Технология MPI

(Message Passing Interface)

д.т.н. Елена Янакова

Компетенции

Способностью осуществлять поиск, хранение, обработку и анализ информации из различных источников и баз данных, представлять ее в требуемом формате с использованием информационных, компьютерных и сетевых технологий.

Знание основ построения и функционирования вычислительных систем, в частности, многопроцессорных и распределенных.

История MPI

MPI (Message Passing Interface, интерфейс передачи сообщений) – это программный интерфейс для передачи сообщений между процессами, параллельно выполняющими одну общую задачу. Определяет API (варианты для Си, C++, Fortran, Java).

Стандарт MPI	Поддерживающие функции
MPI 1.0 и 1.1 1995 год	– передача и получение сообщений между отдельными процессами; – коллективные взаимодействия процессов; – взаимодействия в группах процессов; – реализация топологий процессов.
MPI 2.0 1998 год	– динамическое порождение процессов и управление процессами; – односторонние коммуникации (Get/Put); – параллельный ввод и вывод; – расширенные коллективные операции (процессы могут выполнять коллективные операции не только внутри одного коммуникатора, но и в рамках нескольких коммуникаторов).

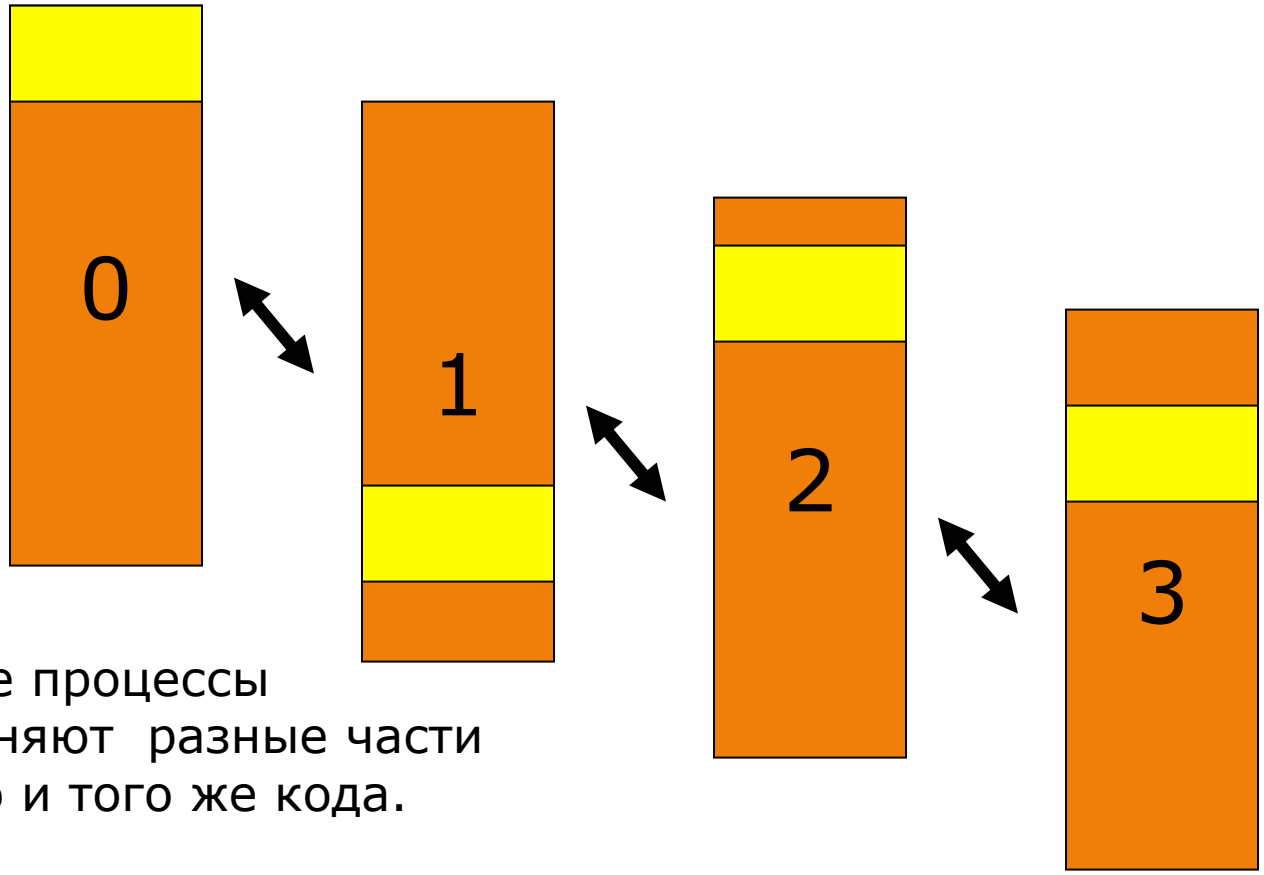
«Комплект поставки» MPI

- Библиотека (msmpi.lib, …).
- Средства компиляции и запуска приложения.

Microsoft HPC Pack SDK - MPI в ходит в Microsoft High Performance Computing Pack SDK от компании Microsoft

•Существуют несколько основных вариантов реализации MPI API:
MPICH
•MVAPICH
•OpenMPI
•Intel MPI

SPMD-модель



Разные процессы выполняют разные части одного и того же кода.

Номер процесса именуется **рангом** процесса (0 до p-1).

MPI “Hello, World”

