

Средства и системы параллельного программирования

Лектор: доцент Н.Н.Попова,

Лекция 8.

25 октября 2021 г.

Тема

1. Коллективные передачи в MPI (продолжение)
2. Виртуальные топологии

Коллективные передачи

- Передача сообщений между группой процессов
- Вызываются ВСЕМИ процессами в коммутаторе

Предыдущая
лекция

Классификация коллективных передач (1)

- **One-To-All**

Один процесс определяет результат. Все процессы получают этот результат..

- `MPI_Bcast`
- `MPI_Scatter`, `MPI_Scatterv`

- **All-To-One**

Все процессы участвуют в создании результата. Один процесс получает результат..

- `MPI_Gather`, `MPI_Gatherv`
- `MPI_Reduce`

Классификация коллективных передач (2)

- **All-To-All**

Все процессы участвуют в создании результата. Все процессы получают результат.

- MPI_Allgather, MPI_Allgatherv
- MPI_Alltoall, MPI_Alltoallv
- MPI_Allreduce, MPI_Reduce_scatter

- **Другие**

Коллективные операции, не попадающие в вышеотмеченные классы.

- MPI_Scan
- MPI_Barrier

Характеристики коллективных передач

- Коллективные операции не являются помехой операциям типа «точка-точка» и наоборот
- Все процессы коммуникатора должны вызывать коллективную операцию
- Синхронизация не гарантируется (за исключением барьера).
Завершение операции – локально в процессе
- Нет тэгов
- Принимающий буфер должен точно соответствовать размеру отсылаемого буфера
- Асинхронные коллективные передачи - в MPI-3

Функции коллективных передач

Collective Communication Routines		
<u>MPI Allgather</u>	<u>MPI Allgatherv</u>	<u>MPI Allreduce</u>
<u>MPI Alltoall</u>	<u>MPI Alltoallv</u>	<u>MPI Barrier</u>
<u>MPI Bcast</u>	<u>MPI Gather</u>	<u>MPI Gatherv</u>
<u>MPI Op create</u>	<u>MPI Op free</u>	<u>MPI Reduce</u>
<u>MPI Reduce scatter</u>	<u>MPI Scan</u>	<u>MPI Scatter</u>
<u>MPI Scatterv</u>		

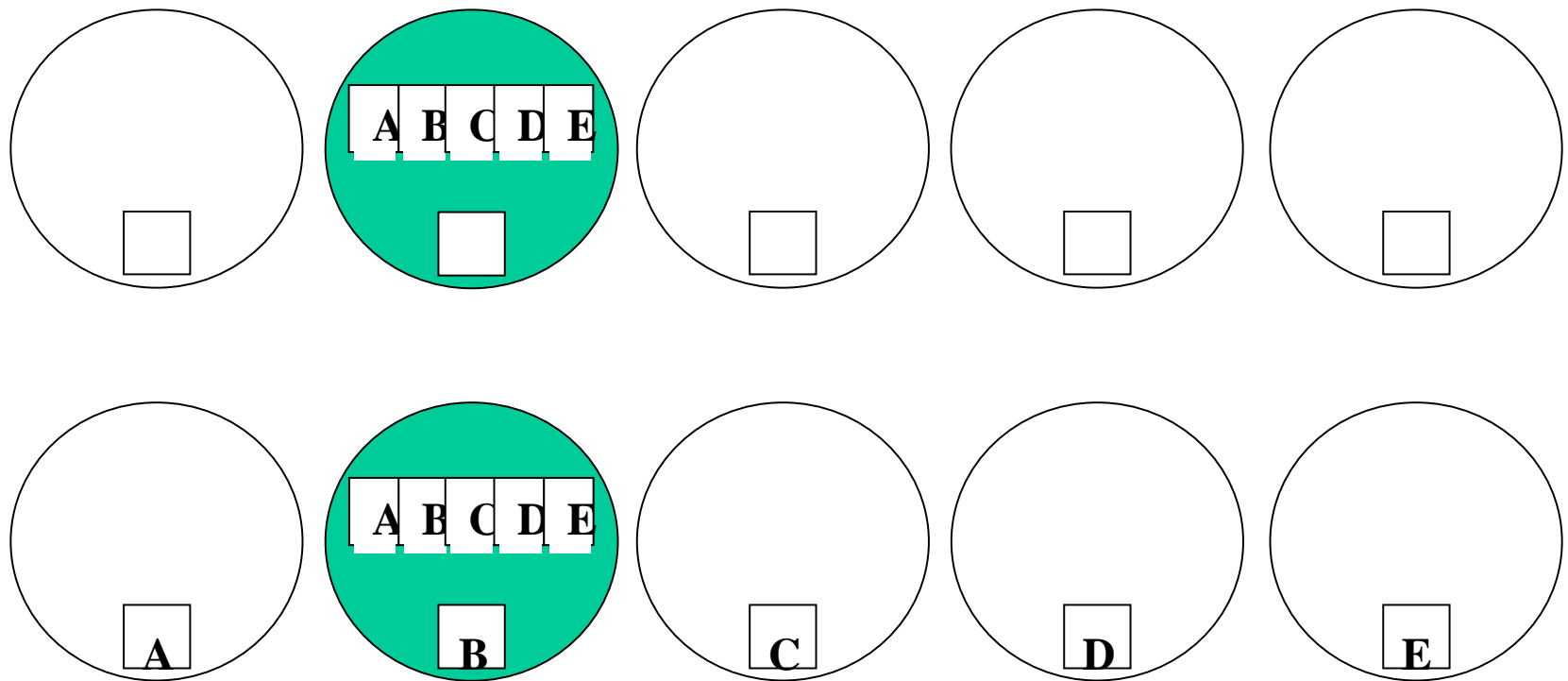
Функция Scatter рассылки блоков данных

- One-to-all передачи: блоки данных одного размера из одного процесса рассылаются всем процессам коммутатора (в порядке их номеров)

