Стандартный (асинхронный режим)

Блокирующий вариант - **`MPI_Send(…);`**

Возвращает управление после того как сообщение
или ушло в сеть,
или скопировано в промежуточный буфер,
или передано процессу получателю
(выбор остается за MPI).

Неблокирующий **- `MPI_Isend(…);`**

Возврат из функции происходит сразу.
Успешность проверки гарантирует, что сообщение ушло …

Синхронный режим

Блокирующий вариант - **`MPI_Ssend(…);`**

Возврат при получении уведомления о начале приема процессом получателем.

Неблокирующий - **`MPI_Issend(…);`**

Успешность проверки гарантирует, что сообщение ушло в сеть и ***начат прием сообщения***.

Буферизированный режим

Блокирующий вариант - **`MPI_Bsend(…);`**

Возврат после копирования сообщения в буфер, в котором оно ожидает операции обмена.

Неблокирующий - **`MPI_Ibsend(…);`**

Успешность проверки гарантирует, что ***сообщение скопировано*** ...

Режим «по готовности»

Блокирующий вариант - **`MPI_Rsend(…);`**

Должна начинаться только если инициирован соответствующий прием!

(***иначе, результат не определен***).

Возврат после того ***как сообщение ушло …***
(не гарантирует получение сообщения).

Неблокирующий - **`MPI_Irsend(…);`**

Успешность проверки гарантирует, что ***сообщение ушло*** ...

Синтаксис: **функции блокирующей отправки**

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

buf: начальный адрес буфера сообщений (IN);

count: число элементов в сообщении (IN);

datatype: тип элементов в сообщении (IN);

dest: ранг задачи назначения в коммуникаторе comm (IN);

tag: тэг сообщения (IN);

comm: коммуникатор (IN).

В MPI *поддерживаются основные типы данных языка* C

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	-

Типы MPI_BYTE и MPI_PACKED

MPI_BYTE используется, если система ***не*** должна выполнять ***преобразование между*** различными ***представлениями данных***.

MPI_PACKED используется для передачи ***упакованных данных.***

Функции упаковки и распаковки:

`MPI_Pack(…)` и `MPI_Unpack(…)`.

Синтаксис: **функции неблокирующей отправки**

```
int MPI_Isend (void *buf,
               int count,
               MPI_Datatype datatype,
               int dest,
               int tag,
               MPI_Comm comm,
               MPI_Request *request)
```

---””---;

request - идентифицирует коммуникационное событие (OUT).

Прием сообщения

Существует две функции приема:

блокирующая **`MPI_Recv`**(…)

и неблокирующая **`MPI_Irecv`**(…).

Каждая из них совместима с любой функцией передачи.

Прием сообщения. **Блокирующий**

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

buf: адрес буфера приема (OUT);
count: максимальное число элементов в принимаемом сообщении (IN);
datatype: тип элементов принимаемого сообщения (IN);
source: номер процесса-отправителя (IN);
msgtag: тэг принимаемого сообщения (IN);
comm: коммуникатор (IN);
status: статус коммуникационного события (OUT).

После возврата из функции все элементы сообщения приняты.

Номер отправителя MPI_ANY_SOURCE означает прием от любого процесса.

MPI_ANY_TAG – прием с любым тэгом.

Прием сообщения. **Неблокирующий**

```
int MPI_Irecv(void *buf,
              int count,
              MPI_Datatype datatype,
              int source,
              int msgtag,
              MPI_Comm comm,
              MPI_Request *request)
```

Окончание процесса ***приема*** можно определить с помощью параметра **request**.

Блокирующая проверка завершения.

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

Возвращает управление, когда операция завершена.

***request** устанавливается в MPI_REQUEST_NULL.

***status** содержит информацию о завершенной операции.

Неблокирующая проверка завершения

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

flag = **true** если коммуникационное событие завершено, иначе значение статуса не определено.

Статусный объект содержит

источник, тэг принятого сообщения
и код ошибки.

Для получения информации о длине принятого сообщения нужно вызвать функцию

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

«Апробация»

