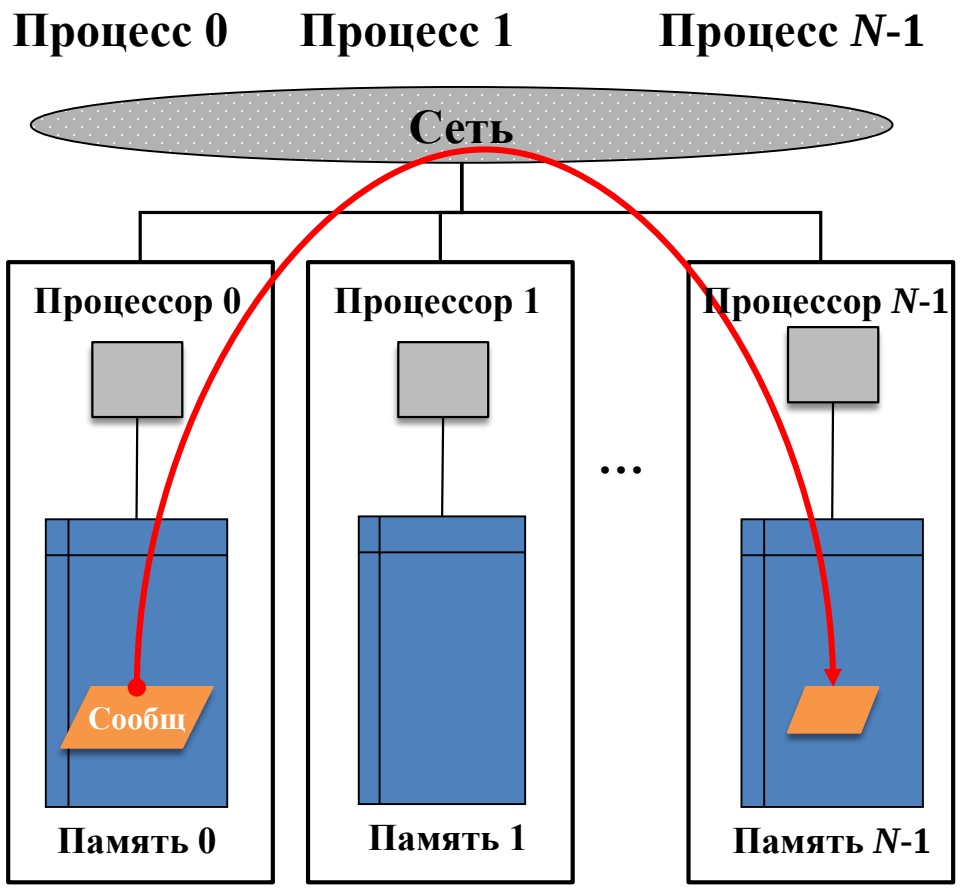
Дж.Б. Шоу

Суперкомпьютеры и их применение

Содержание

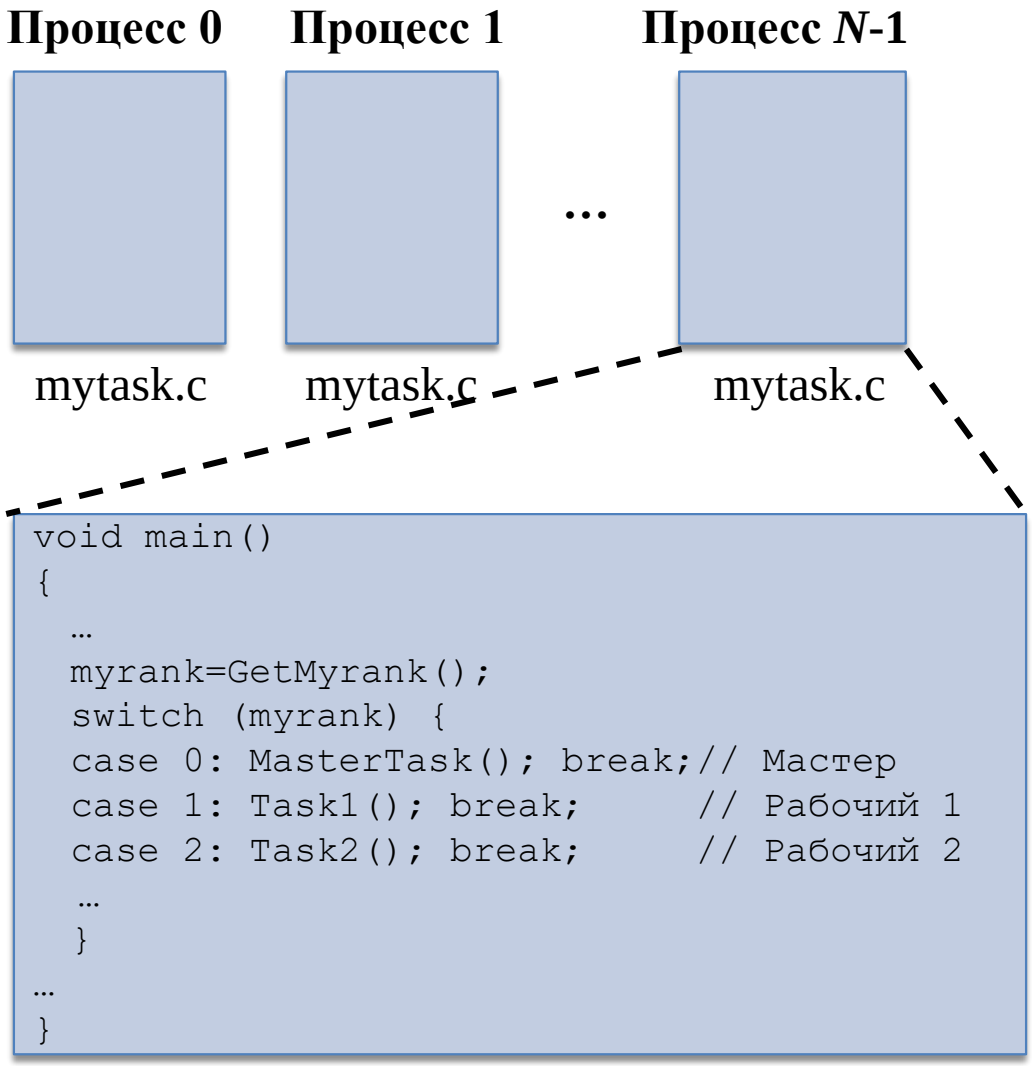
- Модель передачи сообщений для параллельного программирования в системах с распределенной памятью
- Модели SPMD и MPMD запуска параллельных программ
- Стандарт Message Passing Interface (MPI)
- Основные понятия и функции MPI

Модель передачи сообщений



- *Параллельное приложение* состоит из нескольких *процессов*, выполняющихся одновременно.
- Каждый процесс имеет приватную память.
- Обмены данными между процессами осуществляются посредством явной отправки/получения *сообщений*.
- Процессы, как правило, выполняются на различных процессорах.