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

int main(int argc, char* argv[])
{
        // <программный код без использования MPI функций>
        MPI_Init(&argc, &argv);
        // <программный код с использованием MPI функций>
        printf("Hello, World!\n");
        MPI_Finalize();
        // <программный код без использования MPI функций>
        return 0;
}
```

Функции инициализации и завершения работы

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

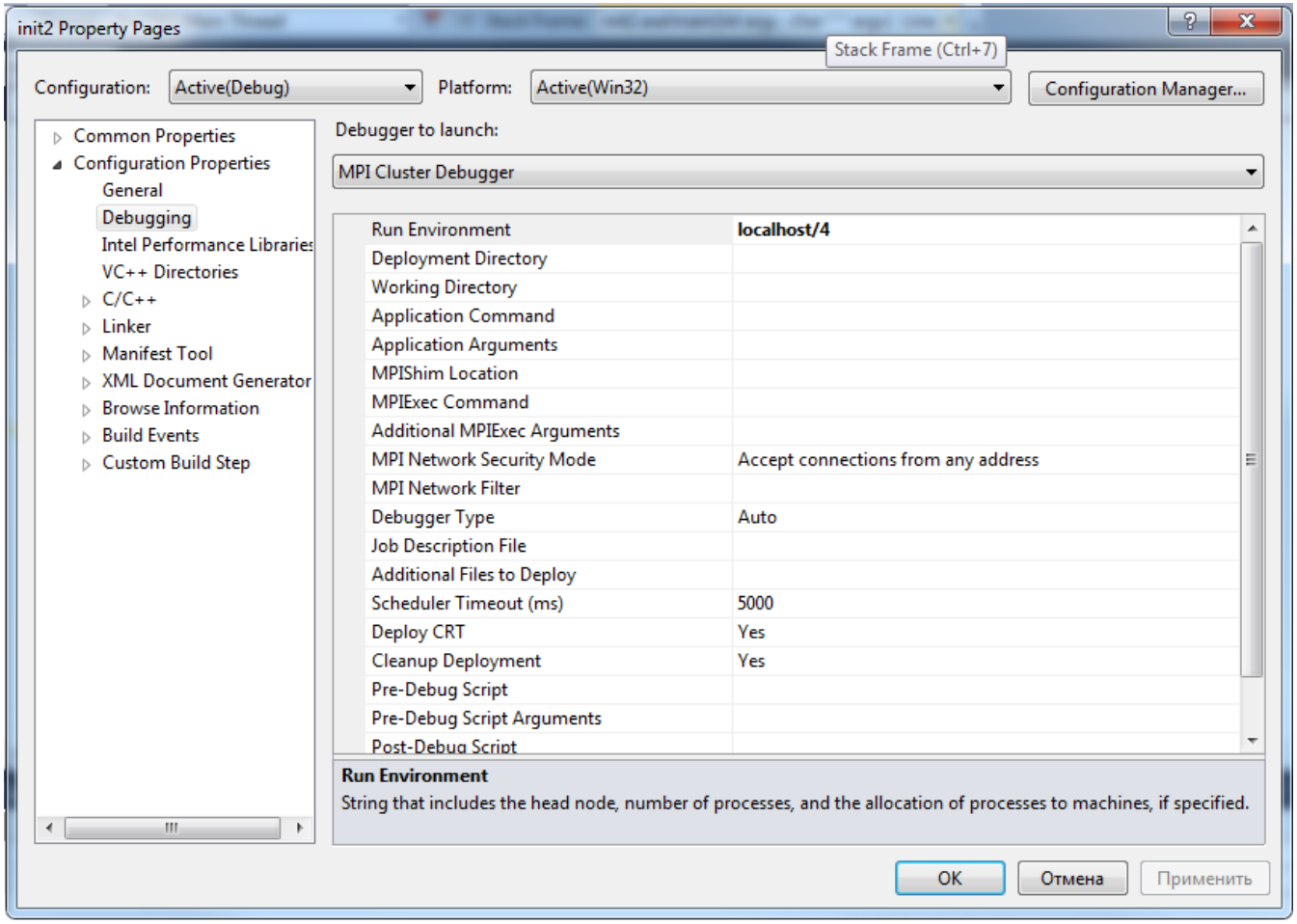
argc – указатель на счетчик аргументов командной строки

argv – указатель на список аргументов

int MPI_Finalize()

Все функции MPI возвращают код выполнения, значения которых описаны в заголовочном файле mpi.h

MPI Cluster Debugger



Группы. Коммуникаторы

Процессы параллельной программы объединяются в **группы**.

Коммуникатор в MPI - служебный *объект*, объединяющий *группу процессов* и ряд дополнительных параметров, используемых при выполнении *операций передачи данных*.

MPI_COMM_WORLD – глобальный коммутатор.

```
….
MPI_Init( &argc, &argv );
 //получение ранга текущего процесса в группе
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
//получение количества процессов в группе
 MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Finalize();
….
```

Функции определения ранга и числа процессов

int MPI_Comm_size (MPI_Comm comm, int* size)

comm - коммуникатор
size – число процессов

int MPI_Comm_rank(MPI_Comm comm, int* rank)

comm – коммуникатор
rank – ранг процесса

Определение времени выполнение MPI-программы

```
double MPI_Wtime(void)
```

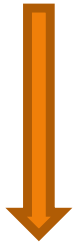
- получение времени текущего момента выполнения программы.

```
double t1, t2, dt;
t1 = MPI_Wtime();
 …
t2 = MPI_Wtime();
dt = t2 - t1;
```

```
double MPI_Wtick(void)
```

- определения текущего значения точности

Операции передачи данных



Парные
(point-to-point)

Коллективные
(collective)

Парные операции передачи данных

Различают следующие блокирующие режимы передачи данных:

- стандартный (Standard);
- синхронный (Synchronous);
- буферизованный (Buffered);
- режим передачи по готовности (Ready).

Стандартный режим передачи данных(1)

Стандартный режим отправки сообщений обеспечивает:

- блокировку процесс-отправителя на время выполнения функции;
- возможность повторного использования буфера после завершения функции;
- поддерживает следующие состояния сообщения: сообщение находится в процессе отправления, сообщение находится в процессе передачи, сообщение хранится в процессе-получателе или сообщение принято процессом-получателем при помощи функции MPI_Recv.

Стандартный режим передачи данных (1)

```
int MPI_Send( buf, count, datatype, dest, tag, comm )

void *buf;                                  /* in */
int  count, dest, tag;                  /* in */
MPI_Datatype datatype;    /* in */
MPI_Comm comm;                /* in */
```

buf - адрес начала буфера посылаемых данных
count - число пересылаемых объектов типа, соответствующего datatype
dest - номер процесса-приемника
tag - уникальный тэг, идентифицирующий сообщение
datatype - MPI-тип принимаемых данных
comm - коммуникатор

Стандартный режим передачи данных(2)

```
int MPI_Recv( buf, count, datatype, source, tag, comm, status )

void *buf;                  /* in */
int count, source, tag;     /* in */
MPI_Datatype datatype;      /* in */
MPI_Comm comm;              /* in */
MPI_Status *status;         /* out */
```

buf - адрес буфера для приема сообщения
count - максимальное число объектов типа datatype, которое может быть записано в буфер
source - номер процесса, от которого ожидается сообщение
tag - уникальный тэг, идентифицирующий сообщение
datatype - MPI-тип принимаемых данных
comm - коммуникатор
status - статус завершения

MPI_Status

```
typedef struct
{
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

MPI_SOURCE - ранг процесса-передатчика данных
MPI_TAG - тэг сообщения
MPI_ERROR - код ошибки