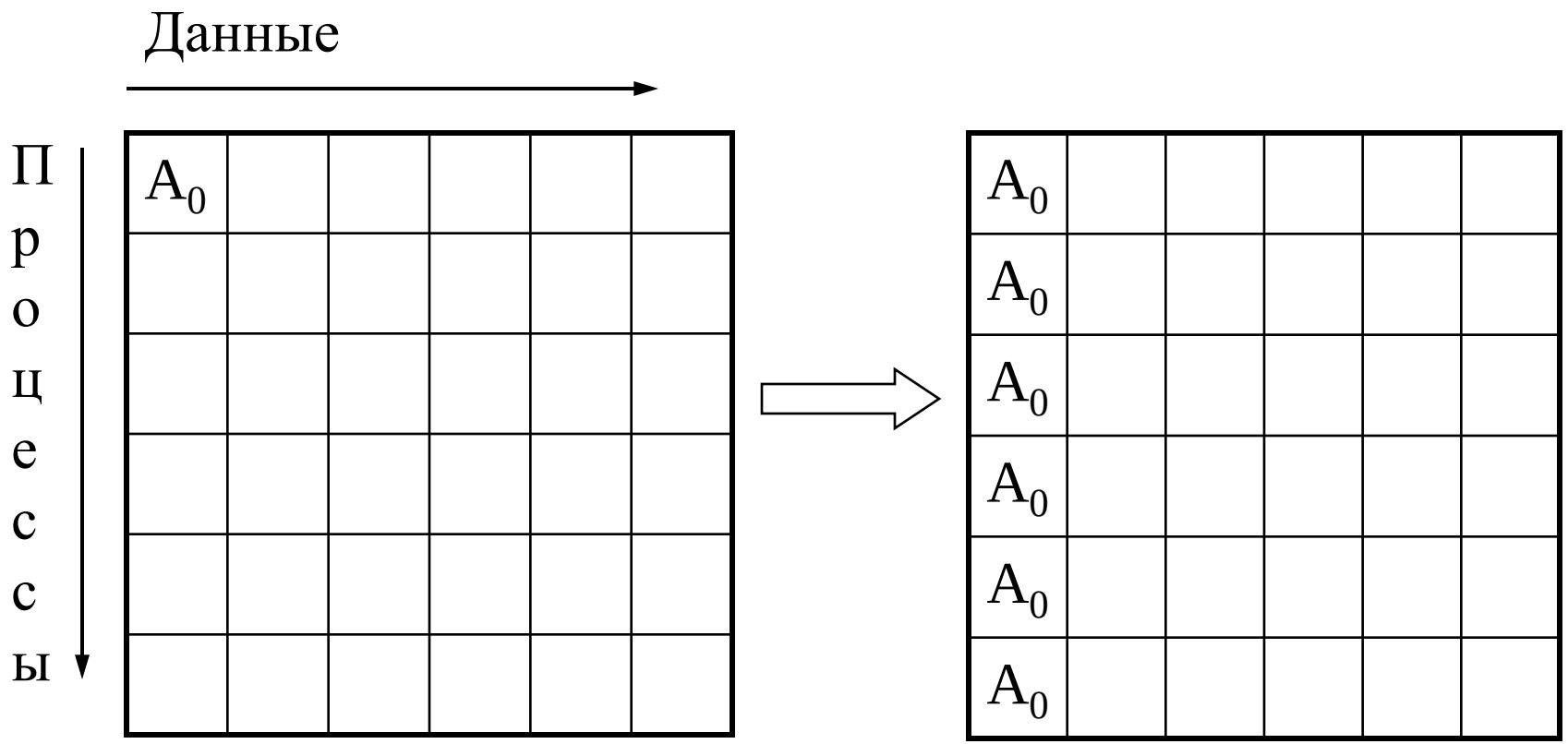
Операции `MPI_Probe` и `MPI_Iprobe` позволяют «проверить» входные сообщения (***получить статусный объект***) без их реального приема.

```
int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status);

int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
```

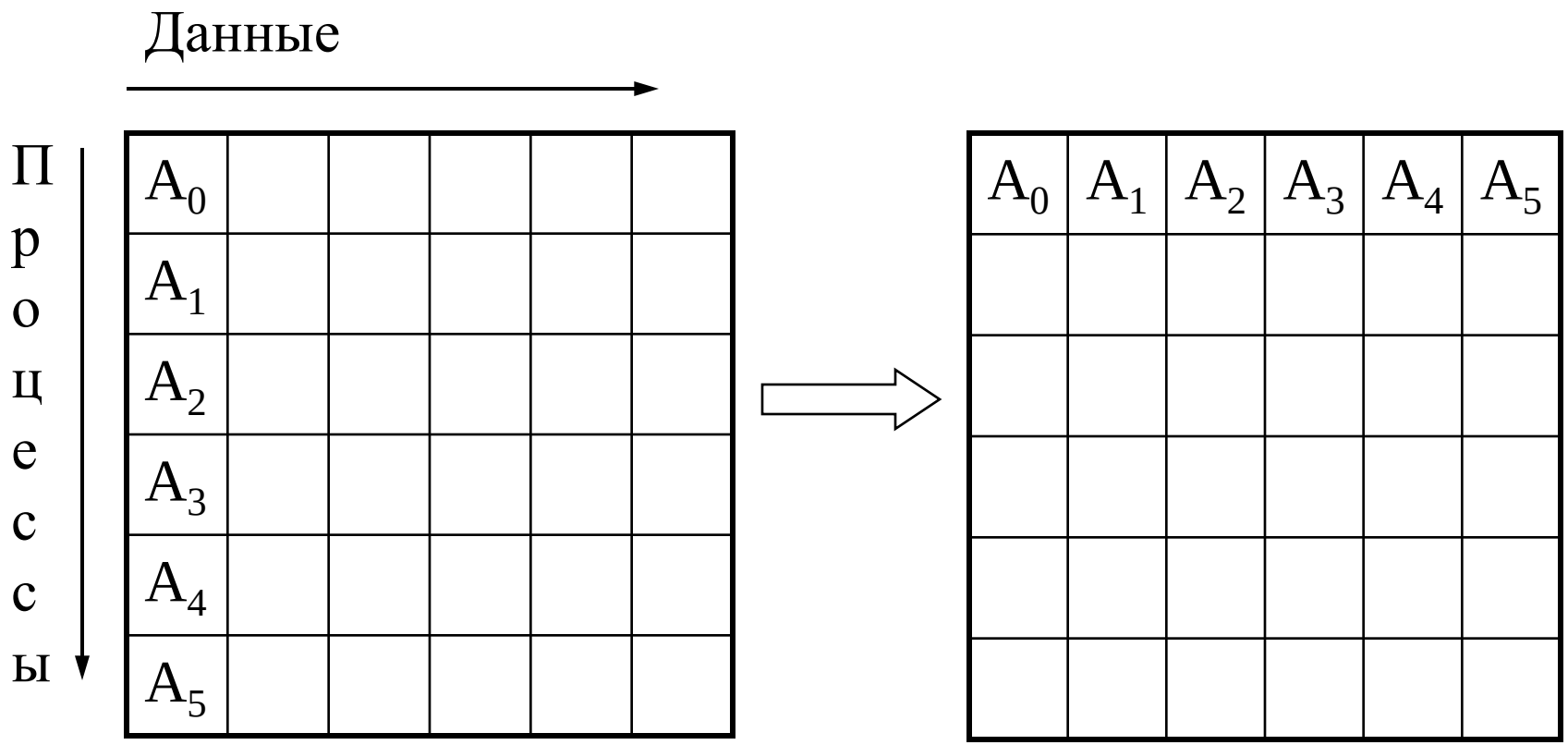
Широковещательный обмен

```
int MPI_Bcast(void* message, int count,
MPI_Datatype datatype,int root, MPI_Comm comm);
```