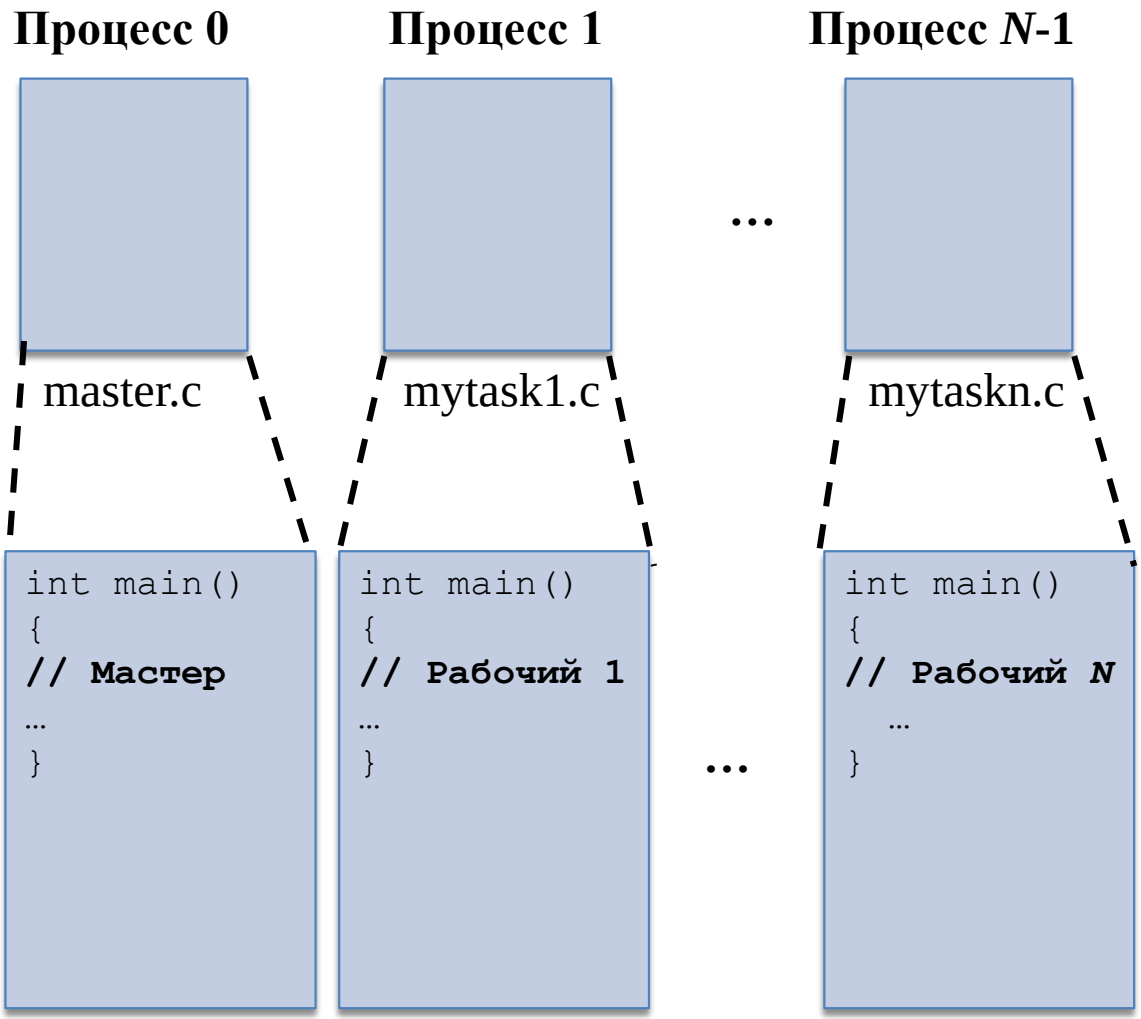
Модель SPMD




```
void main()
{
  …
  myrank=GetMyrank();
  switch (myrank) {
  case 0: MasterTask(); break;// Мастер
  case 1: Task1(); break;     // Рабочий 1
  case 2: Task2(); break;     // Рабочий 2
  …
  }
…
}
```

- *Модель SPMD (Single Program Multiple Data)* предполагает, каждый процесс параллельного приложения имеет один и тот же исходный код.

Модель MPMD

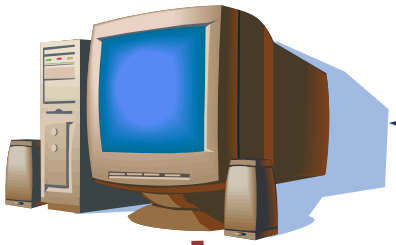


- *Модель MPMD (Multiple Program Multiple Data)* предполагает, процессы параллельного приложения имеют различные исходные коды.

Стандарт MPI

- *MPI (Message Passing Interface)* – стандарт, реализующий модель обмена сообщениями между параллельными процессами. Поддерживает модели выполнения SPMD и (начиная с версии 2.0) MPMD.
- Стандарт представляет собой набор спецификаций подпрограмм (более 120) на языках C, C++ и FORTRAN.
- Стандарт реализуется разработчиками в виде библиотек подпрограмм для различных аппаратно-программных платформ (кластеры, персональные компьютеры, …, Windows, Unix/Linux, …).
- Коммерческие (IntelMPI, MSMPI и др.) и свободно распространяемые реализации стандарта (MPICH и др.).
- Текущая версия стандарта: http://www.mpi-forum.org.

Цикл разработки MPI-программ



- Персональный компьютер
  - Первоначальная разработка и отладка
  - Отладка по результатам тестирования на суперкомпьютере

- Суперкомпьютер
  - Тестирование
  - Эксперименты

MPI-программа

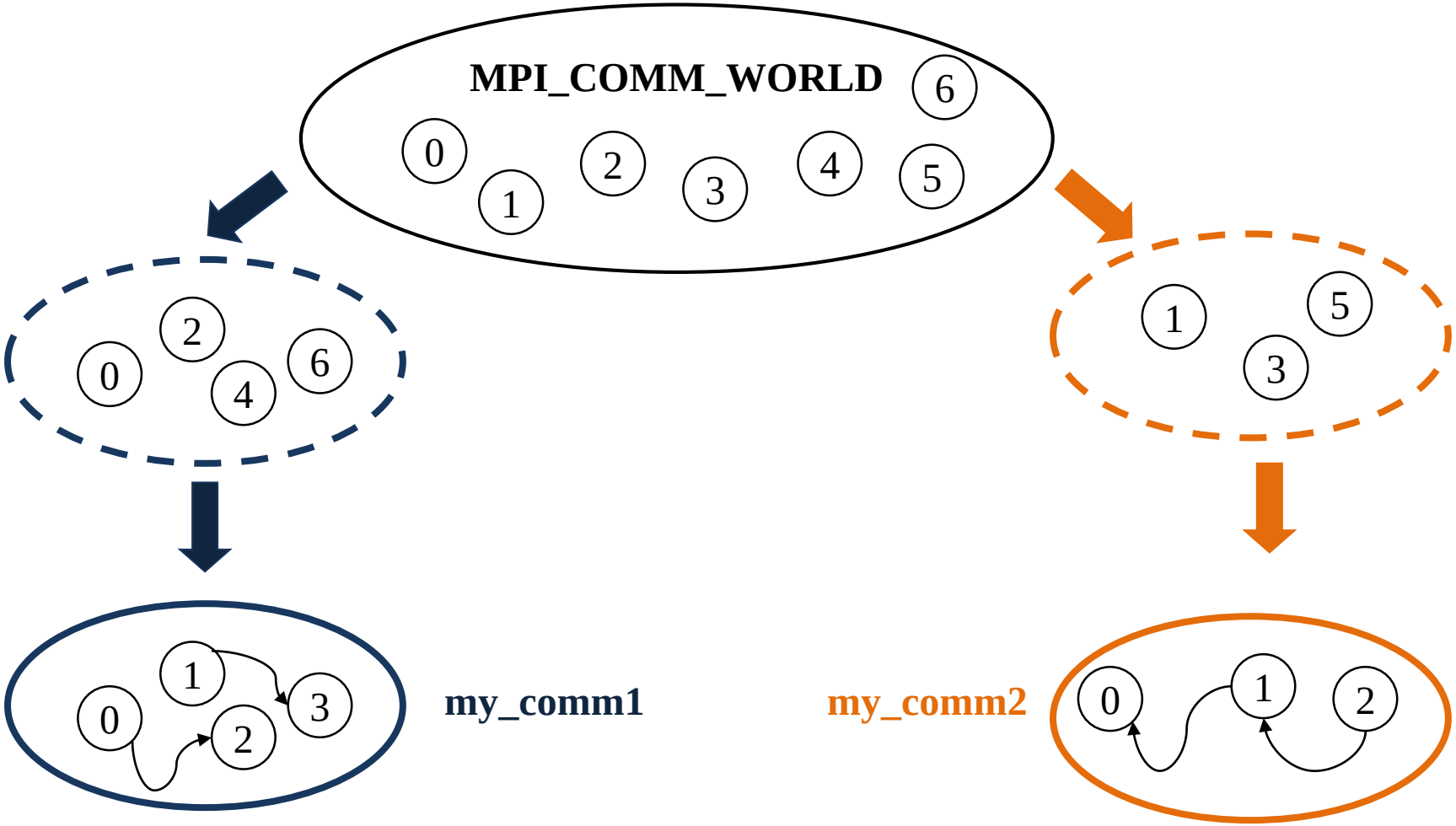
- *MPI-программа* – множество параллельных взаимодействующих процессов.
- Процессы порождаются один раз, в момент запуска программы средствами среды исполнения MPI программ*.
- Все процессы программы последовательно перенумерованы, начиная с 0. Номер процесса именуется *рангом* процесса.
- Каждый процесс работает в своем адресном пространстве, каких-либо общих данных нет. Единственный способ взаимодействия процессов – явный обмен сообщениями.

* Порождение дополнительных процессов и уничтожение существующих возможно только начиная с версии MPI-2.0.