Базовые типы данных MPI

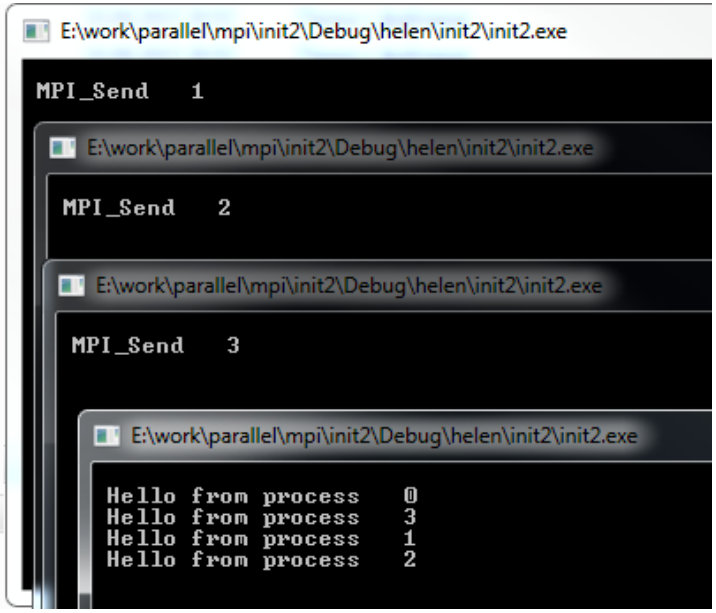
Таблица - Базовые типы данных MPI для алгоритмического языка C	
MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Пример простейшей пересылки

```
int ProcNum, ProcRank, RecvRank, i;
 MPI_Status Status;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if (ProcRank == 0)
 {
  // Действия, выполняемые только процессом с рангом 0
  printf ("\n Hello from process %3d", ProcRank);

  for ( i=1; i < ProcNum; i++ )
  {
   MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
   MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
   printf("\n Hello from process %3d", RecvRank);
  }
 } else {// Сообщение, отправляемое всеми процессами,
       // кроме процесса с рангом 0
  MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
  printf("\n MPI_Send %3d", ProcRank);
 }
 MPI_Finalize();
```



Виды точечных взаимодействий (1)

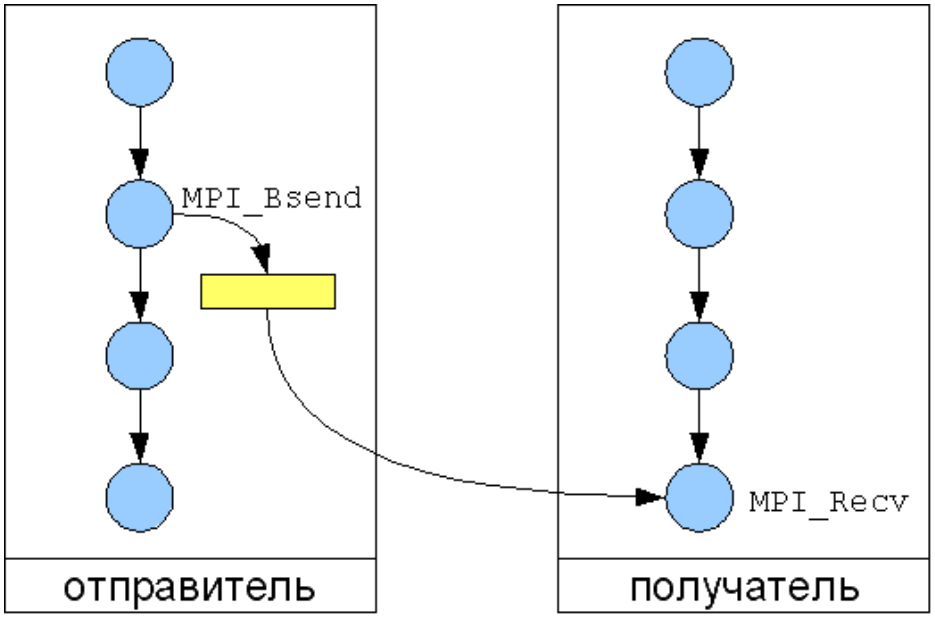
MPI_Send	**Стандартная пересылка** (Standard) функция возвращает управление тогда, когда исходный буфер можно освобождать (т.е. данные или скопированы в промежуточный или отправлены)
MPI_Bsend	**Буферизованная пересылка** (Buffered) функция возвращает управление тогда, когда данные скопированы в буфер, выделяемый пользователем

Виды точечных взаимодействий (2)

MPI_Ssend	Синхронная пересылка (Synchronous) функция возвращает управление тогда, когда процесс-приемник преступил к выполнению соответствующей операции приема
MPI_Rsend	**Режим передачи по готовности (Ready)** поведение функции не определено, если соответствующая операция приема не начала выполнения (для увеличения производительности)

Буферизованная пересылка

- Процесс-**отправитель** выделяет буфер и регистрирует его в системе.
- Функция `MPI_Bsend` помещает данные выделенный буфер, .



Функции работы с буфером обмена

```
//Описание буфера, используемого для буферизации сообщений,
//посылаемых в режиме буферизации.
int MPI_Buffer_attach(void *buffer /*in*/, int  size/*in*/)
buffer - адрес начала буфера
size - размер буфера в байтах

// Отключение буфера
int MPI_Buffer_detach(void *bufferptr /* out */,
                      int  *size     /* out */)
*bufferptr - адрес высвобожденного буфера
*size - размер высвобожденного пространства

функция MPI_Buffer_detach блокирует процесс до тех
пор, пока все данные не отправлены из буфера
```

Вычисление размера буфера

```
int MPI_Pack_size(int incount, MPI_Datatype
datatype, MPI_Comm comm, int *size)
```

Вычисляет размер памяти для хранения одного сообщения.

`MPI_BSEND_OVERHEAD` – дополнительный объем для хранения служебной информации (организация списка сообщений).

Размер буфера для хранения n одинаковых сообщений вычисляется по формуле:

`n x (размер_одного_сообщения + MPI_BSEND_OVERHEAD)`

Порядок организации буферизованных пересылок

1) Вычислить необходимый объем буфера (MPI_Pack_size).
2) Выделить память под буфер (malloc).
3) Зарегистрировать буфер в системе (MPI_Buffer_attach).
4) Выполнить пересылки.
5) Отменить регистрацию буфера (MPI_Buffer_dettach)
6) Освободить память, выделенную под буфер (free).