Сборка данных

```
int MPI_Gather(send_buf, send_count, send_type,
recv_buf, recv_count, recv_type, root, comm);
```



`MPI_Gatherv(…)` - векторный вариант.

Позволяет принимать от каждого процесса **переменное** число элементов данных.

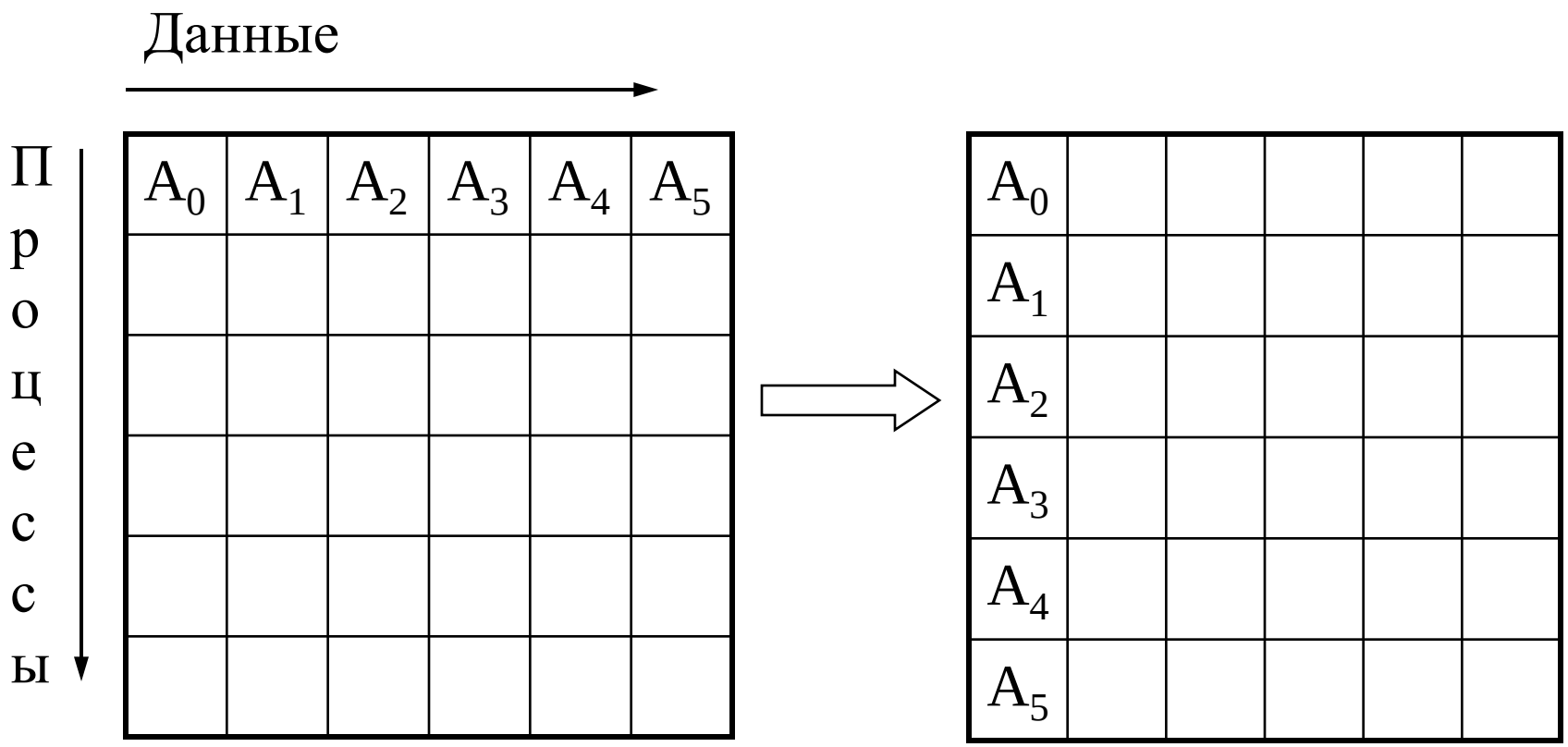
```
MPI_Gatherv (sendbuf,sendcount, sendtype, recvbuf,
recvcounts, displs, recvtype, root, comm)
```