Коммуникаторы

- Для локализации области взаимодействия процессов можно динамически создавать специальные программные объекты – *коммуникаторы*. Один и тот же процесс может входить в разные коммуникаторы.
- Взаимодействия процессов проходят в рамках некоторого коммуникатора. Сообщения, переданные в разных коммуникаторах, не пересекаются и не мешают друг другу.
- Атрибуты процесса MPI-программы:
  - номер коммуникатора;
  - номер в коммуникаторе (от 0 до n-1, n – число процессов в коммуникаторе).
- Стандартные коммуникаторы:
  - MPI_COMM_WORLD – все процессы приложения
  - MPI_COMM_SELF – текущий процесс приложения
  - MPI_COMM_NULL – пустой коммуникатор

Коммуникаторы



Сообщение

- *Сообщение* процесса – набор данных стандартного (определенного в MPI) или пользовательского типа.
- Основные атрибуты сообщения:
  - номер процесса-отправителя (получателя)
  - номер коммуникатора
  - тег (уникальный идентификатор) сообщения (целое число)
  - тип элементов данных в сообщении
  - количество элементов данных
  - указатель на буфер с сообщением

Структура MPI-программы


```
#include "mpi.h" // Подключение библиотеки

int main (int argc, char *argv[])
{
// Здесь код без использования MPI функций

  MPI_Init(&agrc, &argv); // Инициализация выполнения

// Здесь код, реализующий обмены

  MPI_Finalize(); // Завершение

// Здесь код без использования MPI функций

 return 0;
}
```

MPI-функции

- Имеют имена вида `MPI_...`
- Возвращают целое число – `MPI_SUCCESS` или код ошибки.
- Простые функции общего назначения:
  - ```
    // Количество процессов в коммуникаторе
    int MPI_Comm_size(MPI_Comm comm, int * size);
    ```
  - ```
    // Номер (ранг) процесса в коммуникаторе
    int MPI_Comm_rank(MPI_Comm comm, int * rank);
    ```
  - ```
    // Замер времени (сек.)
    double MPI_Wtime(void);
    ```
  - ```
    // Имя текущего процессорного узла
    int MPI_Get_processor_name(char * name, int * len);
    ```

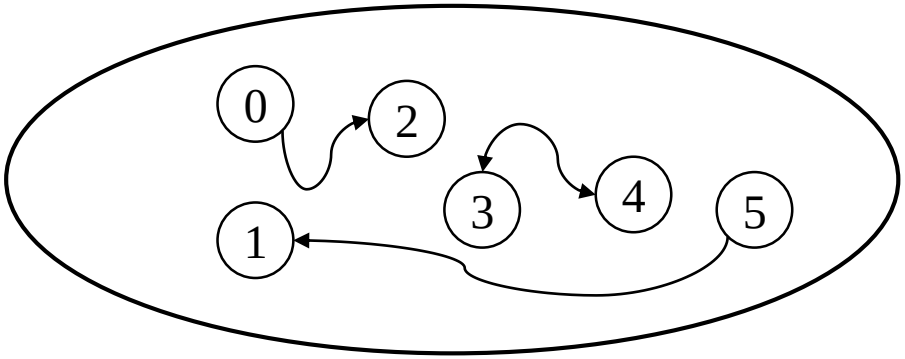
Простая MPI-программа


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

int main (int argc, char *argv[])
{
  int num, rank;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &num);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  printf("Привет! Я %d-й процесс из %d.\n", rank, num);
  MPI_Finalize();
  return 0;
}
```

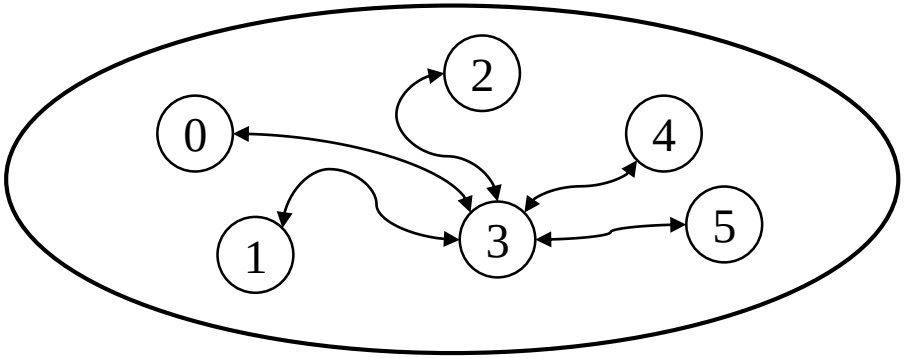
```
Привет! Я 1-й процесс из 4.
Привет! Я 0-й процесс из 4.
Привет! Я 4-й процесс из 4.
Привет! Я 3-й процесс из 4.
Привет! Я 2-й процесс из 4.
```

Виды взаимодействия процессов

- *Взаимодействие "точка-точка"* – обмен между двумя процессами одного коммуникатора.



- *Коллективное взаимодействие* – обмен между всеми процессами одного коммуникатора.

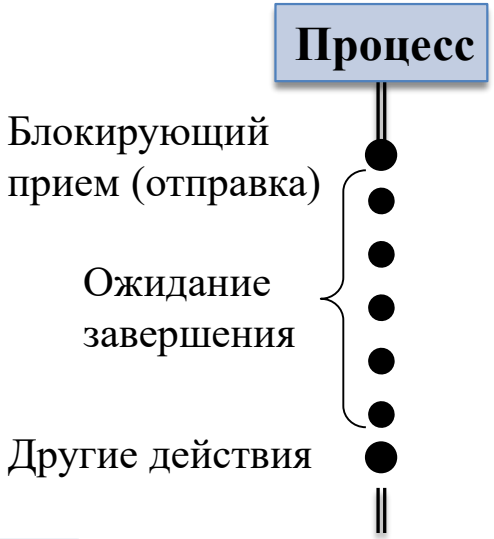


Взаимодействие "точка-точка"

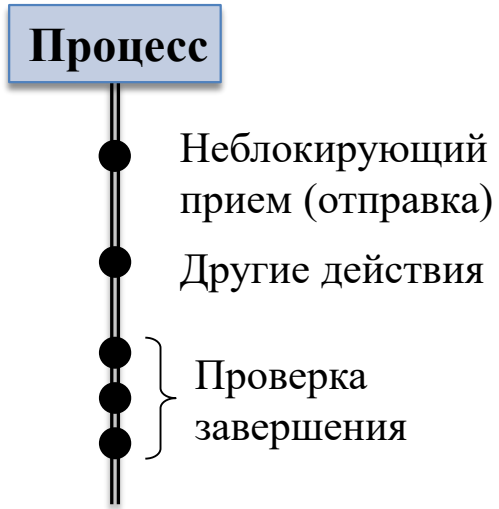
- Участвуют два процесса: отправитель сообщения и получатель сообщения.
  - Отправитель должен вызвать одну из функций отправки сообщения и явно указать атрибуты получателя (коммуникатор и номер в коммуникаторе) и тег сообщения.
  - Получатель должен вызвать одну из функций получения сообщения и указать (тот же) коммуникатор отправителя; получатель может не знать номер отправителя и/или тег сообщения.
- Свойства:
  - Сохранение порядка (если P0 передает P1 сообщения A и затем B, то P1 получит A, а затем B).
  - Гарантированное выполнение обмена (если P0 вызвал функцию отправки, а P1 вызвал функцию получения, то P1 получит сообщение от P0).

Виды коммуникационных функций "точка-точка"

- *Блокирующая* функция запускает операцию и возвращает управление процессу только после ее завершения.
  - После завершения допустима модификация отправленного (принятого) сообщения.



- *Неблокирующая* функция запускает операцию и возвращает управление процессу немедленно.
  - Факт завершения операции проверяется позднее (с помощью другой функции).
  - До завершения операции недопустима модификация отправляемого (получаемого) сообщения.



Блокирующие *vs* неблокирующие операции

- Блокирующие операции
  - Имеют более простую семантику.
  - Относительно легко используются в программе, не требуют дополнительных действий для завершения обмена.
  - Могут снизить быстродействие программы.
  - Могут привести к тупикам.
- Неблокирующие операции
  - Имеют более сложную семантику.
  - Относительно трудно используются в программе, требуют дополнительных действий для завершения обмена
  - Могут повысить быстродействие программы.
  - Не вызывают тупиков.

Режимы отправки сообщений "точка-точка"

- *Стандартный* – операция завершается сразу после отправки сообщения.
- *Синхронный* – операция завершается после приема подтверждения от адресата.
- *Буферизованный* – операция завершается, как только сообщение копируется в системный буфер для дальнейшей отправки.
- *"По готовности"* – операция начинается, если адресат инициализировал прием и завершается сразу после отправки.

Режимы отправки сообщений "точка-точка"

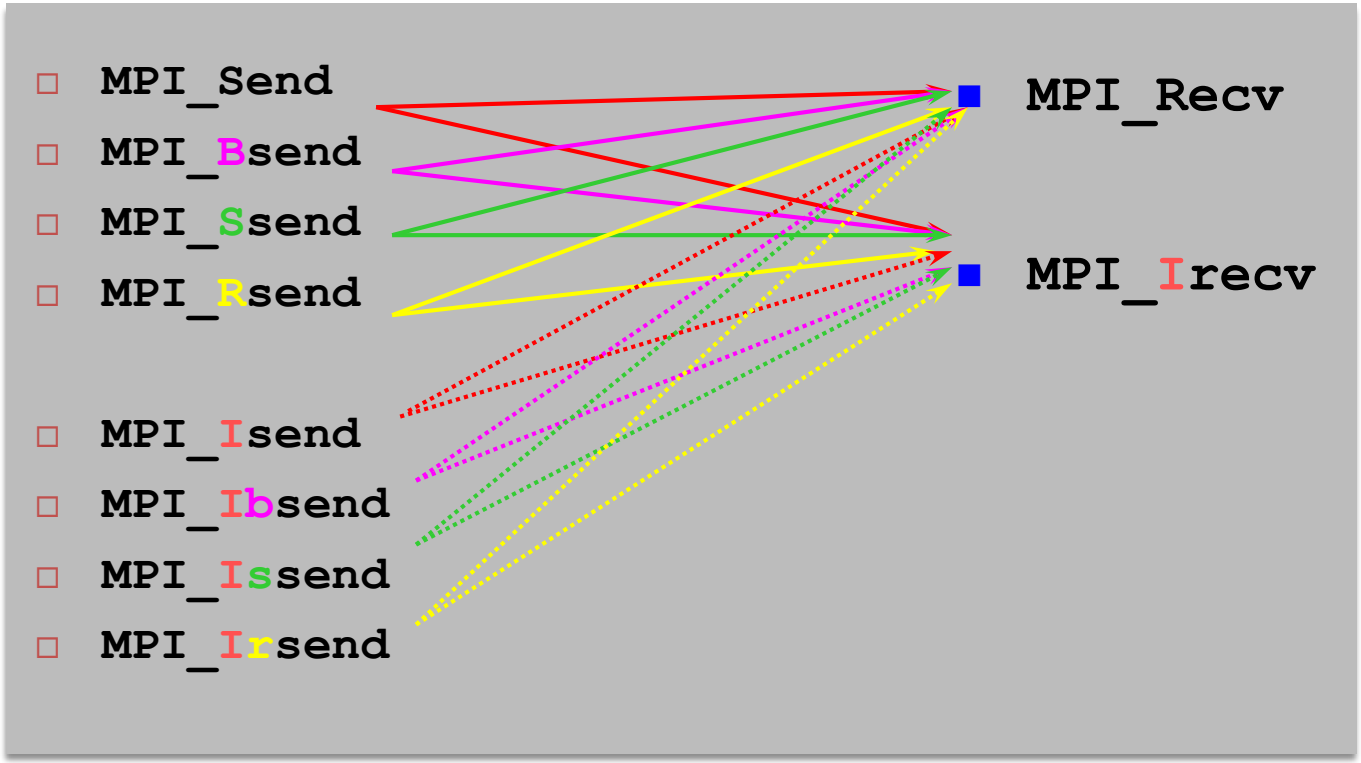
- Режим *по готовности* формально является наиболее быстрым, но используется достаточно редко, т.к. обычно сложно гарантировать готовность операции приема.
- *Стандартный* и *буферизованный* режимы также выполняются достаточно быстро, но могут приводить к большим расходам ресурсов (памяти), и могут быть рекомендованы для передачи коротких сообщений.
- *Синхронный* режим является наиболее медленным, т.к. требует подтверждения приема. В то же время этот режим наиболее надежен, и может быть рекомендован для передачи длинных сообщений.

Коммуникационные функции "точка-точка"

- Отправка: **MPI_[I][R, S, B]Send**
- Прием: **MPI_[I]Recv**

Блокирующие			**Неблокирующие**		
Отправка		***Прием***	***Отправка***		***Прием***
Стандартная	`MPI_Send`	`MPI_Recv`	*Стандартная*	`MPI_Isend`	`MPI_Irecv`
Синхронная	`MPI_Ssend`		*Синхронная*	`MPI_Issend`	
Буферизованная	`MPI_Bsend`		*Буферизованная*	`MPI_Ibsend`	
По готовности	`MPI_Rsend`		*По готовности*	`MPI_Irsend`	

Коммуникационные функции "точка-точка"



Блокирующая стандартная отправка сообщения

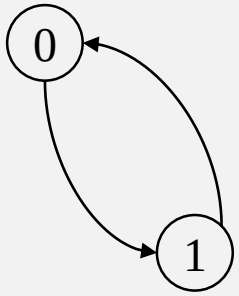
- **`int MPI_Send`**
  - `IN void * buf` – указатель на буфер с сообщением
  - `IN int count` – количество элементов в буфере
  - `IN MPI_Datatype datatype` – MPI-тип данных элементов в буфере
  - `IN int dest` – номер процесса-получателя
  - `IN int tag` – тег сообщения
  - `IN MPI_Comm comm` – коммуникатор

Блокирующее стандартное получение сообщения

- **`int MPI_Recv`**
  - `OUT  void * buf` – указатель на буфер с сообщением
  - `IN  int count` – количество элементов в буфере
  - `IN  MPI_Datatype datatype` – MPI-тип данных элементов в буфере
  - `IN  int src` – номер процесса-отправителя
  - `IN  int tag` – тег сообщения
  - `IN  MPI_Comm comm` – коммуникатор
  - `OUT  MPI_Status* status` – информация о фактически полученных данных (указатель на структуру с двумя полями: `source` – номер процесса-источника, `tag` – тег сообщения)

Пример программы