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

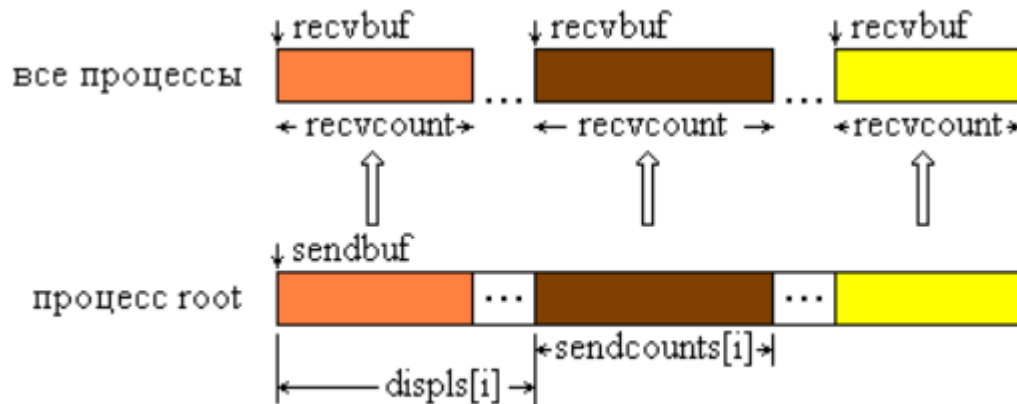
- *sendcount* – число элементов, посланных каждому процессу, **не общее число отосланных элементов**;
- send параметры (sendbuf, sendcount, sendtype) имеют смысл только для процесса root

Scatter – графическая иллюстрация

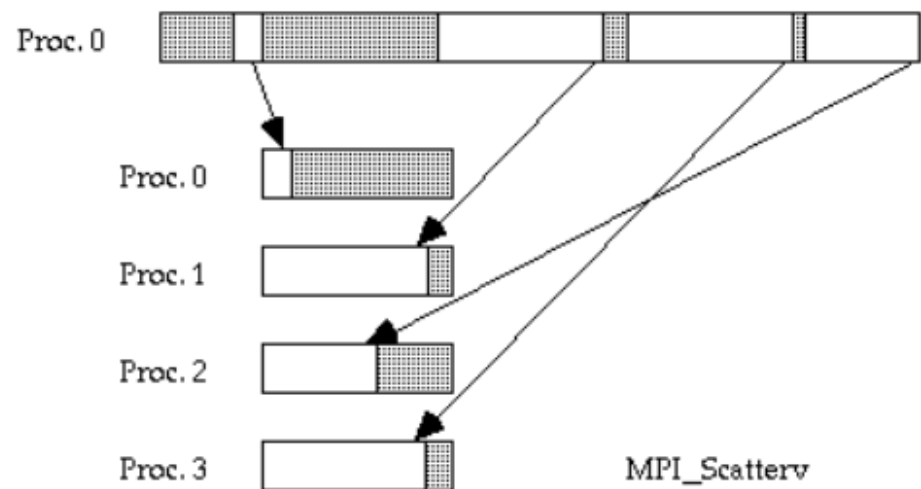
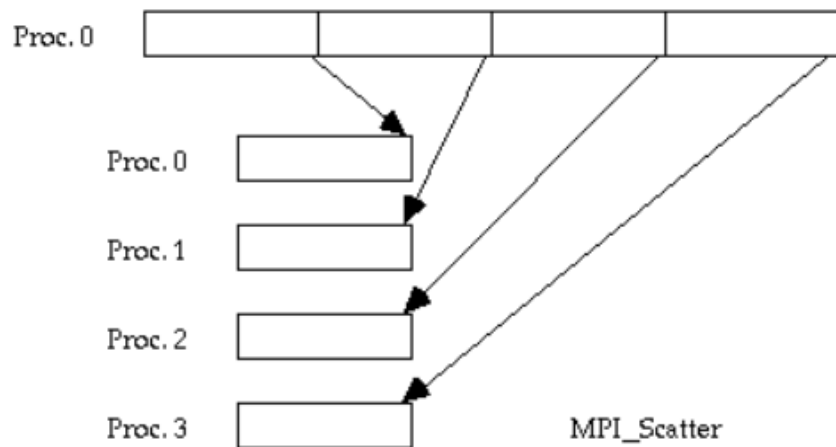