Особенности работы с буфером

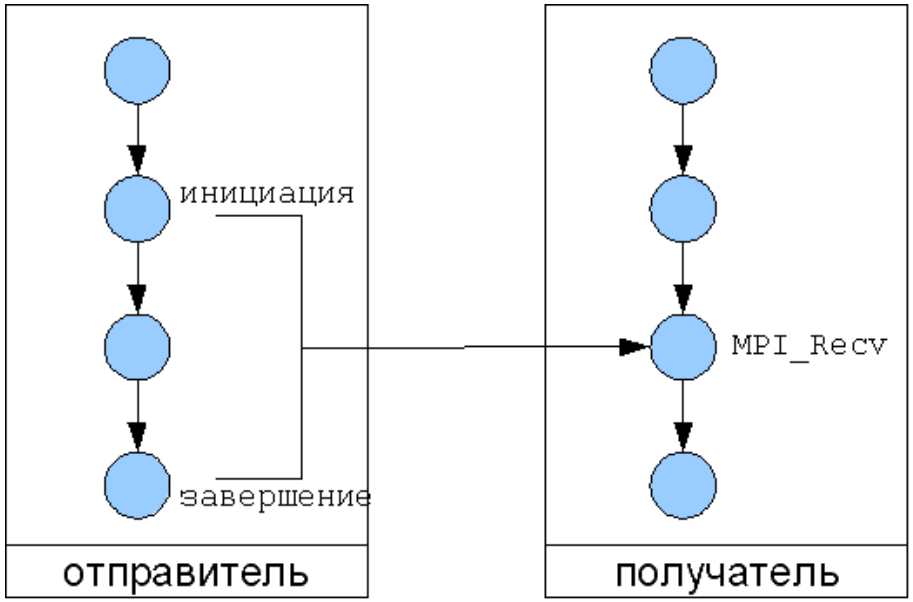
- Буфер всегда один.
- Для изменения размера буфера сначала следует отменить регистрацию, затем увеличить размер буфера и снова его зарегистрировать.
- Освобождать буфер следует только после того, как отменена регистрация.

Пример буферизованной пересылки

```
MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD,&msize)
blen = M * (msize + MPI_BSEND_OVERHEAD);
buf = malloc(blen);
MPI_Buffer_attach(buf, blen);
for(i = 0; i < M; i ++) {
  n = i;
  MPI_Bsend(&n, 1, MPI_INT, 1, i, MPI_COMM_WORLD);
}
MPI_Buffer_detach(&abuf, &ablen);
free(abuf);
```

Неблокирующие пересылки (1)

- Предназначены для перекрытия обменов и вычислений.
- Операция расщепляется на две: инициация и завершение.



Неблокирующие пересылки (2)

Инициация:

```
int MPI_Isend( buf, count, datatype, dest, tag, comm,
request)

MPI_Request       *request; /* out */
```

MPI_Ibsend(…), MPI_Issend(…), MPI_Irsend(…)

```
int MPI_Irecv( buf, count, datatype, source, tag, comm,
request )

MPI_Request       *request; /* out */
```

Неблокирующие пересылки (3)

Завершение:

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

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

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
      MPI_Status array_of_statuses[] )
```

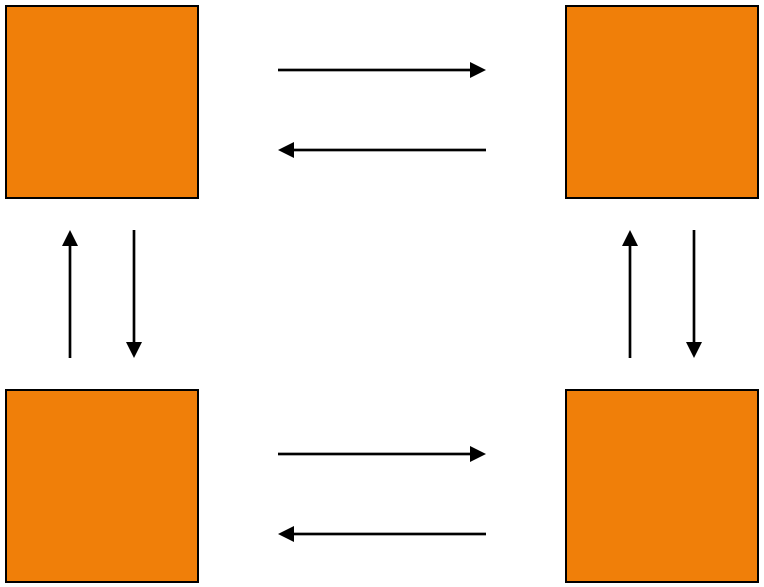
- завершились все операции

```
int MPI_Waitany(int count, MPI_Request array_of_requests[],
      int* index, MPI_Status *status )
```

- завершилась по крайней мере одна операция

L

Пример: кольцевой сдвиг данных



Пример: кольцевой сдвиг данных

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


void main (int argc, char* argv[])
{
  int numtasks, rank, next, prev, buf[2], tag1 = 1, tag2 = 2;
  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 == 0) ? (numtasks - 1) : (rank - 1);
  next = (rank == (numtasks - 1)) ? 0 : (rank + 1);
```

```
  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);

  printf("rank: %d, buf[0]: %d, buf[1]: %d\n",
       rank, buf[0], buf[1]);


  MPI_Finalize ();
}
```

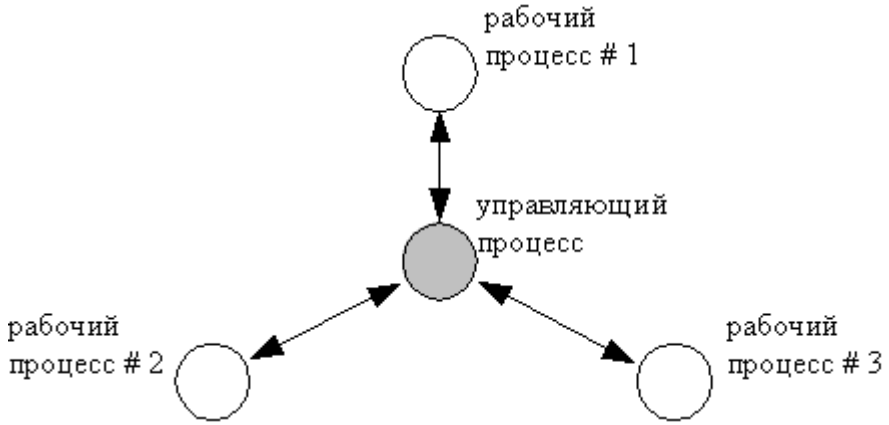
Прием по шаблону

В качестве параметров **source** и **tag** в функции **MPI_Recv** могут быть использованы константы