```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int numtasks, rank, dest, src, rc, tag=777;
  char inmsg, outmsg='x';
  MPI_Status Stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    dest = 1; src = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD, &Stat);
} else
  if (rank == 1) {
    dest = 0; src = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```



Неблокирующая стандартная отправка сообщения

- **`int MPI_Isend`**
  - `IN  void * buf` – указатель на буфер с сообщением
  - `IN  int count` – количество элементов в буфере
  - `IN  MPI_Datatype datatype` – MPI-тип данных элементов в буфере
  - `IN  int dest` – номер процесса-получателя
  - `IN  int tag` – тег сообщения
  - `IN  MPI_Comm comm` – коммуникатор
  - `OUT  MPI_Request *request` – дескриптор операции (для последующей проверки завершения операции)

Неблокирующее стандартное получение сообщения

- **`int MPI_Irecv`**
  - `OUT  void * buf` – указатель на буфер с сообщением
  - `IN  int count` – количество элементов в буфере
  - `IN  MPI_Datatype datatype` – MPI-тип данных элементов в буфере
  - `IN  int src` – номер процесса-отправителя
  - `IN  int tag` – тег сообщения
  - `IN  MPI_Comm comm` – коммуникатор
  - `OUT  MPI_Request *request` – дескриптор операции (для последующей проверки завершения операции)

Завершение неблокирующих обменов

- Проверка завершения

  int MPI_Test

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

  - int MPI_Testany (…)
  - int MPI_Testall (…)
  - int MPI_Testsome (…)

- Ожидание завершения

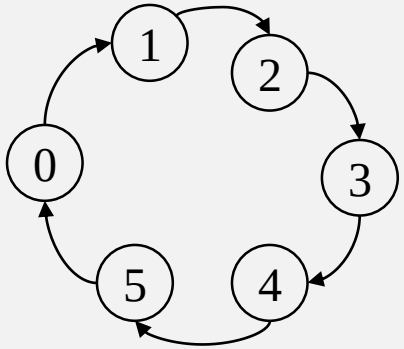
  int MPI_Wait

  (MPI_Request *request, MPI_Status *status)

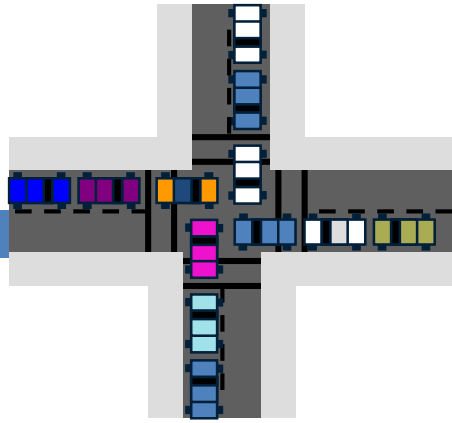
  - int MPI_Waitany (…)
  - int MPI_Waitall (…)
  - int MPI_Waitsome (…)

Пример


```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  int numtasks, rank, next, prev, buf[2], tag1=111, tag2=222;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = rank - 1;
  if (prev < 0) prev = numtasks - 1;
   next = rank + 1;
  if (next > numtasks - 1) next = 0;
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
  MPI_Waitall(4, reqs, stats);

  MPI_Finalize();
}
```



Тупики (deadlocks)



- Гарантированный тупик

P0	P1
`MPI_Recv (от процесса P1);` `MPI_Send (процессу P1);`	`MPI_Recv (от процесса P0);` `MPI_Send (процессу P0);`

- Возможный тупик

P0	P1
`MPI_Send (процессу P1);` `MPI_Recv (от процесса P1);`	`MPI_Send (процессу P0);` `MPI_Recv (от процесса P0);`