Функция Scatterv рассылки блоков разной длины

```
int MPI_Scatterv(void* sendbuf, int *sendcounts,  
int *displs, MPI_Datatype sendtype, void* recvbuf,  
int recvcount,  
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



Сравнение: MPI_Scatter & MPI_Scatterv



Пример использования MPI_Scatterv

Рассылка массива в случае, если размер массива N НЕ ДЕЛИТСЯ нацело на число процессов size

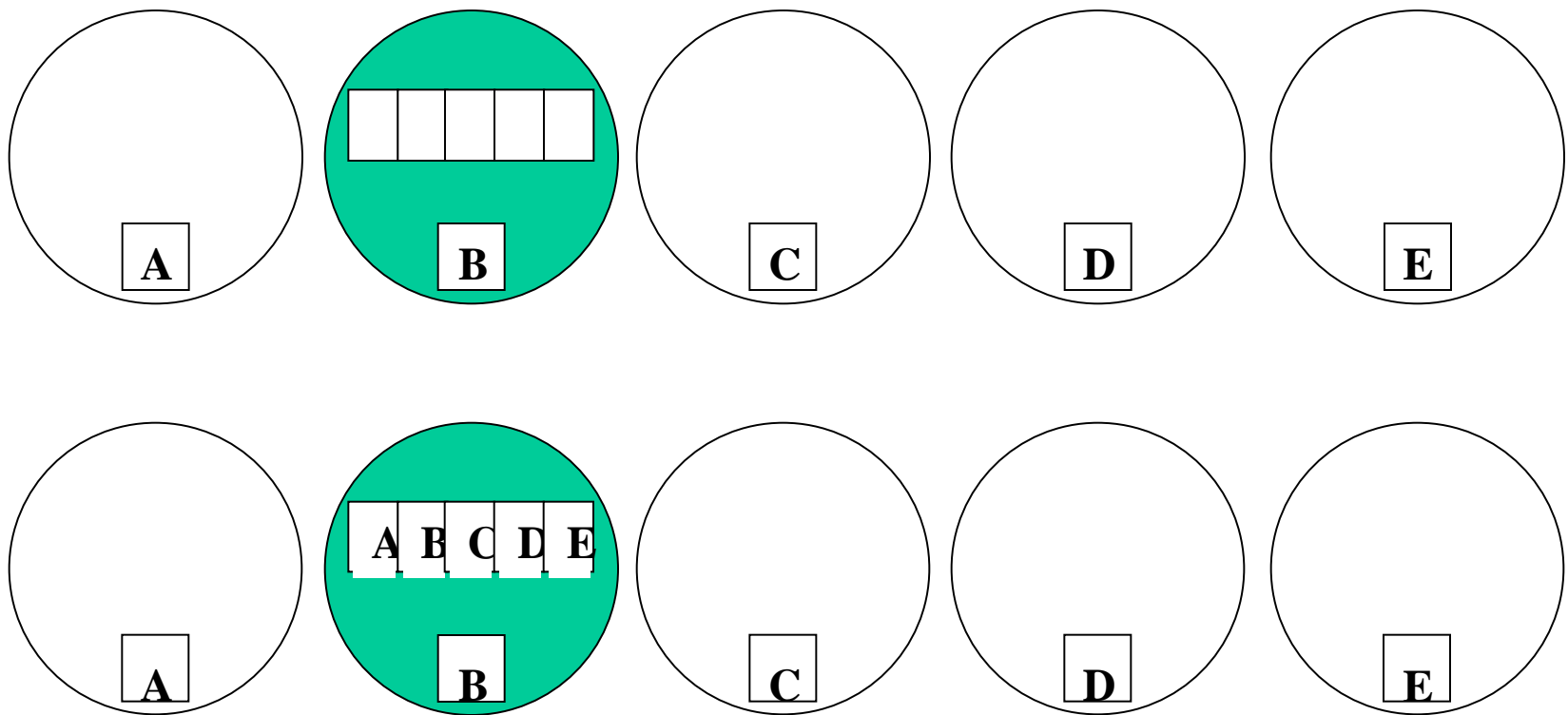
```
nmin = N/size;
nextra = N%size;
k = 0;
for (i=0; i<size; i++) {
    if (i<nextra) sendcounts[i] = nmin+1;
    else sendcounts[i] = nmin; displs[i] = k; k = k+sendcounts[i]; }
// need to set recvcnt also ...
MPI_Scatterv( sendbuf, sendcounts, displs, ...
```

Функция Gather сбора данных

- All-to-one передачи: блоки данных одинакового размера собираются процессом root
- Сбор данных выполняется в порядке номеров процессов
- Длина блоков предполагается одинаковой, т.е. данные, посланные процессом i из своего буфера `sendbuf`, помещаются в i -ю порцию буфера `recvbuf` процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

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

Gather – графическая иллюстрация



Функция `Gatherv` сбора блоков данных разной длины

- Сбор данных разного размера в процессе `root` в порядке номеров процессов
- Длина блоков предполагается разной для процессов, т.е. данные, посланные процессом `i` из своего буфера `sendbuf`, помещаются в `i`-ю порцию буфера `recvbuf` процесса `root`. Начало `i`-ой порции определяется смещением, указанным в массиве `displs`. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

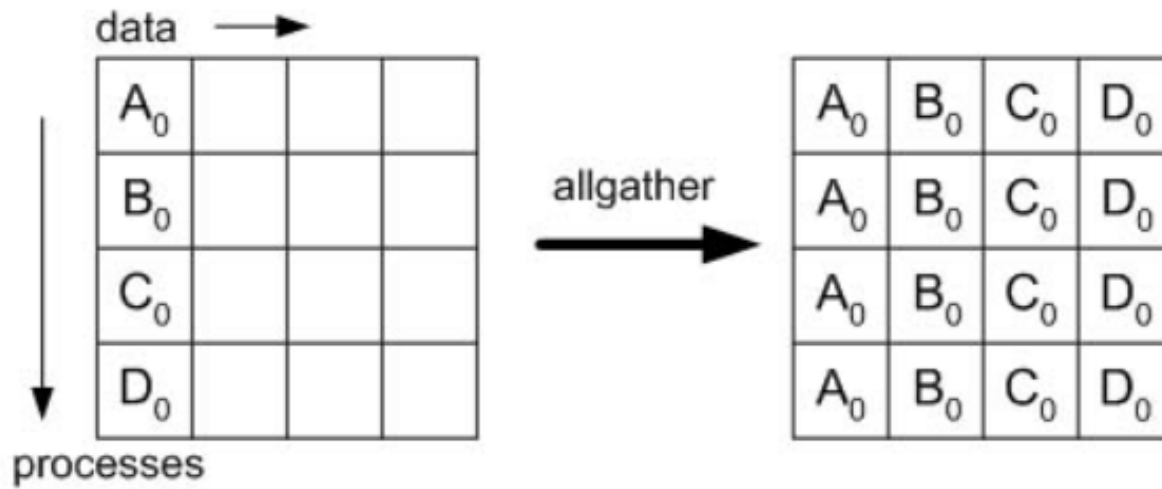
```
int MPI_Gatherv (void *sendbuf, int sendcount,  
MPI_Datatype sendtype, void *recvbuf, int *recvcounts,  
int *displs, MPI_Datatype recvtype, int root, MPI_Comm  
comm)
```

MPI_Allgather

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

```
int MPI_Allgatherv(const void* sbuf, int scounts, MPI_Datatype stype,  
void* rbuf, const int rcounts[], const int displs[], MPI_Datatype rtype,  
MPI_Comm comm)
```


Иллюстрация MPI_Allgather



In Place

Позволяет избежать локальное копирование, например из **send** буфер в **receive** буфер.

В качестве одного из буферов (всегда меньшего, если они различаются по размеру) можно использовать специальное значение `MPI_IN_PLACE`. Тип данных и количество передаваемых элементов этого буфера игнорируются.

Для `MPI_Gather` и `MPI_Reduce` это значение используется для **send** buffer.

Для `MPI_Scatter` это значение используется для **receive** buffer.

Пример использования In Place