MPI_ANY_SOURCE и **MPI_ANY_TAG**

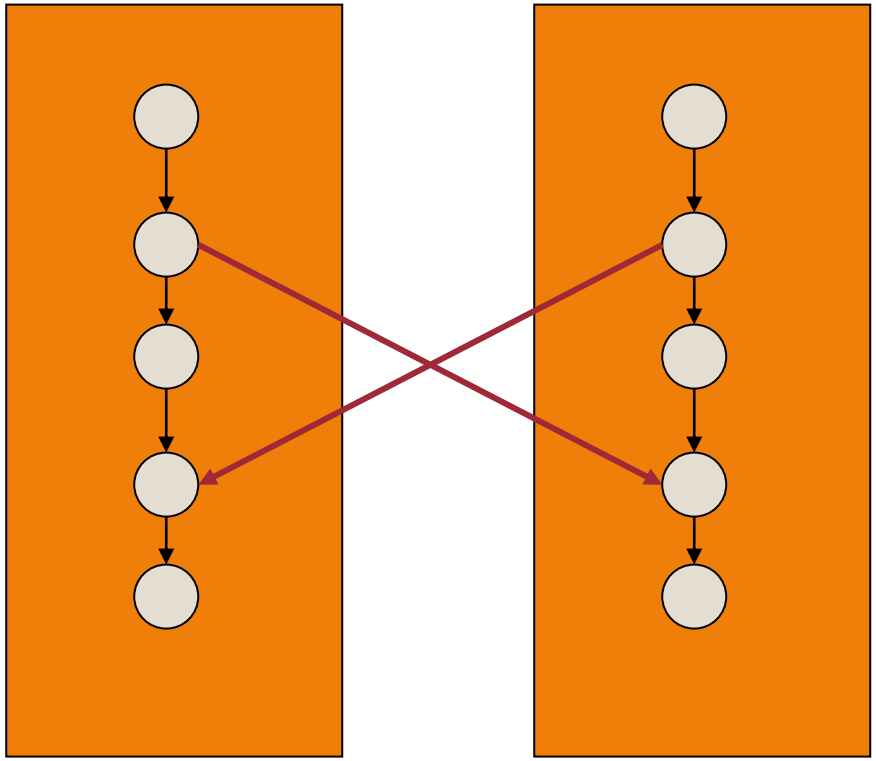
соответственно. Допускается прием от процесса с произвольным номером и/или сообщения с произвольным тэгом.

Стратегия управляющий-рабочие

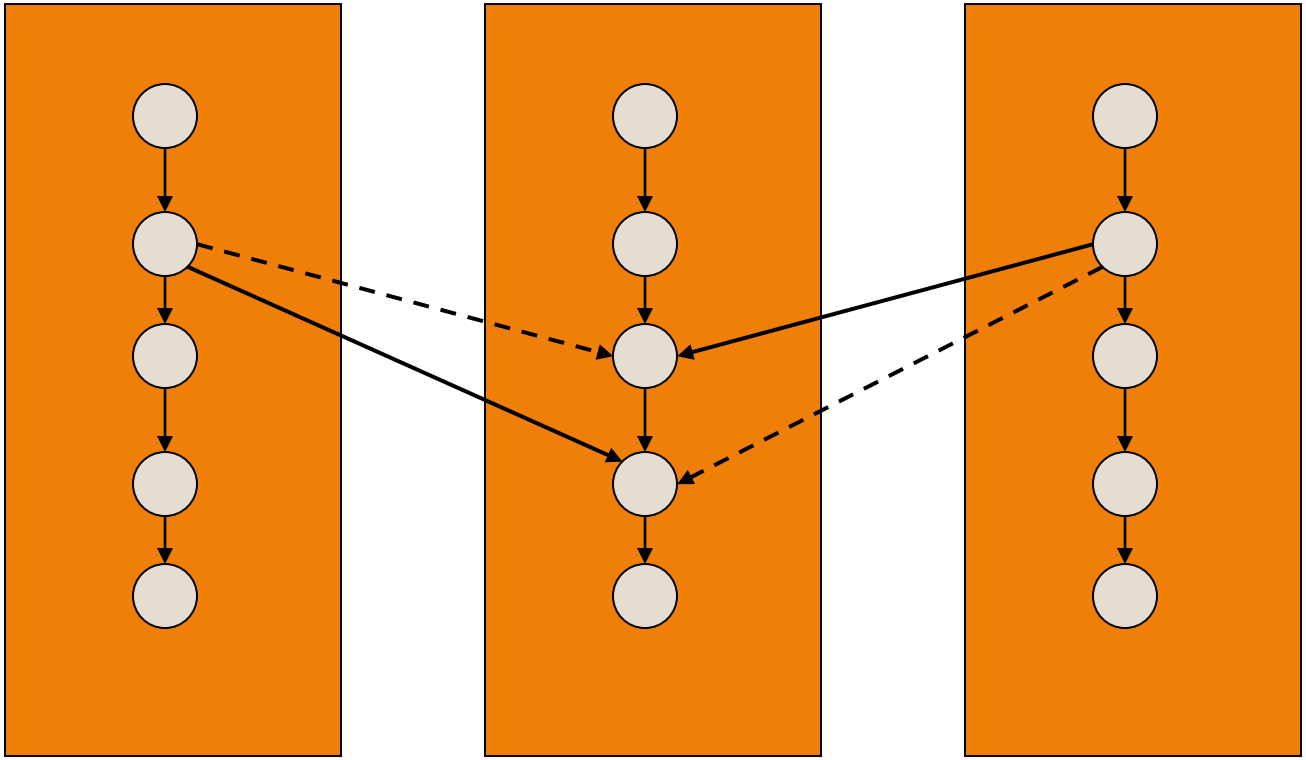


Deadlock

```
if(rank == 0) {
        // MPI_Send(… 1 …)
        MPI_Ssend(… 1 …)
        MPI_Recv(…1…)
} else {
        // MPI_Send(… 0 …)
        MPI_Ssend(… 0 …)
        MPI_Recv(…0…)
}
```



Недетерминизм за счет разницы в относительных скоростях процессов (race condition)



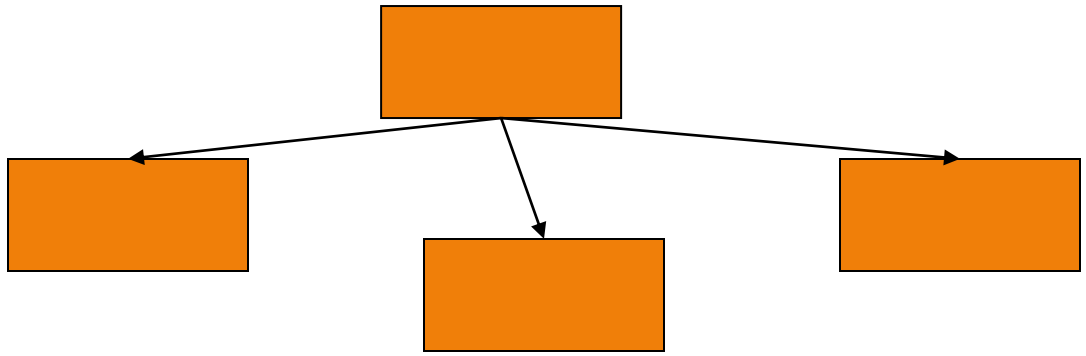
Коллективные взаимодействия процессов

MPI предоставляет ряд функций для коллективного взаимодействия процессов.