Разрешение тупиков

P0

```
MPI_Send (процессу P1);
MPI_Recv (от процесса P1);
```

P1

```
MPI_Recv (от процесса P0);
MPI_Send (процессу P0);
```

P0

```
MPI_Send (процессу P1);
MPI_Recv (от процесса P1);
```

P1

```
MPI_Irecv (от процесса P0);
MPI_Send (процессу P0);
MPI_Wait
```

P0

```
// Совмещенные прием и передача
MPI_Sendrecv
```

P1

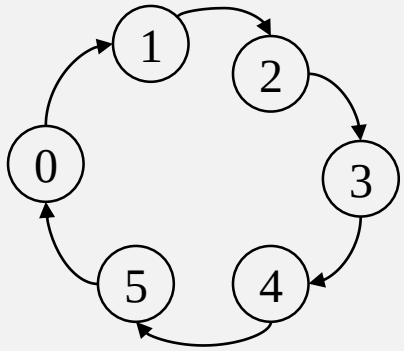
```
// Совмещенные прием и передача
MPI_Sendrecv
```

Совмещение приема и передачи сообщения

- **`int MPI_Sendrecv`**
  - `OUT void * sbuf` - адрес начала буфера с посылаемым сообщением;
  - `IN int scount` - число передаваемых элементов в сообщении;
  - `IN MPI_Datatype stype` - тип передаваемых элементов;
  - `IN int dest` - номер процесса-получателя;
  - `IN int stag` - идентификатор посылаемого сообщения;
  - `OUT void *rbuf` - адрес начала буфера приема сообщения;
  - `IN int rcount` - число принимаемых элементов сообщения;
  - `IN MPI_Datatype rtype` - тип принимаемых элементов;
  - `IN int source` - номер процесса-отправителя;
  - `IN int rtag` - идентификатор принимаемого сообщения;
  - `IN MPI_Comm comm` - идентификатор коммуникатора;
  - `OUT MPI_Status status` - параметры принятого сообщения.
- Объединяет в едином запросе посылку и прием сообщений и гарантирует отсутствие тупиков.
- Принимающий и отправляющий процессы могут являться одним и тем же процессом. Буфера приема и посылки обязательно должны быть различными. Сообщение, отправленное операцией MPI_Sendrecv, может быть принято обычным образом, и точно также операция MPI_Sendrecv может принять сообщение, отправленное обычной операцией MPI_Send.

Пример


```
#include "mpi.h"
#include <stdio.h>
int rank, numtasks, prev, next, buf[2];
MPI_Status status1, status2;

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev=rank-1; if (prev < 0) prev=numtasks-1;
  next=rank+1; if (next > size-1) next=0;
  MPI_Sendrecv(rank, 1, MPI_INTEGER, prev, 777, buf[1], 1, MPI_INTEGER, next,
777, MPI_COMM_WORLD, status2);
  MPI_Sendrecv(rank, 1, MPI_INTEGER, next, 888, buf[0], 1, MPI_INTEGER, prev,
888, MPI_COMM_WORLD, status1);
  printf("Process %d, prev=%d, next=%d\n", rank, buf[0], buf[1]);

  MPI_Finalize();
  return 0;
}
```



Получение сообщений

- "Джокеры"
  - ранг процесса `MPI_ANY_SOURCE`
  - тег сообщения `MPI_ANY_TAG`
- Информация об ожидаемом сообщении
  - С блокировкой
    ```
    int MPI_Probe(int src, int tag,
    MPI_Comm comm, MPI_Status * status);
    ```
  - Без блокировки
    ```
    int MPI_Iprobe(int src, int tag,
    MPI_Comm comm, int * flag, MPI_Status *
    status);
    ```
  - Количество элементов в принятом сообщении
    ```
    int MPI_Get_count(MPI_Status * status,
    MPI_Datatype type, int * cnt);
    ```

Пример

- Процесс-получатель не знает заранее длины ожидаемого сообщения.

```
MPI_Probe(MPI_ANY_SOURCE, msgtag_INT, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_INT, &cnt);
buffer = malloc(sizeof(int) * cnt);
MPI_Recv(buffer, cnt, MPI_INT, MPI_ANY_SOURCE, msgtag_INT, MPI_COMM_WORLD,
&status);
```

- Процесс-получатель собирает сообщения от разных отправителей с содержимым различных типов.

```
for (i=0; i<3; i++) {
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  switch (status.MPI_TAG) {
    case msgtag_FLOAT: MPI_Recv(floatBuf, floatBufSize, MPI_FLOAT,
MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status); break;
    case msgtag_INT: MPI_Recv(intBuf, intBufSize, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status); break;
    case msgtag_CHAR: MPI_Recv(charBuf, charBufSize, MPI_CHAR, MPI_ANY_TAG,
MPI_COMM_WORLD, &status); break;
  }
}
```

Коллективные операции

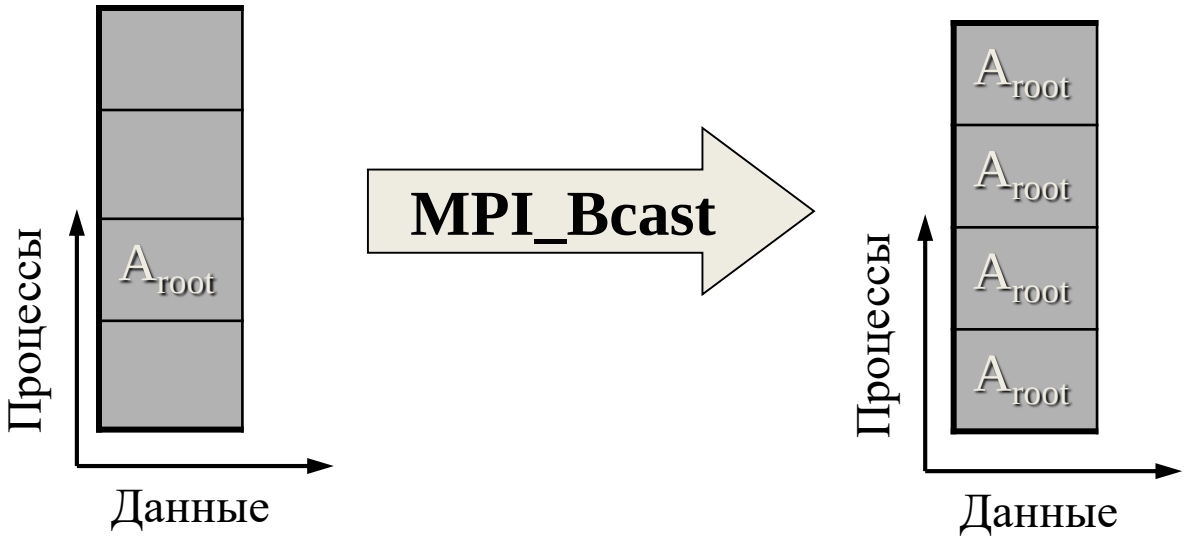
- Прием и/или передачу выполняют одновременно *все* процессы коммуникатора.
- Коллективная функция имеет большое количество параметров, часть которых нужна для приема, а часть для передачи. При вызове в разных процессах та или иная часть игнорируется.
- Значения *всех* параметров коллективных функций (за исключением адресов буферов) должны быть идентичными во всех процессах.
- MPI назначает теги для сообщений автоматически.

Барьерная синхронизация

- `int` **`MPI_Barrier`**`(MPI_Comm comm)`
  - Обеспечивает *синхронизацию* процессов – одновременное достижение процессами указанной точки вычислений.
  - Должна вызываться всеми процессами коммуникатора.
  - Продолжение вычислений любого процесса произойдет по окончании выполнения функции MPI_Barrier всеми процессами коммуникатора.