```
Int value = ...;
if (rank == root) {
    recv_buf[root] = value;
    MPI_Gather(MPI_IN_PLACE, 1, MPI_INT,
              recv_buf, 1, MPI_INT,
              root, comm);}
else {
    MPI_Gather(&value, 1, MPI_INT,
              recv_buf, 1, MPI_INT,
              root, comm);
}
```

Требуется различать root и не-root

MPI_ALLTOALL

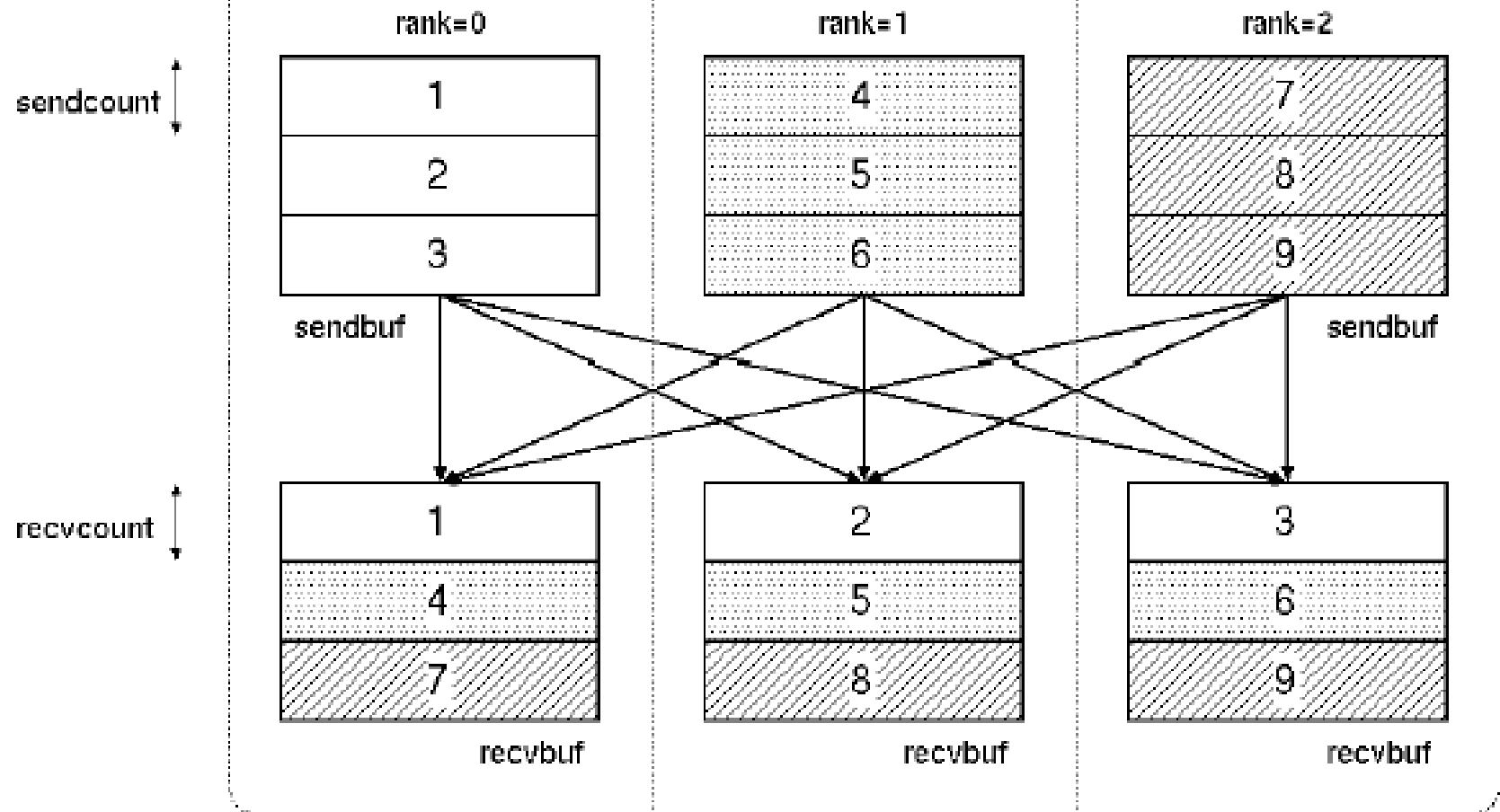
```
int MPI_Alltoall(  
void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* recvbuf, int recvcount, MPI_Datatype recvtype,  
MPI_Comm comm);
```

Описание:

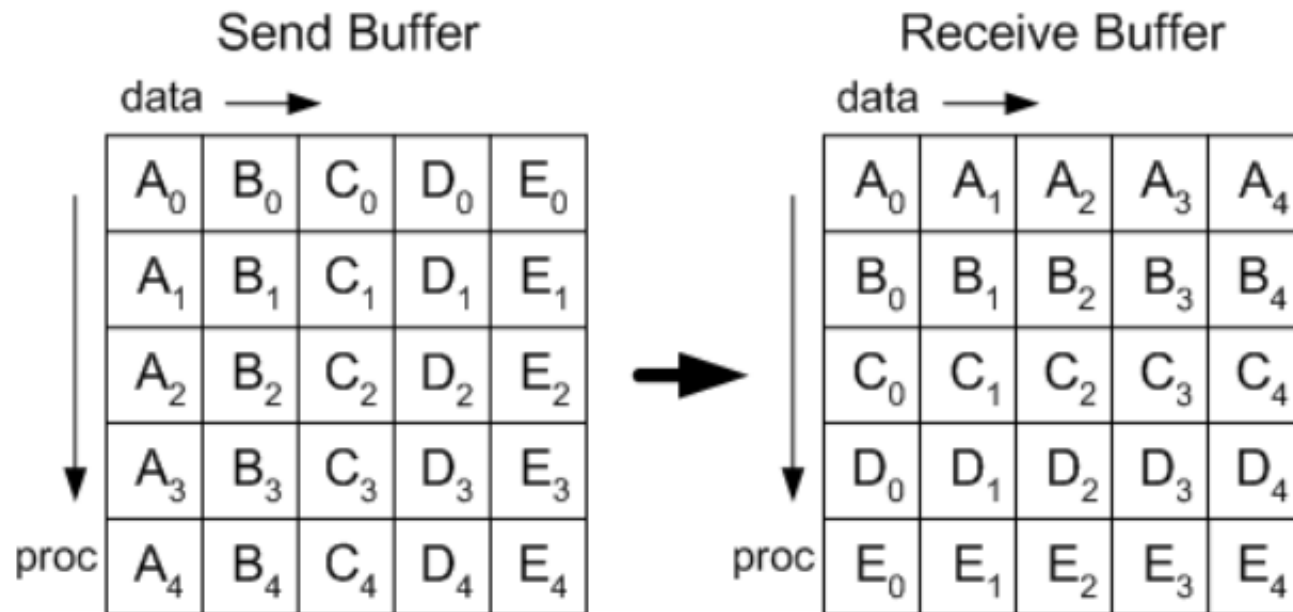
- Рассылка сообщений от каждого процесса каждому
- j-ый блок данных из процесса i принимается j-ым процессом и размещается в i-ом блоке буфера recvbuf

MPI ALLTOALL

comm



MPI_ALLTOALL

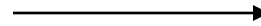


Детали реализации вариантов

- MPI_Reduce возвращает результат в один процесс;
- MPI_Allreduce возвращает результат всем процессам;
- MPI_Reduce_scatter_block раздает результат (вектор) по всем процессам, блоками одинакового размера.
- MPI_Reduce_scatter раздает вектор результата блоками разной длины
- MPI_Scan префиксная редукция данных, распределенных в группе.

processes	data		
	a0	b0	c0
	a1	b1	c1
	a2	b2	c2

reduce



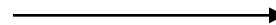
processes	data		
	a	b	c

Note:

$a = \text{sum}(a_i, \text{all } i)$
 $b = \text{sum}(b_i, \text{all } i)$
 $c = \text{sum}(c_i, \text{all } i)$
 (here $i=0..2$)

processes	data		
	a0	b0	c0
	a1	b1	c1
	a2	b2	c2

allreduce



processes	data		
	a	b	c
	a	b	c
	a	b	c

processes	data		
	a0	b0	c0
	a1	b1	c1
	a2	b2	c2

reduce-scatter



processes	data		
	a		
	b		
	c		

processes	data		
	a0	b0	c0
	a1	b1	c1
	a2	b2	c2

scan



processes	data		
	a0	b0	c0
	a0+a1	b0+b1	c0+c1
	a	b	c

Асинхронные коллективные передачи

- Введены в MPI-3
- Семантика такая же, как и для синхронных аналогичных функций
- В список параметров добавляется параметр request :