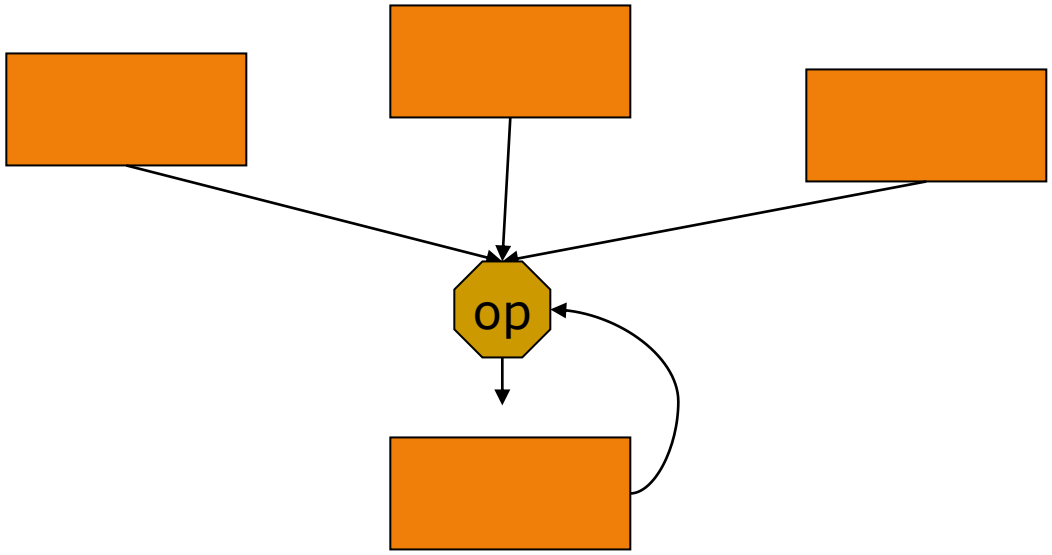
Эти функции называют коллективными, поскольку они должны вызываться на всех процессах, принадлежащих некоторому коммуникатору.

Коллективные взаимодействия процессов

int **MPI_Bcast** (buffer, count, datatype, root, comm)
void* buffer - начальный адрес буфера для передачи собщений
int count - число передаваемых элементов данных
MPI_Datatype datatype - тип передаваемых данных
int root - ранг процесса, посылающего данные
MPI_Comm comm - коммуникатор



```
int MPI_Reduce ( sendbuf, recvbuf, count,
      datatype, op, root, comm )
void  *sendbuf;     буфер операндов
void *recvbuf;      буфер приема
int  count;         число данных
MPI_Datatype datatype; тип данных
MPI_Op op;          операция
int root;           ранг процесса, содержащего
результат
MPI_Comm comm;      коммуникатор
```



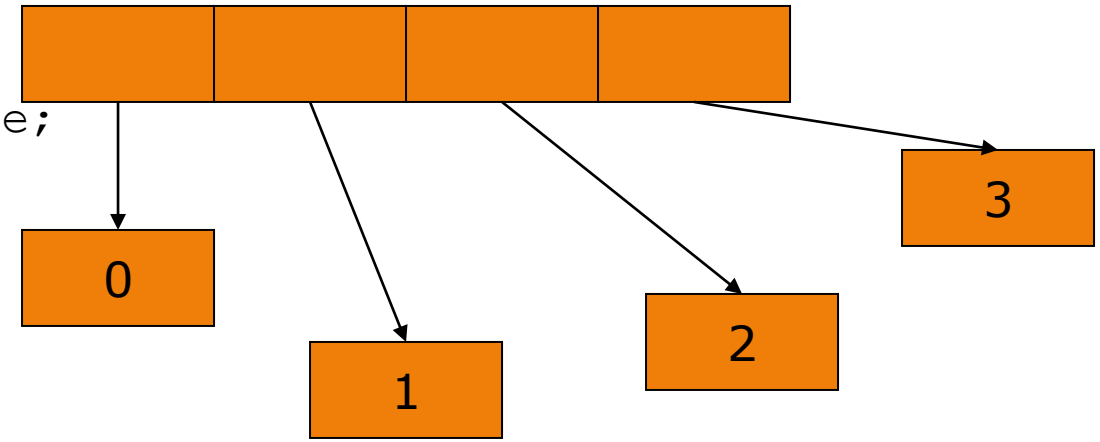
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое "и"
MPI_BAND	побитовое "и"
MPI_LOR	логическое "или"
MPI_BOR	побитовое "или"
MPI_LXOR	логическое исключающее "или"
MPI_BXOR	побитовое исключающее "или"

Коллективные операции передачи данных

```
int MPI_Scatter ( sendbuf, sendcnt, sendtype,
recvbuf, recvcnt, recvtype, root, comm )
```

- обобщенная передача данных от одного процесса всем процессам.

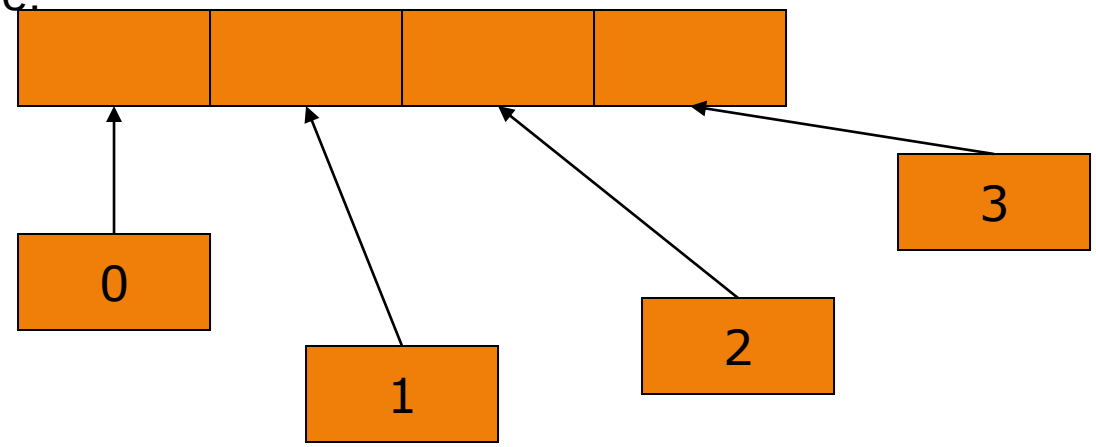
```
void *sendbuf;
int  sendcnt; // определяет количество элементов,
              // передаваемых на каждый процесс
MPI_Datatype  sendtype;
void *recvbuf;
int recvcnt;
MPI_Datatype recvtype;
int root;
MPI_Comm comm;
```