массивы

Рассылка

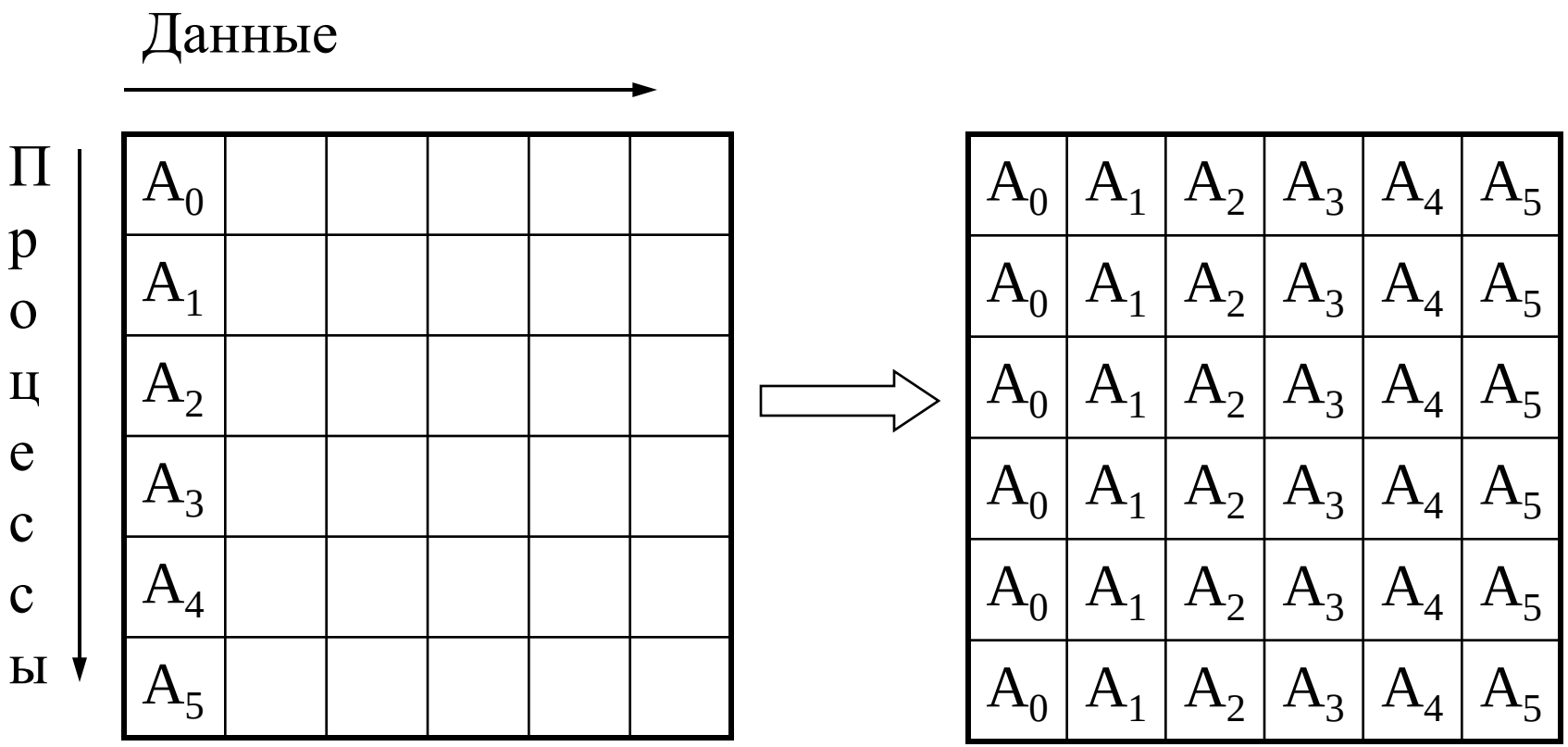
```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm);

MPI_Scatterv (sendbuf, sendcounts, displs, sendtype, recvbuf,
recvcount, recvtype,  root, comm);
```