```
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int  
root, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Ireduce(const void *sendbuf, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,  
MPI_Request *request)
```

...

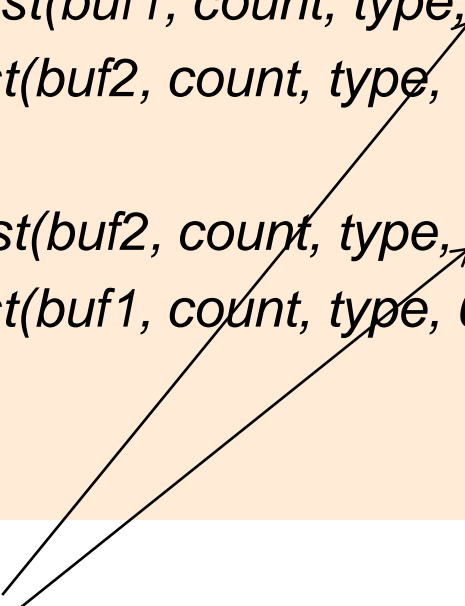
Ошибки. Пример 1.

```
switch(rank) {  
    case 0: MPI_Bcast(buf1, count, type, 0, comm);  
            MPI_Bcast(buf2, count, type, 1, comm);  
            break;  
    case 1: MPI_Bcast(buf2, count, type, 1, comm);  
            MPI_Bcast(buf1, count, type, 0, comm);  
            break;  
}
```

В чем ошибка?

Ошибки. Пример 1.

```
switch(rank) {  
    case 0: MPI_Bcast(buf1, count, type, 0, comm);  
            MPI_Bcast(buf2, count, type, 1, comm);  
            break;  
    case 1: MPI_Bcast(buf2, count, type, 1, comm);  
            MPI_Bcast(buf1, count, type, 0, comm);  
            break;  
}
```



Коллективные операции должны выполняться в одном и том же порядке всеми участниками коммуникационной группы. .

Ошибки. Пример 2.

```
switch(rank) {  
  case 0: MPI_Bcast(buf1, count, type, 0, comm);  
          MPI_Send(buf2, count, type, 1, tag, comm);  
          break;  
  case 1: MPI_Recv(buf2, count, type, 0, tag, comm, status);  
          MPI_Bcast(buf1, count, type, 0, comm);  
          break;  
}
```

В чем ошибка?

Ошибки. Пример 2.