Коллективные операции передачи данных

//Обобщенная передача данных от всех процессов одному процессу

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



Коллективные операции передачи данных

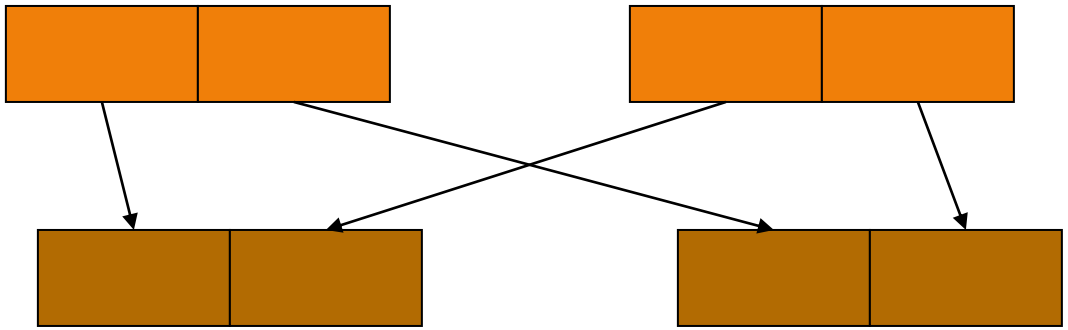
// для получения всех собираемых данных на каждом из процессов коммуникатора
int MPI_Allgather (sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype, comm);

//для получения результатов *редукции* данных на каждом из
// процессов *коммуникатора*
int MPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm)

Коллективные операции передачи данных

int MPI_Alltoall(sendbuf, sendcount,
sendtype, recvbuf, recvcnt, recvtype, comm)
- общая передача данных от всех процессов всем процессам

//параметры передаваемых сообщений
void *sendbuf; int sendcount; MPI_Datatype sendtype;
//параметры принимаемых сообщений
void *recvbuf; int recvcnt; MPI_Datatype recvtype;
//коммуникатор
MPI_Comm comm;



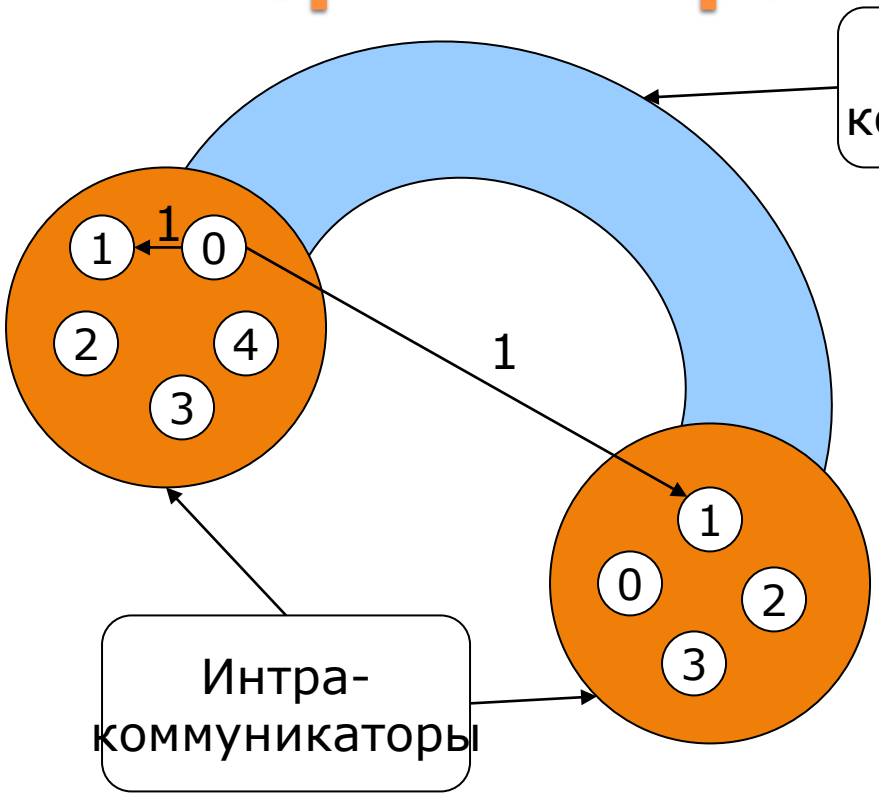
Синхронизация вычислений

Функция синхронизации процессов:

```
int MPI_Barrier ( comm ) ;

MPI_Comm comm;
```

Интер- и интра-коммуникаторы



Интра-коммуникаторы объединяют процессы из одной группы.

Интер-коммуникатор позволяет передавать данные между процессами из разных **интра**-коммуникаторов.

Интер-коммуникаторы не могут использоваться в коллективных взаимодействиях.

Коммуникатор – служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (контекст), используемых при выполнении операций передачи данных.

Создание коммуникаторов

Разбиение коммуникатора на несколько:

int **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)

comm – «старый коммуникатор»

color – селектор коммуникатора

key – задает порядок на создаваемых коммуникаторах

newcomm – создаваемый коммуникатор

color >= 0

для
color = MPI_UNDEFINED будет создан коммуникатор MPI_COMM_NULL

ранги во вновь создаваемых коммуникаторах присваиваются в соответствии с возрастанием key

Создание коммуникаторов

```
MPI_comm comm, newcomm;
int myid, color;
 . . . . . .
// определяем номер процесса
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid,
&newcomm);
```

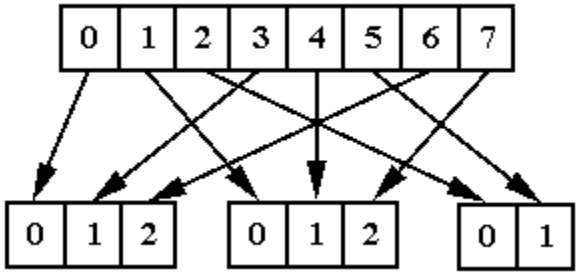


Рисунок - Разбиение группы из восьми процессов на три подгруппы.

Группы и коммуникаторы

Совокупности MPI-процессов образуют **группы**.

Понятие ранга процесса имеет смысл только по отношению к определенной группе или коммуникатору.

Каждому интра-коммуникатору соответствует группа процессов. По группе процессов можно построить коммуникатор.