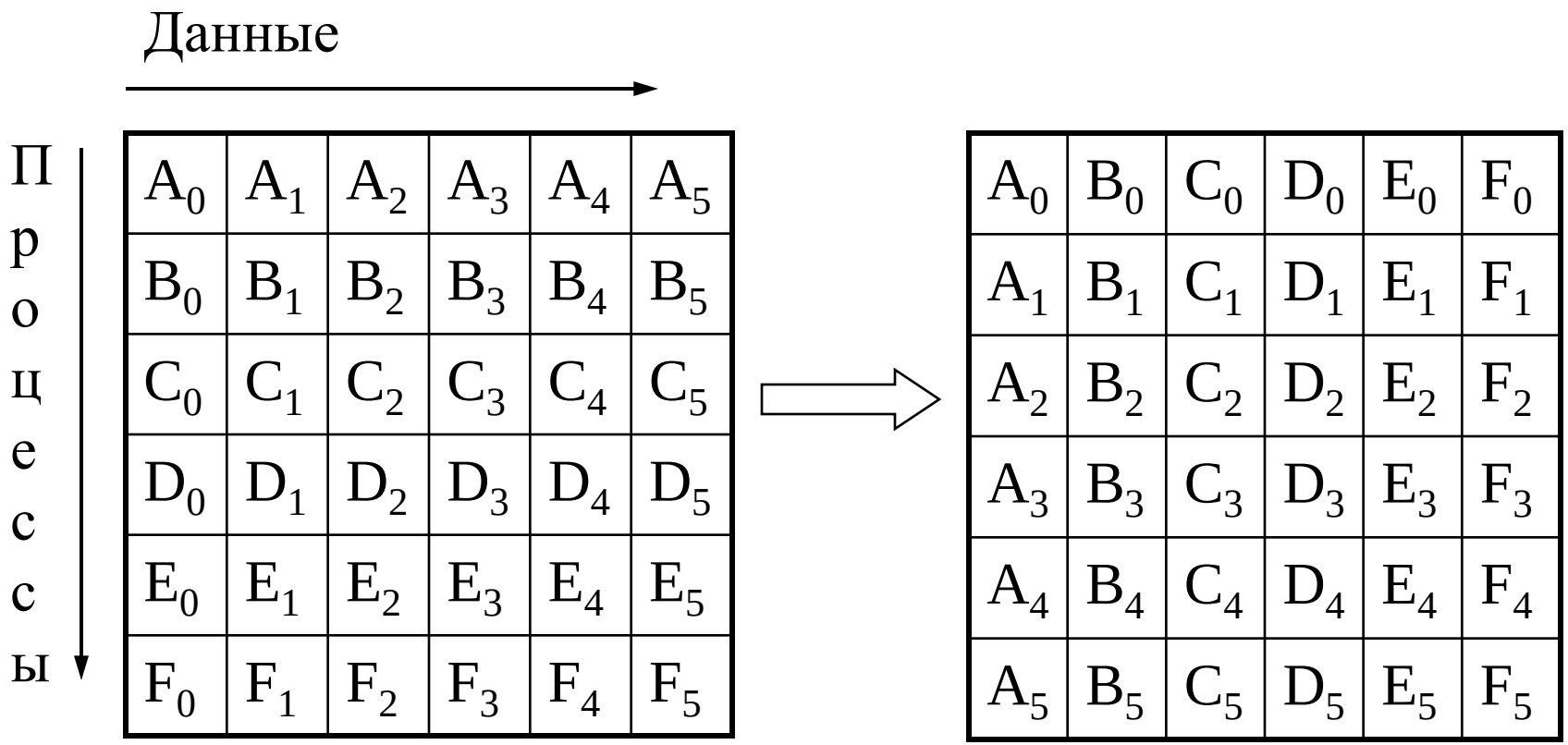
Сборка для всех процессов

```
MPI_Allgather (…);
MPI_Allgatherv (…);
```