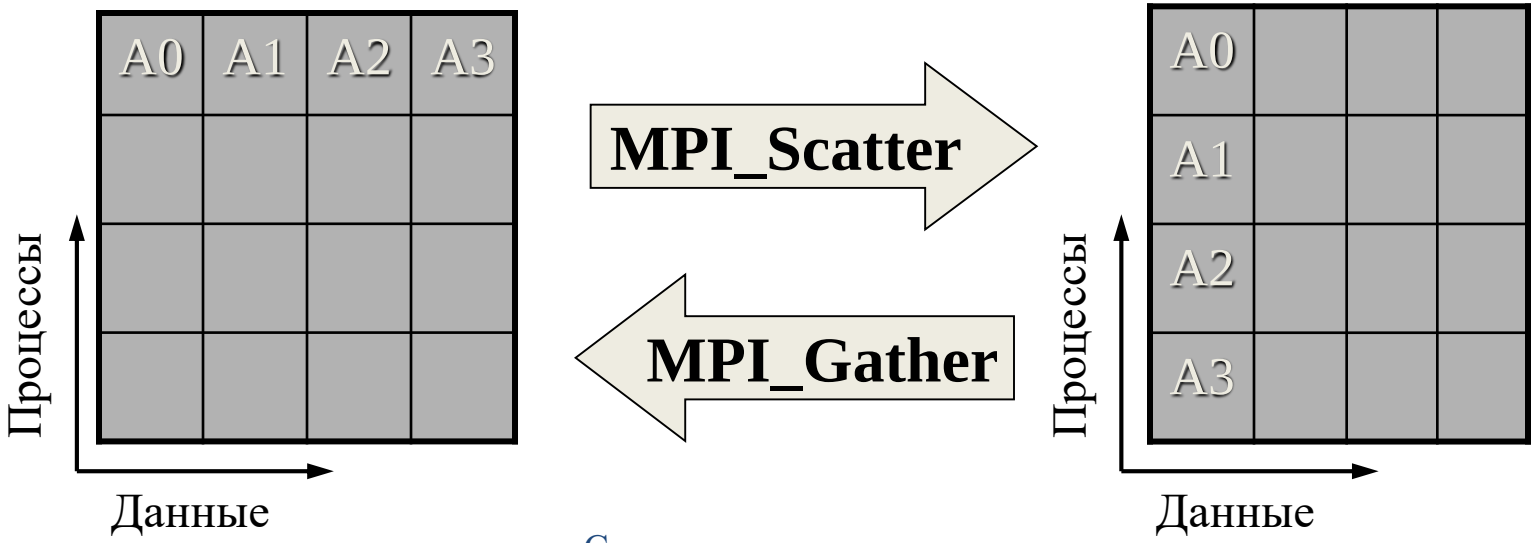
Широковещательная рассылка

- `int` **`MPI_Bcast`**
  - `IN/OUT void *buf` – указатель на буфер с сообщением (отправляемым – для процесса с рангом `root`, принимаемым – для остальных процессов)
  - `IN int count` – количество элементов в буфере
  - `IN MPI_Datatype datatype` – MPI-тип данных элементов в буфере
  - `IN int root` – ранг процесса, выполняющего рассылку данных
  - `IN MPI_Comm comm` – коммуникатор

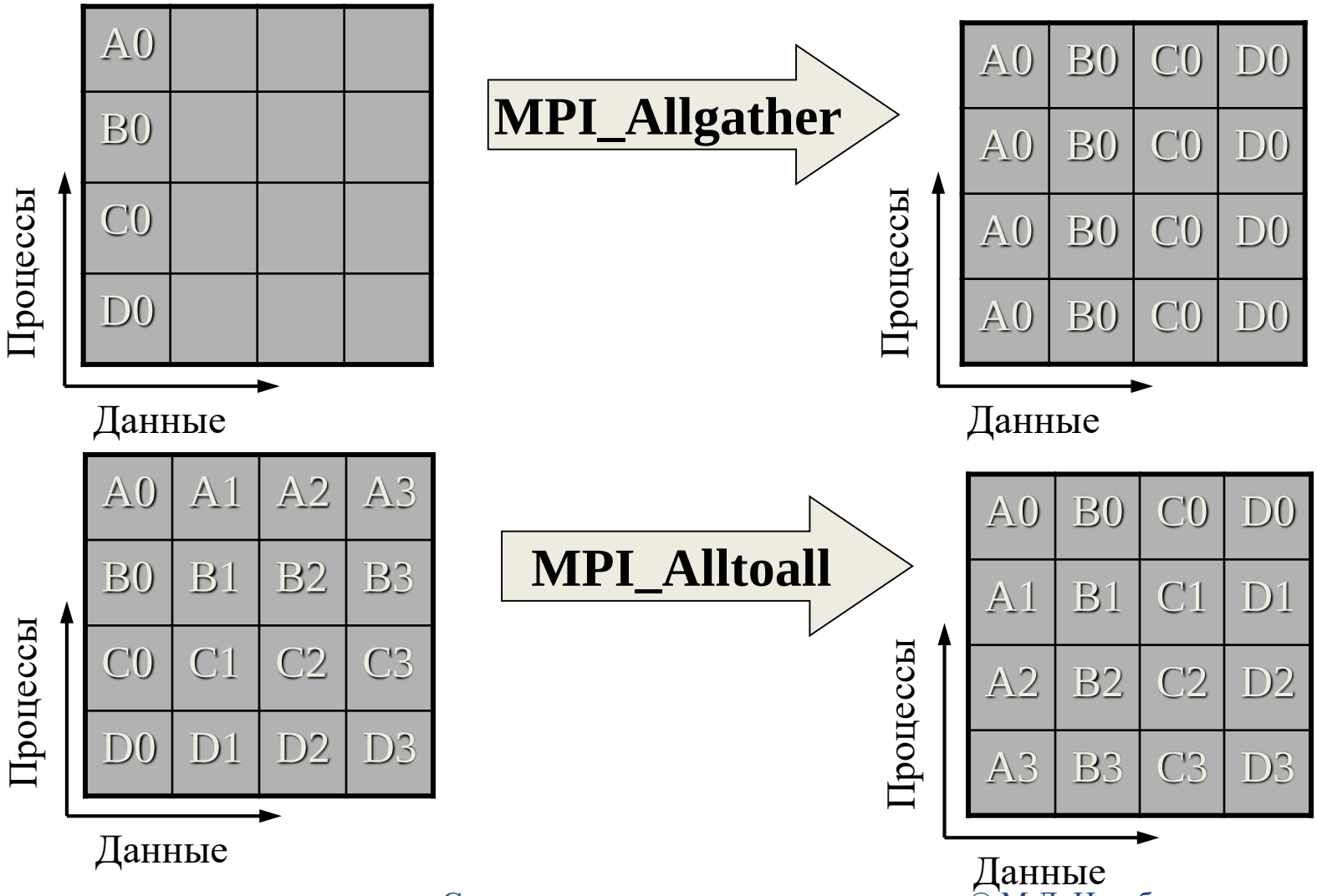


Коллективный прием сообщения

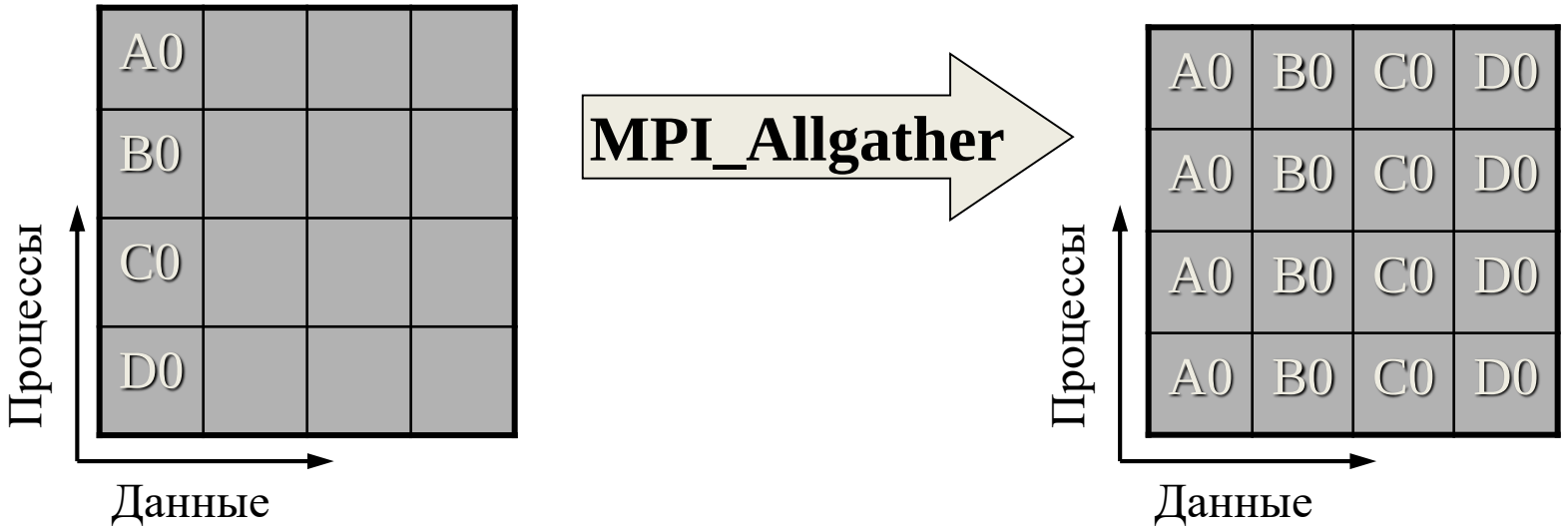
- Сборка элементов данных из буферов всех процессов в буфере процесса с рангом `root`
  `int MPI_Gather``(void * sbuf, int scount, MPI_Datatype stype, void * rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm);`
- Рассылка элементов данных из буфера процесса с рангом `root` в буферы всех процессов (обратная к `MPI_Gather`)
  `int MPI_Scatter`