Информационные функции для работы с группами

Определение размера группы:

int MPI_Group_size(MPI_Group group, int *size)

group – группа;

size – указатель на область памяти для записи информации о количестве процессов в группе;

Определение номера процесса, выполняющего вызов функции, в группе:

int MPI_Group_rank(MPI_Group group, int *rank)

group – группа;

rank – указатель на область памяти для сохранения номера процесса;

Информационные функции для работы с группами

Установление соответствия между номерами процессов в различных группах:

int MPI_Group_translate_ranks (
MPI_Group group1, int n, int *ranks1,
MPI_Group group2, int *ranks2)

group1 – первая группа;
n – число элементов массивов **ranks1** и **ranks2**;
ranks1 – массив номеров процессов в первой группе;
group2 – вторая группа;
ranks2 – массив для сохранения номеров процессов во второй группе;

Эта функция заполняет массив **ranks2** номерами процессов в группе **group2**, которые имеют номера, перечисленные в **ranks**1 в группе **group**1.

Информационные функции для работы с группами

Сравнение двух групп процессов:

int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

group1 – первая группа;

group2 – вторая группа;

result – указатель на область памяти для сохранения результата;

Если группа **group1** содержит те же процессы, что и группа **group2**, и порядок процессов в этих группах совпадает, группы считаются одинаковыми и по адресу **result** записывается константа **MPI_IDENT**, в противном случае результатом будет **MPI_UNEQUAL**.

Предопределенные группы

Педопределенные группы:
MPI_GROUP_EMPTY – «пустая» группа (не содержит процессов);
MPI_GROUP_NULL – «нулевая группа» (не соответствует никакой группе, аналог NULL.

Конструкторы и деструткоры групп

// уничтожение группы
int MPI_Group_free(MPI_Group* group)
group – идентификатор освобождаемой группы.

//Получение коммуникатора по группе
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
comm – коммуникатор;
group – указатель на область памяти для сохранения полученной группы;

Объединение двух групп:

int MPI_Group_union(MPI_Group gr1, MPI_Group g2, MPI_Group* gr3)

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции;

Набор процессов, входящих в gr3 получается объединением процессов, входящих в gr1 и gr2, причем элементы группы gr2, не вошедшие в gr1, следуют за элементами gr1.

Пересечение двух групп:

int MPI_Group_intersection(MPI_Group gr1, MPI_Group g2, MPI_Group* gr3)

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции;

Группа gr3 составлена из процессов, входящих как в gr1,так и в gr2, расположенных в том же порядке, что и в gr1.

Разность двух групп:

int MPI_Group_difference(MPI_Group gr1, MPI_Group g2, MPI_Group* gr3)

gr1 – первая группа;

gr2 – вторая группа;

gr3 – указатель на область для сохранения результата операции;

Группа gr3 составлена из процессов, входящих в gr1, но не входящих в gr2, расположенных в том же порядке, что и в gr1.

Переупорядочивание (с возможным удалением) процессов в существующей группе:

int MPI_Group_incl(MPI_Group* group, int n, int* ranks, MPI_Group* newgroup)

group – исходная группа;

n – число элементов в массиве ranks;

ranks – массив номеров процессов, из которых будет создана новая группа;

newgroup – указатель на область для сохранения результата операции;

Созданная группа newgroup содержит элементы группы group, перечисленные в массиве ranks: i-й процесс создаваемой группы newgroup совпадает с процессом, имеющим номер ranks[i] в группе group.

Удаление процессов из группы:

int MPI_Group_excl(MPI_Group* group, int n, int* ranks, MPI_Group* newgroup)

group – исходная группа;

n – число элементов в массиве ranks;

ranks – массив номеров удаляемых процессов;

newgroup – указатель на область для сохранения результата операции;

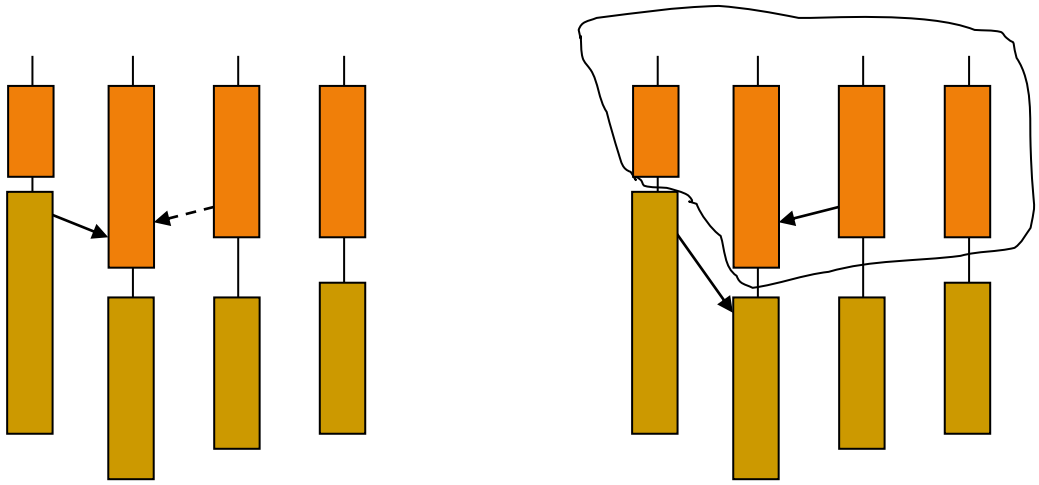
В результате выполнения этой операции создается новая группа newgroup, получаемая удалением из исходной группы процессов с номерами, перечисленными в массиве ranks.

Дублирование коммуникатора

Получение дубликата коммуникатора:

int **MPI_Comm_dup**(MPI_Comm comm, MPI_Comm* newcomm)

Используется для того, чтобы снабдить библиотечную функцию новым коммуникатором, совпадающим по характеристикам со старым, но вместе с тем, создающим новый контекст для коммуникаций. Цель: исключить проблемы, связанные с «перемешиванием коммуникаций».



Создание коммуникатора по группе

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

Требования:

1. Конструктор вызывается на всех процессах коммуникатора **comm**;
2. **group** – подгруппа группы коммуникатора comm, одинакова на всех процессах.

Результат:

newcomm – на процессах, вошедших в **group**, новый коммуникатор, на остальных – **MPI_COMM_NULL**

Удаление коммуникатора

Освобождение коммуникатора:

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

При освобождении коммуникатора все незавершенные операции будут завершены, только после этого коммуникатор будет удален физически.

Выводы

Рассмотрели:
- общую информацию о MPI;
- функции инициализации и завершения работы;
- понятие групп и коммуникаторов;
- операции передачи данных;
- определение времени выполнения MPI программы;
- функции обмена точка-точка;
- базовые типы данных MPI;
- виды точечных взаимодействий;
- прием по шаблону;
- потенциальные ошибки при работе с MPI;
- коллективные взаимодействия процессов;
- коллективные операции передачи данных;
- синхронизация вычислений;
- интер- и интра-коммуникаторы;
- функции работы с коммуникаторами (создание, разбиение, дублирование, удаление);
- информационные функции для работы с группами.