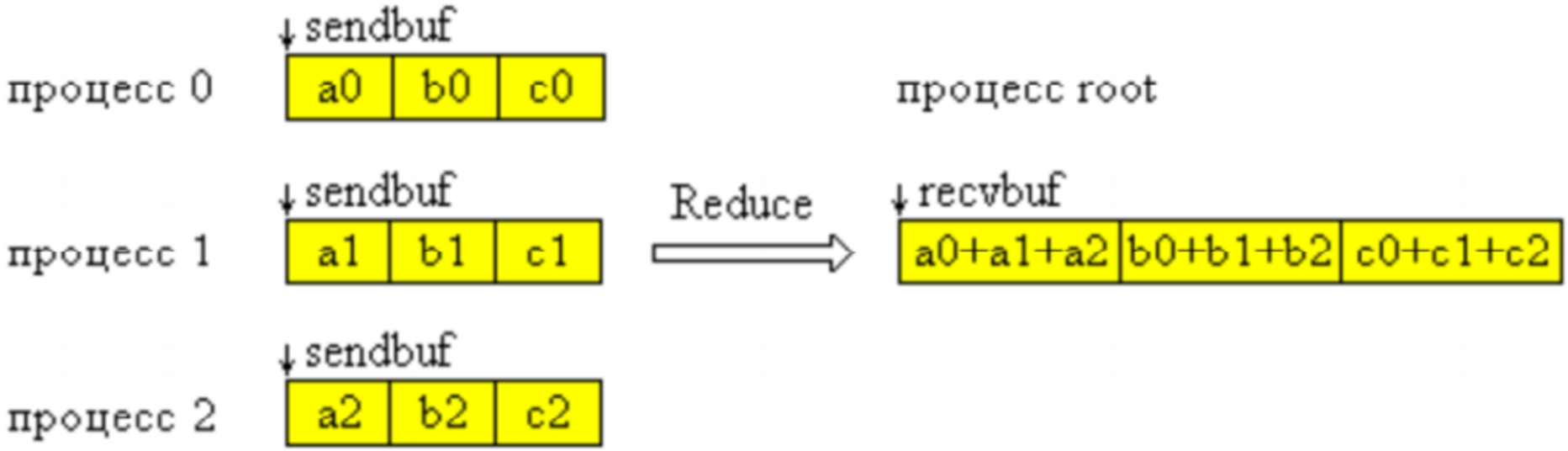


Полный обмен

```
MPI_Alltoall (…);
MPI_Alltoallv (…);
```



```
int MPI_Reduce( void* sendbuf,
                void* recvbuf,
                int count,
                MPI_Datatype datatype,
                MPI_Op op, int root,
                MPI Comm comm)
```



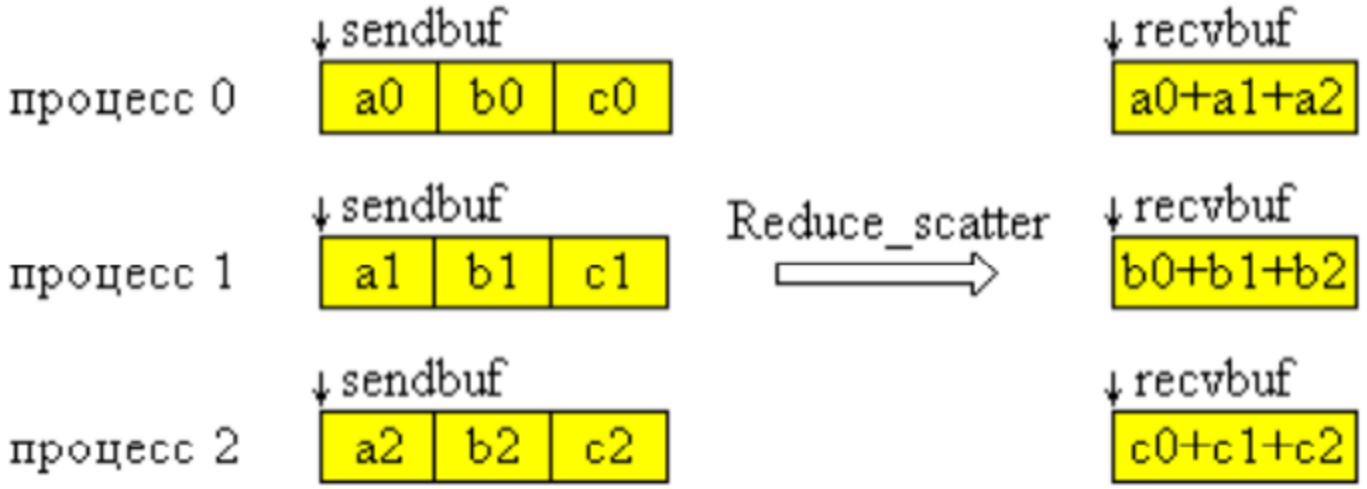
`int` **`MPI_Allreduce`**(…) - результат возвращается всем процессам.

Предопределенные операции:

Имя	**Значение**
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое И
MPI_BAND	поразрядное И
MPI_LOR	логическое ИЛИ
MPI_BOR	поразрядное ИЛИ
MPI_LXOR	логическое исключающее ИЛИ
MPI_BXOR	поразрядное исключающее ИЛИ

Можно определять собственные операции с помощью функции **`MPI_Op_create`**(…).

```
MPI_Reduce_scatter(   void* sendbuf,
                      void* recvbuf,
                      int *recvcounts,
                      MPI_Datatype datatype,
                      MPI_Op op,
                      MPI_Comm comm)
```



Производные типы данных

Назначение:

-передавать сообщения, которые содержат значения различных типов;

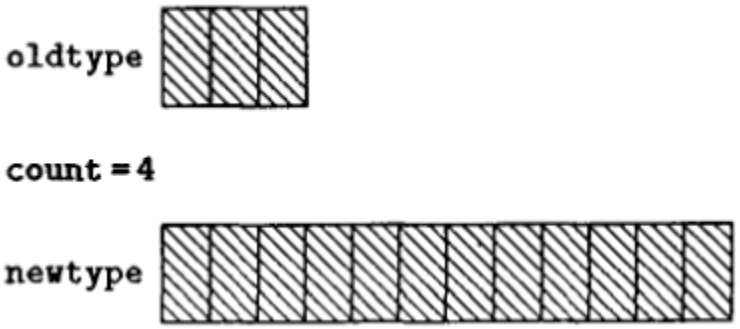
-посылать несмежные данные (например, подблоки матрицы).