Широковещательные прием и передача



Широковещательный прием

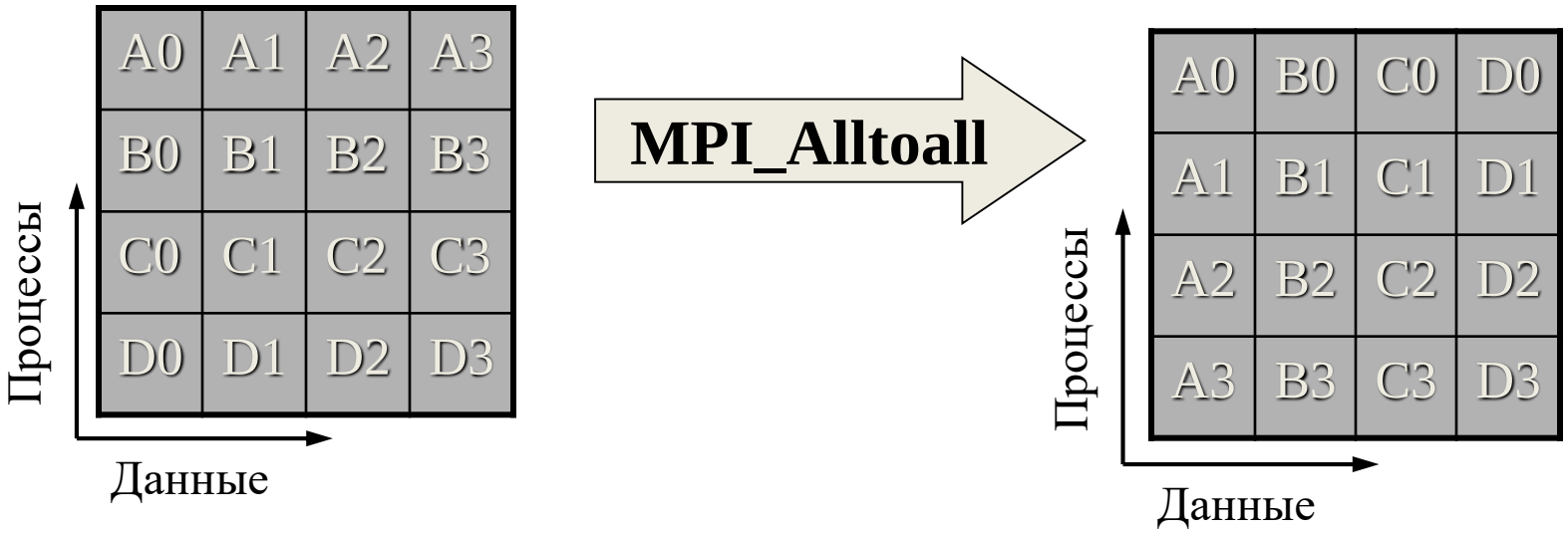



```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,
    void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm)
```

Широковещательная передача


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

- **sbuf**, **scount**, **stype** - параметры передаваемых сообщений,
- **rbuf**, **rcount**, **rtype** - параметры принимаемых сообщений
- **comm** - коммуникатор, в рамках которого выполняется передача данных



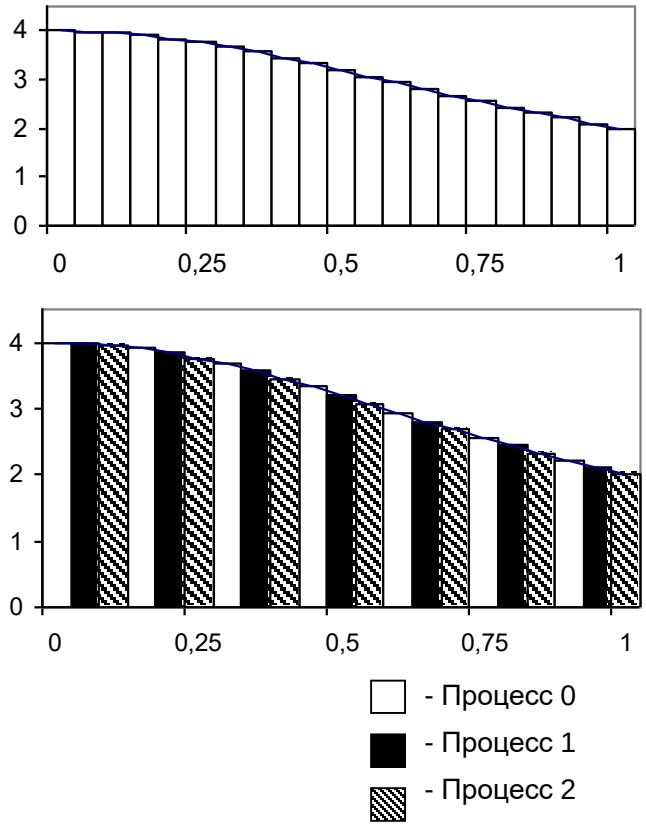
Глобальные операции над данными

- Выполнение `count` независимых глобальных операций `op` над соответствующими элементами массивов `sbuf`.
  Результат выполнения над *i*-ми элементами `sbuf` записывается в *i*-й элемент массива `rbuf` процесса с рангом `root`.
  `int MPI_Reduce``(void * sbuf, void * rbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm);`
- Глобальные операции:
  - `MPI_MAX, MPI_MIN`
  - `MPI_SUM, MPI_PROD`
  - …
  - `MPI_Op_Create()`

Пример: вычисление π

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

- Метод прямоугольников для численного интегрирования.
- Циклическая схема распределения вычислений
  - Получаемые на отдельных процессорах частные суммы должны быть просуммированы.



Пример: вычисление π


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

int N; // Количество интервалов
double PI25DT=3.14159265358979323846264643;

double f(double a)
{
  return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[])
{
  int rank, num, i;
  double mypi, pi, h, sum, x, t1, t2;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &num);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    printf("Number of intervals? "\n);
    scanf("%d\n", &N)
    t1 = MPI_Wtime();
  }
  MPI_Bcast(&N, 1, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
  // Вычисление локальной суммы
  h = 1.0 / (double) N;
  sum = 0.0;
  for (i = rank + 1; i <= N; i += num) {
     x = h * ((double)i  0.5);
     sum += f(x);
  }
  mypi = h * sum;

  // Сложение локальных сумм (редукция)
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_COMM_WORLD);

// вывод результатов
  if (rank == 0) {
    t2 = MPI_Wtime();
    printf("Pi is %.16f,
            Error is %.16f\n",
            pi, fabs(pi  PI25DT));
    printf("Time is %f\n", t2-t1);
  }
  MPI_Finalize();
  return 0;
}
```

Типы данных в MPI

- Поддерживаются стандартные типы данных и возможность создания пользовательских типов данных.
- Приложение MPI может работать на гетерогенном вычислительном комплексе.
  - Одни и те же типы данных на разных машинах могут иметь разное представление:
    - Стандарты IEEE, IBM, Cray представления float-чисел.
    - Кодировки Windows и KOI-8.
    - Ориентация байтов в числах (на компьютерах с процессорами Intel младший байт занимает младший адрес, у компьютеров с др. процессорами – наоборот).
    - "Выравнивание" данных (добавление "пустот" в данные структурного типа для кратности их размера 2, 4 или 8 для ускорения доступа).
  - Процессы обмениваются данными в формате XDR (eXternal Data Representation), принятом в Internet. Поэтому среда исполнения должна знать не просто количество передаваемых байтов, но и тип содержимого.

Стандартные типы данных MPI

Тип MPI	Соответствующий 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	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-
MPI_PACKED	-

Пример


```
#define msgTag 777
struct {
  int i;
  float f[4];
  char c[8];
} myStruct;
…
MPI_Send(&myStruct, sizeof(myStruct), MPI_BYTE, dest, msgTag,
MPI_COMM_WORLD );
```

```
char tempBuf[sizeof(s)];

MPI_Recv(tempBuf, sizeof(tempBuf), MPI_BYTE, src, msgTag,
MPI_COMM_WORLD, &status);
```

- Ненадежно: отправитель и получатель могут иметь разные двоичные представления (и, следовательно, размер) сообщения.

Пример


```
#define msgTag 777
struct {
  int i;
  float f[4];
  char c[8];
} myStruct;
…
MPI_Send(&myStruct.i, 1, MPI_INT, dest, msgTag, MPI_COMM_WORLD );
MPI_Send( myStruct.f, 4, MPI_FLOAT, dest, msgTag+1, MPI_COMM_WORLD );
MPI_Send( myStruct.c, 8, MPI_CHAR, dest, msgTag+2, MPI_COMM_WORLD );
…
```

```
MPI_Recv(…);
MPI_Recv(…);
MPI_Recv(…);
```

- Неэффективно: каждый элемент структуры передается отдельно.

Пример