```
switch(rank) {  
  case 0: MPI_Bcast(buf1, count, type, 0, comm);  
          MPI_Send(buf2, count, type, 1, tag, comm);  
          break;  
  case 1: MPI_Recv(buf2, count, type, 0, tag, comm, status);  
          MPI_Bcast(buf1, count, type, 0, comm);  
          break;  
}
```

Относительный порядок выполнения коллективных операций и операций «точка-точка» должен быть таким, чтобы даже в случае синхронизации коллективных операций и операций «точка-точка» не возникало тупиковой ситуации.

Тема

- Виртуальные топологии

Понятие коммуникатора MPI

- Коммуникатор - управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом

Группы и коммутаторы

■ Группа:

- Упорядоченное множество процессов
- Каждый процесс в группе имеет уникальный номер
- Процесс может принадлежать нескольким группам
 - rank всегда относителен группы

■ Коммутаторы:

- Все обмены сообщений всегда проходят в рамках коммутатора
 - С точки зрения программирования группы и коммутаторы эквивалентны
 - Коммутаторы – глобальные объекты.
- Группы и коммутаторы – динамические объекты, должны создаваться и уничтожаться в процессе работы программы

Специальные типы MPI

- MPI_Comm
- MPI_group

Создание новых коммуникаторов

2 способа создания новых коммуникаторов:

- Использовать функции для работы с группами и коммуникаторами (создать новую группу процессов и по новой группе создать коммуникатор, разделить коммуникатор и т.п.)
- Использовать встроенные в MPI виртуальные топологии

Предопределенные коммуникаторы

- MPI-1 поддерживает 3 предопределенных коммуникатора:
 - MPI_COMM_WORLD
 - MPI_COMM_NULL
 - MPI_COMM_SELF
- Только MPI_COMM_WORLD используется для передачи сообщений
- Нужны для реализации ряда функций MPI

Использование MPI_COMM_WORLD

- Содержит все доступные на момент старта программы процессы
- Обеспечивает начальное коммуникационное пространство
- Простые программы часто используют только MPI_COMM_WORLD
- Сложные программы дублируют и производят действия с копиями MPI_COMM_WORLD

Использование MPI_COMM_NULL

- Нет реализации такого коммуникатора
- Не может быть использован как параметр ни в одной из функций
- Может использоваться как начальное значение коммуникатора
- Возвращается в качестве результата в некоторых функциях
- Возвращается как значение после операции освобождения коммуникатора

Использование MPI_COMM_SELF

- Содержит только локальный процесс
- Обычно не используется для передачи сообщений
- Содержит информацию:
 - кэшированные атрибуты, соответствующие процессу
 - предоставление единственного входа для определенных ВЫЗОВОВ

Виртуальные топологии

- Удобный способ именования процессов
- Упрощение написания параллельных программ
- Оптимизация передач
- Возможность выбора топологии, соответствующей логической структуре задачи

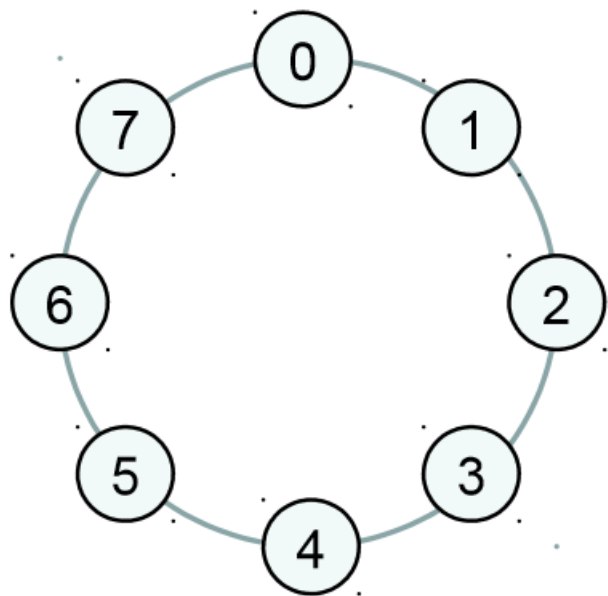
Как использовать виртуальные ТОПОЛОГИИ

- Создание топологии – новый коммуникатор
- MPI обеспечивает “mapping functions”
- Mapping функции вычисляют ранг процессов, базирясь на топологии

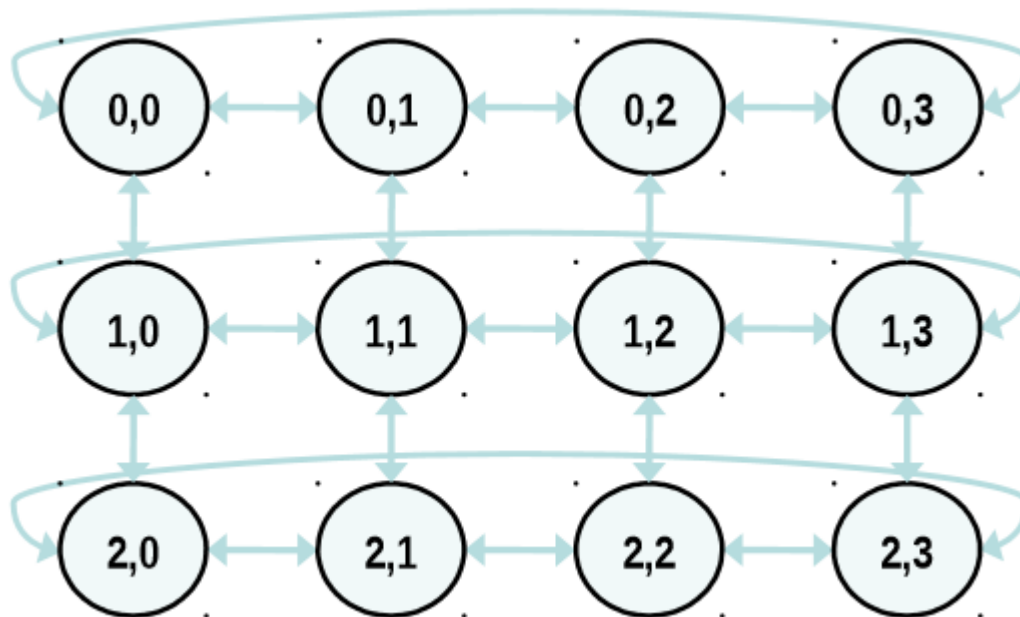
Типы виртуальных топологий

- Декартовская (многомерная решетка)
- Графовая

Пример декартовых топологий

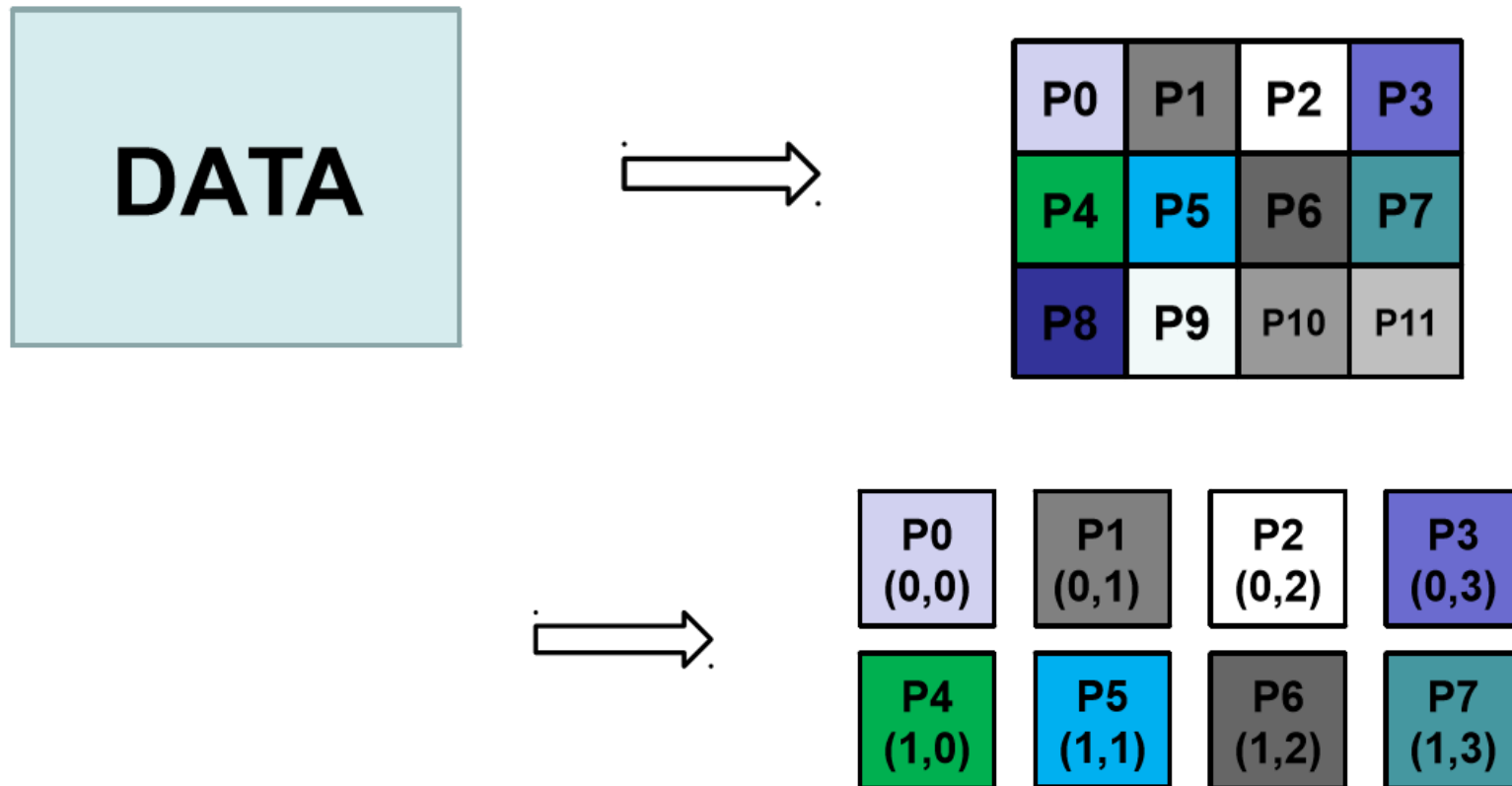


1D



2D

Отображение данных на виртуальную топологию

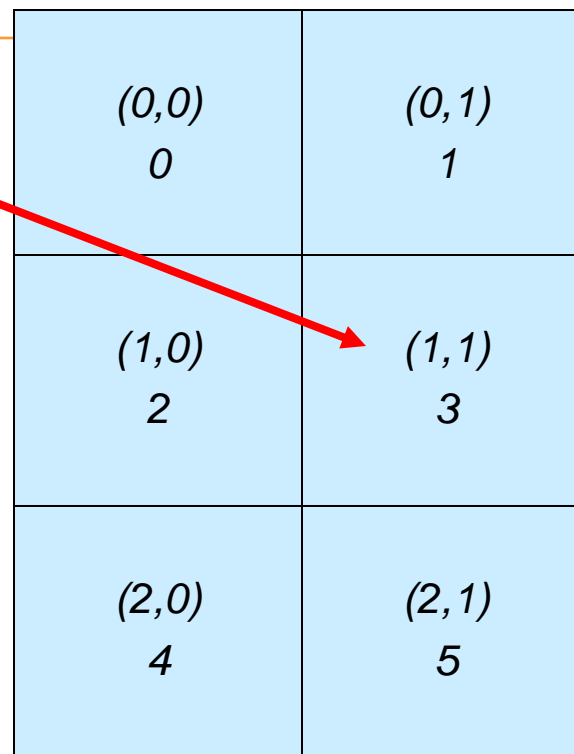


Основные функции декартовых топологий

- MPI_CART_CREATE
- MPI_DIMS_CREATE
- MPI_CART_COORDS
- MPI_CART_RANK
- MPI_CART_SUB
- MPI_CARTDIM_GET
- MPI_CART_GET
- MPI_CART_SHIFT

2D решетка

- Отображает линейно упорядоченный массив в 2-мерную решетку (2D Cartesian topology),
- Пример: номер 3 адресуется координатами (1,1).
- Каждая клетка представляет элемент 3x2 матрицы.
- Нумерация начинается с 0.
- Нумерация построчная.



$(0,0)$ 0	$(0,1)$ 1
$(1,0)$ 2	$(1,1)$ 3
$(2,0)$ 4	$(2,1)$ 5

Создание виртуальной топологии решетка