Конструкторы типа данных

- Contiguous
- Vector
- Hvector
- Indexed
- Hindexed
- Struct

MPI_Type_contiguous (count, oldtype, newtype);

newtype – конкатенация count копий oldtype.



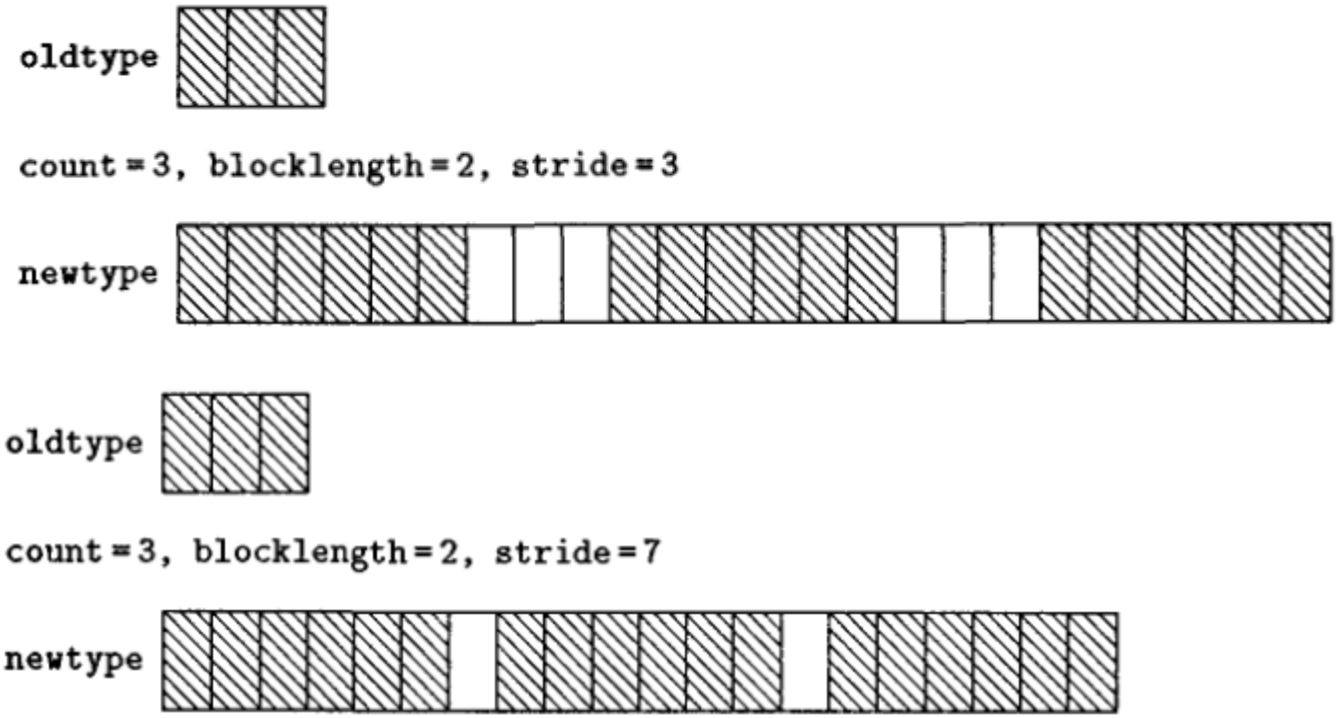
Пример

```
#define M 5
MPI_Datatype type;
int sendbuf[M], *recvbuf;
//...
if(rank==root) // rank-номер процесса
{
recvbuf=(int*)malloc(M*size*sizeof(int)); // size-количество процессов
MPI_Type_contiguous(M,MPI_INT,&type);
MPI_Type_commit(&type);
}
MPI_Gather(sendbuf, M, MPI_INT, recvbuf, 1, type, root,
MPI_COMM_WORLD);
//...
if(rank==root)
  MPI_Type_free(&type);
//...
```

MPI_Type_vector (count, blocklength, **stride**, oldtype, newtype);

MPI_Type_hvector (…);

stride - число **элементов**/**байт** между началами каждого блока.