```
int bufSize = 0;
void *tempBuf;

MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &bufSize);
MPI_Pack_size(4, MPI_FLOAT, MPI_COMM_WORLD, &bufSize);
MPI_Pack_size(8, MPI_CHAR, MPI_COMM_WORLD, &bufSize );
tempBuf = malloc(bufSize);
MPI_Pack(&myStruct.i, 1, MPI_INT, tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD);
MPI_Pack( myStruct.f, 4, MPI_FLOAT, tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD);
MPI_Pack( myStruct.c, 8, MPI_CHAR, tempBuf, sizeof(tempBuf), &bufPos, MPI_COMM_WORLD);
MPI_Send( tempBuf, bufPos, MPI_BYTE, dest, msgTag, MPI_COMM_WORLD);
```

```
int bufPos = 0;
char tempBuf[sizeof(myStruct)];
…
MPI_Recv(tempBuf, sizeof(tempBuf), MPI_BYTE, src, msgTag, MPI_COMM_WORLD, &status);
MPI_Unpack(tempBuf, sizeof(tempBuf), &bufPos,&s.i, 1, MPI_INT, MPI_COMM_WORLD);
MPI_Unpack(tempBuf, sizeof(tempBuf), &bufPos, s.f, 4, MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(tempBuf, sizeof(tempBuf), &bufPos, s.c, 8, MPI_CHAR, MPI_COMM_WORLD);
```

- Более эффективно: данные упаковываются перед пересылкой. Неудобно при программировании.

Пользовательские типы данных

- Создание типа "массив"

```
int MPI_Type_contiguous(int count,
MPI_Datatype oldtype, MPI_Datatype *newtype);
```

```
#define N 100
int A[N];
MPI_Datatype MPI_INTARRAY100;
…
MPI_Type_contiguous(N, MPI_INT, & MPI_INTARRAY100);
MPI_Type_commit(&MPI_INTARRAY100);
…
MPI_Send(A, 1, MPI_INTARRAY100, ... );
// то же, что и MPI_Send(A, N, MPI_INT, ... );
…
MPI_Type_free(&intArray100);
```

Группы процессов

- Процессы параллельной программы объединяются в *группы*. В группу могут входить все процессы параллельной программы или в группе может находиться только часть имеющихся процессов. Один и тот же процесс может принадлежать нескольким группам,
- Управление группами процессов предпринимается для создания на их основе коммуникаторов.
- Группы процессов могут быть созданы только из уже существующих групп. В качестве исходной группы может быть использована группа, связанная с предопределенным коммуникатором `MPI_COMM_WORLD`.

```
int MPI_Comm_group ( MPI_Comm comm, MPI_Group *group )
```

Группы процессов

- На основе существующих групп, могут быть созданы новые группы
  - создание новой группы ***newgroup*** из существующей группы ***oldgroup***, которая будет включать в себя ***n*** процессов, ранги которых перечисляются в массиве ***ranks***:

```
int MPI_Group_incl(MPI_Group oldgroup,int n, int *ranks,
                   MPI_Group *newgroup);
```

  - создание новой группы ***newgroup*** из группы ***oldgroup***, которая будет включать в себя ***n*** процессов, ранги которых не совпадают с рангами, перечисленными в массиве ***ranks***:

```
int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks,
                   MPI_Group *newgroup);
```

Группы процессов

- На основе существующих групп, могут быть созданы новые группы
  - создание новой группы ***newgroup*** как объединения групп ***group1*** и ***group2***:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
        MPI_Group *newgroup);
```

  - создание новой группы ***newgroup*** как пересечения групп ***group1*** и ***group2***:

```
int MPI_Group_intersection ( MPI_Group group1,
      MPI_Group group2, MPI_Group *newgroup );
```

  - создание новой группы ***newgroup*** как разности групп ***group1*** и ***group2***:

```
int MPI_Group_difference ( MPI_Group group1,
      MPI_Group group2, MPI_Group *newgroup );
```

Группы процессов

- Получение информации о группе процессов:
  - получение количества процессов в группе:

```
int MPI_Group_size ( MPI_Group group, int *size );
```

  - получение ранга текущего процесса в группе:

```
int MPI_Group_rank ( MPI_Group group, int *rank );
```

- После завершения использования группа должна быть удалена:

```
int MPI_Group_free ( MPI_Group *group );
```

Коммуникаторы

- *Коммуникатор* – специально создаваемый служебный объект MPI, объединяющий в себе группу процессов и ряд дополнительных параметров (*контекст*), используемых при выполнении операций передачи данных,
- Будем рассматривать управление *интракоммуникаторами*, используемыми для операций передачи данных внутри одной группы процессов.

Коммуникаторы

- Создание коммуникатора:
  - дублирование уже существующего коммуникатора:

```
int MPI_Comm_dup ( MPI_Comm oldcom, MPI_comm *newcomm );
```

  - создание нового коммуникатора из подмножества процессов существующего коммуникатора:

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group,
       MPI_Comm *newcomm);
```

  - Операция создания коммуникаторов является коллективной (должна выполняться всеми процессами исходного коммуникатора). После завершения использования коммуникатор должен быть удален:

```
int MPI_Comm_free ( MPI_Comm *comm );
```

Коммуникаторы

- Одновременное создание нескольких коммуникаторов:

```
int MPI_Comm_split ( MPI_Comm oldcomm, int split, int key,
  MPI_Comm *newcomm ),
где
- oldcomm – исходный коммуникатор,
- split   – номер коммуникатора, которому должен
принадлежать процесс,
- key     – порядок ранга процесса в создаваемом
коммуникаторе,
- newcomm – создаваемый коммуникатор
```

  - Вызов функции *MPI_Comm_split* должен быть выполнен в каждом процессе коммуникатора *oldcomm*,
  - Процессы разделяются на непересекающиеся группы с одинаковыми значениями параметра *split*. На основе сформированных групп создается набор коммуникаторов. Порядок нумерации процессов соответствует порядку значений параметров *key* (процесс с большим значением параметра *key* будет иметь больший ранг).

Коммуникаторы


```
#include "stdio.h"
#include "mpi.h"

void main(int argc, char *argv[])
{
  int num, p;
  int Neven, Nodd, members[6], even_rank, odd_rank;
  MPI_Group group_world, even_group, odd_group;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &num);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

Коммуникаторы


```
  Neven = (p + 1)/2;
  Nodd = p - Neven;
  members[0] = 2;
  members[1] = 0;
  members[2] = 4;
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Group_incl(group_world, Neven, members, &even_group);
  MPI_Group_excl(group_world, Neven, members,  &odd_group);
  MPI_Barrier(MPI_COMM_WORLD);
  if(num == 0) {
    printf("Number of processes is %d\n", p);
    printf("Number of odd processes is %d\n", Nodd);
    printf("Number of even processes is %d\n", Neven);
    printf("members[0] is assigned rank %d\n", members[0]);
    printf("members[1] is assigned rank %d\n", members[1]);
    printf("members[2] is assigned rank %d\n", members[2]);
    printf("\n");
    printf("     num     even      odd\n");
  }
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Group_rank(even_group, &even_rank);
  MPI_Group_rank( odd_group,  &odd_rank);
  printf("%8d %8d %8d\n",num, even_rank, odd_rank);
  MPI_Finalize();
}
```

```
Number of processes is 4
Number of odd processes is 2
Number of even processes is 2
members[0] is assigned rank 2
members[1] is assigned rank 0
members[2] is assigned rank 4

     Iam      even       odd
       0         1    -32766
       2         0    -32766
       1    -32766         0
       3    -32766         1
```

MPI-2: нововведения

- Динамическое порождение процессов
  - Разрешается создание и уничтожение процессов по ходу выполнения программы.
  - Предусмотрен специальный механизм, позволяющий устанавливать связь между только что порожденными процессами и уже работающей частью MPI-программы.
  - Имеется возможность установить связь между двумя приложениями даже в том случае, когда ни одно из них не было инициатором запуска другого.
- Одностороннее взаимодействие процессов
  - Обмен сообщениями по схеме Put/Get вместо традиционной схемы Send/Receive. Активной стороной может быть один процесс (при обмене не требуется активность отправителя либо получателя).
- Параллельный ввод-вывод
  - Специальный интерфейс для работы процессов с файловой системой.
- Расширенные коллективные операции
  - Во многие коллективные операции добавлена возможность взаимодействия между процессами, входящими в разные коммуникаторы.
  - Многие коллективные операции внутри коммуникатора могут выполняться в режиме, при котором входные и выходные буфера совпадают.
- Интерфейс для C++

Заключение

- Модель передачи сообщений для параллельного программирования в системах с распределенной памятью
- Модели SPMD и MPMD запуска параллельных программ
- Стандарт Message Passing Interface (MPI)
- Основные понятия и функции MPI