```
int MPI_Cart_create (MPI_Comm  comm_old,  
                    int ndims,  
                    int *dims,  
                    int *periods,  
                    int reorder,  
                    MPI_Comm *comm_cart)
```

Параметры

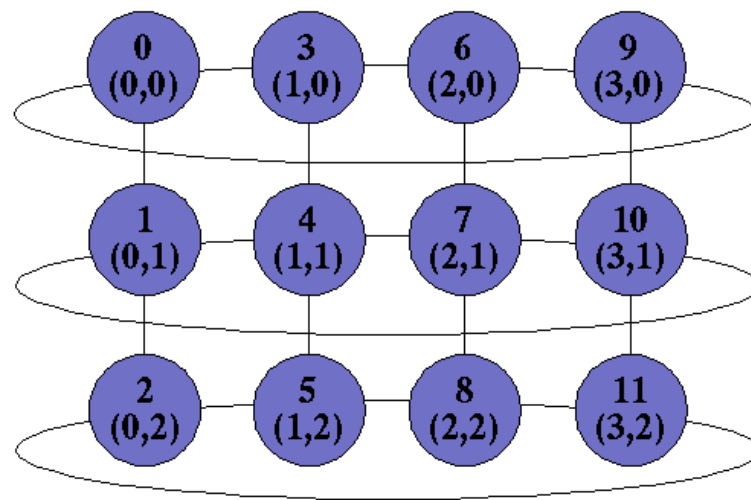
<i>comm_old</i>	старый коммуникатор
<i>ndims</i>	размерность
<i>Periods</i>	логический массив, указывающий на циклическое замыкание: TRUE/FALSE => циклическое замыкание на границе
<i>reorder</i>	возможная перенумерация процессов в новом коммуникаторе
<i>comm_cart</i>	новый коммуникатор

Пример виртуальной топологии решетки

```
MPI_Comm vu;  
int dim[2], period[2], reorder;
```

```
dim[0]=4; dim[1]=3;  
period[0]=TRUE; period[1]=FALSE;  
reorder=TRUE;
```

```
MPI_Cart_create(MPI_COMM_WORLD,2,  
               dim,period,reorder,&vu);
```



Координаты процесса в виртуальной решетке

```
int MPI_Cart_coords (  
    MPI_Comm comm,           /* Коммуникатор */  
    int rank,                /* Ранг процесса */  
    int numb_of_dims,        /* Размер решетки */  
    int coords[]             /* координаты процесса в решетке */  
)
```

MPI_CART_RANK

*int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)*

Перевод логических координат процесса в решетке в ранг процесса.

- Если *i*-ое направление размерности периодическое и *i*-ая координата выходит за пределы, значение автоматически сдвигается *0 < coords(i) < dims(i)*.
- В противном случае - ошибка

Пример MPI_Cart_create (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <mpi.h>
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    // Size of the default communicator
    int size;
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    // Ask MPI to decompose our processes in a 2D cartesian grid for us
    int dims[2] = {0, 0};
    MPI_Dims_create(size, 2, dims);
```

Пример MPI_Cart_create (2)

```
// Make both dimensions periodic
int periods[2] = {true, true};

// Let MPI assign arbitrary ranks if it deems it necessary
int reorder = true;

// Create a communicator given the 2D torus topology.
MPI_Comm new_communicator;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder,
    &new_communicator);

// My rank in the new communicator
int my_rank;
MPI_Comm_rank(new_communicator, &my_rank);
```

Пример MPI_Cart_create (3)

```
// Get my coordinates in the new communicator
int my_coords[2];
MPI_Cart_coords(new_communicator, my_rank, 2, my_coords);

// Print my location in the 2D torus.
printf("[MPI process %d] I am located at (%d, %d).\n", my_rank,
    my_coords[0], my_coords[1]);

MPI_Finalize();

return EXIT_SUCCESS;
}
```