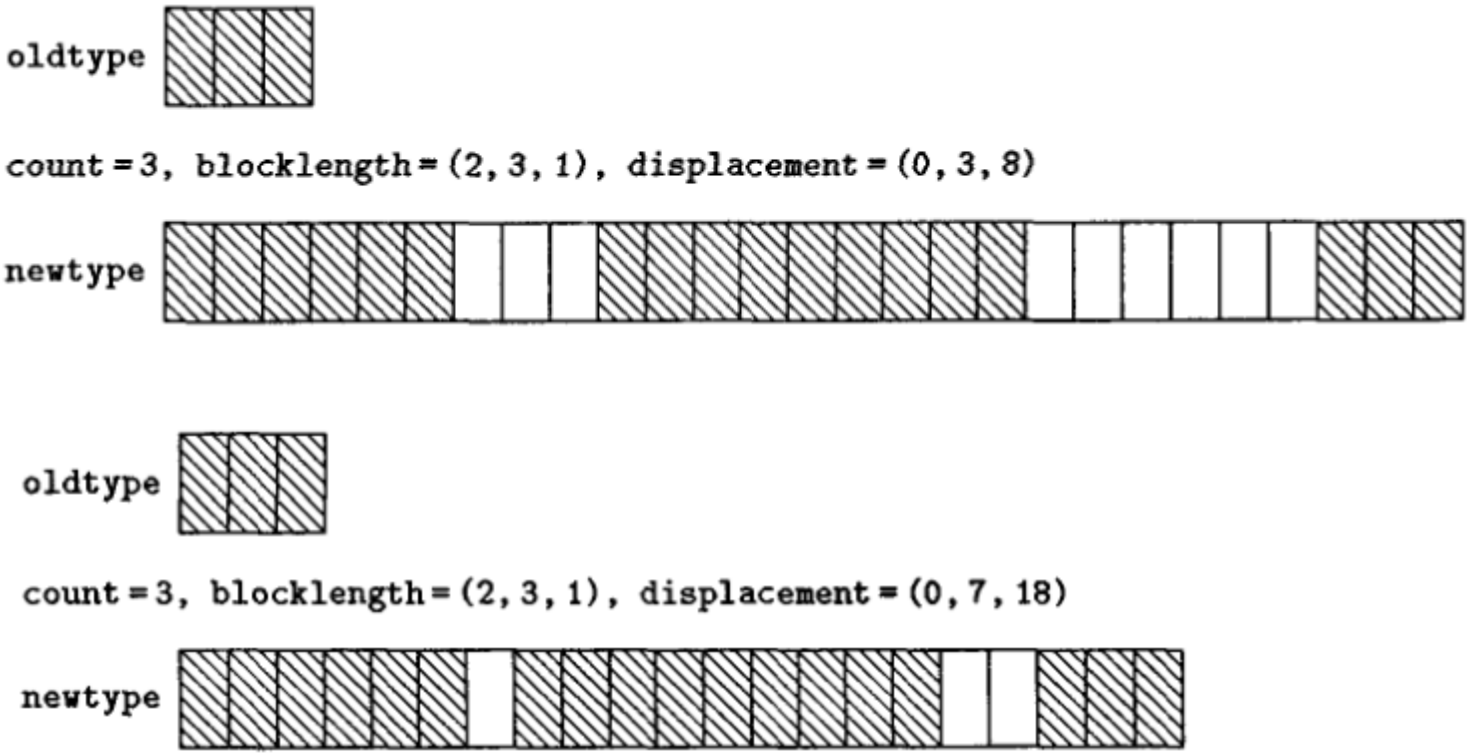
Пример

```
#define M 5
MPI_Datatype row,xpos;
int sendbuf[M][M], sizeint, *recvbuf;
//...
if(rank==1)// rank-номер процесса
recvbuf=(int*)calloc((M*M),sizeof(int));

if(rank==0)
{
MPI_Type_extent(MPI_INT,&sizeint);
MPI_Type_vector(M, 1, M, MPI_INT, &row);
MPI_Type_hvector(M, 1, sizeint, row, &xpos);
MPI_Type_commit(&xpos);
}
//...
if(rank==0)
MPI_Send(sendbuf, 1, xpos, 1, 3, MPI_COMM_WORLD);
if(rank==1)
MPI_Recv(recvbuf, M*M, MPI_INT, 0, 3, MPI_COMM_WORLD);
//...
```

MPI_Type_indexed (count, array_of_blocklengths, array_of_displacements, oldtype, newtype);

MPI_Type_hindexed (…);



Пример

```
#define M 5
MPI_Datatype upper;
MPI_Status st;
int sendbuf[M][M], *disp, *blocks, *recvbuf;
//...
disp =(int*)malloc(M*sizeof(int));
blocks =(int*)malloc(M*sizeof(int));
for(int i=0;i<M;i++){
disp[i]=M*i+i;
blocks[i]=M-i; }
MPI_Type_indexed(M, blocks, disp, MPI_INT, &upper);
MPI_Type_commit(&upper);
if(rank==1)// rank-номер процесса
recvbuf =(int*)calloc((M*M),sizeof(int));
//...
if(rank==0)
MPI_Send(sendbuf, 1, upper, 1, 3, MPI_COMM_WORLD);
if(rank==1)
MPI_Recv(recvbuf, 1, upper, 0, 3, MPI_COMM_WORLD, &st);
//...
```

```
MPI_Type_struct (count,
                 array_of_blocklengths,
                 array_of_displacements,
                 array_of_types,
                 newtype);
```