Определение сбалансированного распределения процессов по решетке

int **MPI_Dims_create** (int **nnodes**, int **ndims**, int ***dims**)

nnodes - число процессов

ndims - размер решетки

dims - число элементов по измерениям решетки

- Помогает определить сбалансированное распределение процессов по измерениям решетки.
- Если `dims[i]` – положительное целое, это измерение не будет модифицироваться

dims before call	Function call	dims on return
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 2, dims)	erroneous call

Пример использования MPI_Dims_create

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

```
int dim[3];  
dim[0] = 0; // let MPI arrange  
dim[1] = 0; // let MPI arrange  
dim[2] = 3; // I want exactly 3 planes
```

```
MPI_Dims_create(nprocs, 3, dim);
```

```
if (dim[0]*dim[1]*dim[2] < nprocs) {  
    fprintf(stderr, "WARNING: some processes are not in use!\n")  
}
```

```
int period[] = {1, 1, 0};  
int reorder = 0;
```

```
MPI_Cart_create(MPI_COMM_WORLD, 3, dim, period, reorder,  
&cube_comm);
```

Определение соседей: MPI_CART_SHIFT

- Получение номеров посылающего (**source**) и принимающего (**dest**) процессов в декартовой топологии коммутатора **comm** для осуществления сдвига вдоль измерения **direction** на величину **disp**.

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int  
    *source, int *dest )
```


MPI_CART_SHIFT

int **MPI_Cart_shift**(MPI_Comm comm, int direction, int displ, int *source, int *dest)

comm - коммуникатор с декартовой топологией;

direction - измерение, вдоль которого выполняется сдвиг;

disp - величина сдвига (может быть как положительной, так и отрицательной; >0 – сдвиг влево/вверх, <0 – сдвиг вправо/вниз)

source - номер процесса, от которого должны быть получены данные;

dest - номер процесса, которому должны быть посланы данные.

MPI_CART_SHIFT

Для периодических измерений осуществляется циклический сдвиг, для непериодических – линейный сдвиг.

Для n -мерной декартовой решетки значение `direction` должно быть в пределах от 0 до $n-1$.

Значения `source` и `dest` можно использовать, например, для обмена функцией `MPI_Sendrecv`.

В случае линейного сдвига в качестве `source` или `dest` можно использовать `MPI_PROC_NULL`.

Пример: Sendrecv в 1D решетке

```
...  
int dim[1], period[1];  
dim[0] = nprocs;  
period[0] = 1;  
MPI_Comm ring_comm;
```

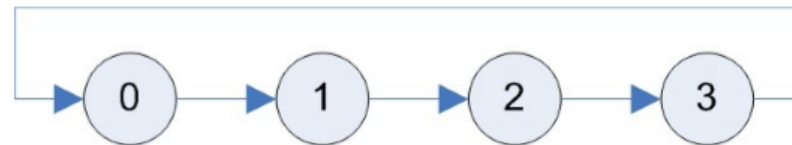
```
MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &ring_comm);
```

```
int source, dest;
```

```
MPI_Cart_shift(ring_comm, 0, 1, &source, &dest);
```

```
MPI_Sendrecv(right_boundary, n, MPI_INT, dest, rtag,  
             left_boundary, n, MPI_INT, source, ltag,  
             ring_comm, &status);
```

```
...
```

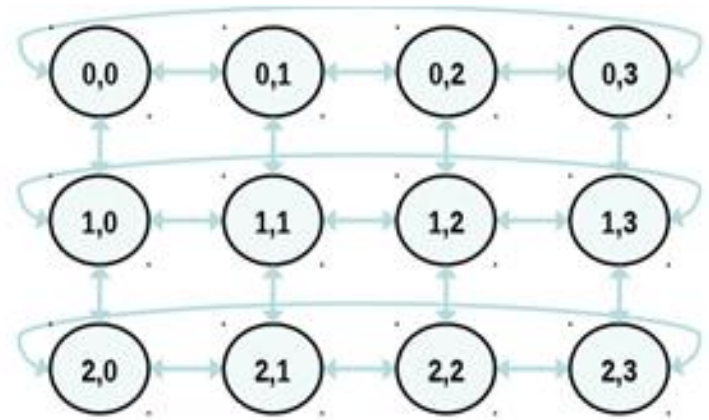


Пример: Sendrecv в 2D решетке

```
...
int dim[] = {4,3};
int period [] = {1,0};
int source, dest;

MPI_Comm grid_comm;

MPI_Cart_create (MPI_COMM_WORLD,2,
    dim, period, 0 , &grid_comm);
for (int dimension = 0; dimension <2; dimension++) {
    for ( int versus = -1; versus < 2; versus+=2) {
        MPI_Cart_shift (grid_comm, dimension, versus, &source, &dest);
        MPI_Sendrecv( buffer, n, MPI_INT, source, stag,
            buffer, n, MPI_INT, dest, dtag,
            grid_comm, &status);
    }
}
```



Создание подрешетки

```
int MPI_Cart_sub (MPI_Comm comm_old,  
                  int remain_dims[], MPI_Comm *new_comm)
```

comm_old – старый коммуникатор

i-ый элемент в **remain_dims** показывает, содержится ли *i*-ая размерность в подрешетке (true) или нет (false) (вектор логических элементов)

new_comm – новый коммуникатор

```
int dim[] = {2,3,4};
```

```
int remain_dims[]={1,0,1}; // 3 comm with 2x4 processes 2D grid
```

```
...
```

```
int remain_dims[]={0,0,1}; // 6 comm with 4 processes 1D topology
```

MPI_CARTDIM_GET

- Определение числа измерений в решетке.

```
int MPI_Cartdim_get( MPI_Comm comm, int* ndims )
```

- comm коммуникатор (решетка)
- ndims число измерений

Пример декартовой решетки (send&recv, mesh)

```
MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

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

if (numtasks == SIZE) {
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims,
periods, reorder, &cartcomm);
    MPI_Comm_rank(cartcomm, &rank);
    MPI_Cart_coords(cartcomm, rank, 2, coords);
    MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP],
&nbrs[DOWN]);
    MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT],
&nbrs[RIGHT]);

    outbuf = rank;
```

```
for (i=0; i<4; i++) {
    dest = nbrs[i];
    source = nbrs[i];
    MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD, &reqs[i]);
    MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
MPI_COMM_WORLD, &reqs[i+4]);
}
MPI_Waitall(8, reqs, stats);
printf("rank= %d coords= %d %d
neighbors(u,d,l,r)= %d %d %d %d inbuf(u,d,l,r)=
%d %d %d %d\n",
rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],n
brs[LEFT],inbuf[UP],inbuf[DOWN],inbuf[LEFT],in
buf[RIGHT]);
}
else
    printf("Must specify %d tasks.
Terminating.\n",SIZE);
MPI_Finalize();
}
```