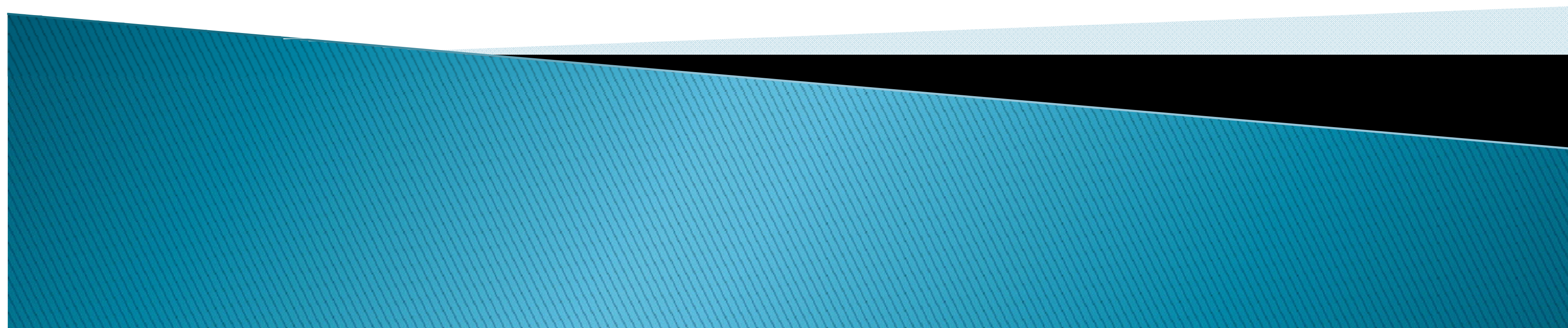
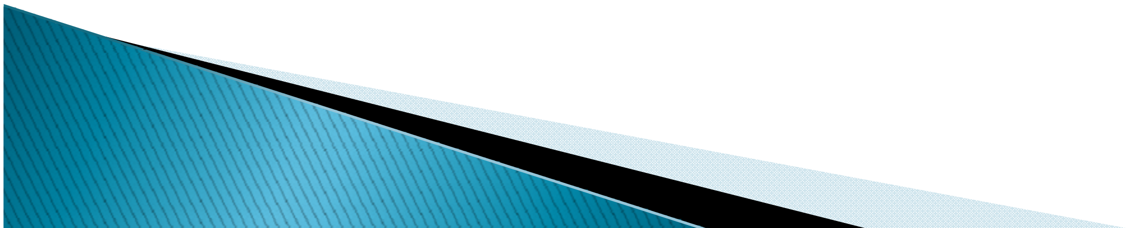
Обзор технологии параллельного программирования MPI



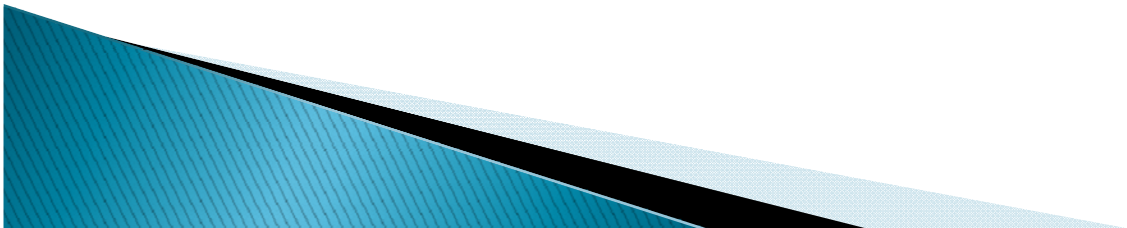
Алгоритм Якоби. MPI-версия

```
/* dynamically allocate data structures */
A = malloc ((nrow+2) * L * sizeof(double));
B = malloc ((nrow) * L * sizeof(double));
for(i=1; i<=nrow; i++)
        for(j=0; j<=L-1; j++)
        {
                A[i][j]=0.;
                B[i-1][j]=1.+startrow+i-1+j;
        }
```



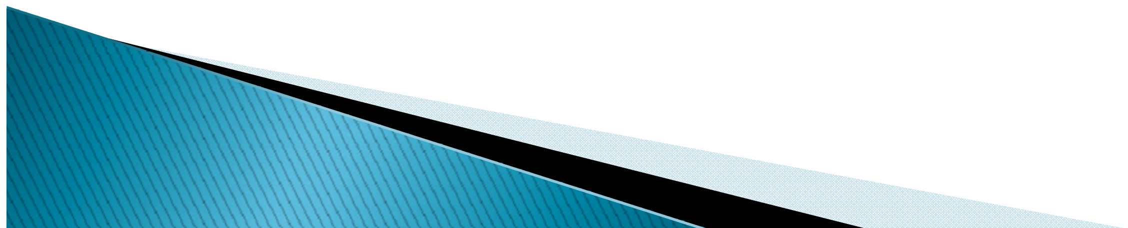
Алгоритм Якоби. MPI-версия

```
/******  iteration loop  *************************/
t1=MPI_Wtime();
 for(it=1; it<=ITMAX; it++)
{
        for(i=1; i<=nrow; i++)
        {
                if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1)))
                    continue;
                for(j=1; j<=L-2; j++)
                {
                        A[i][j] = B[i-1][j];
                }
        }
```



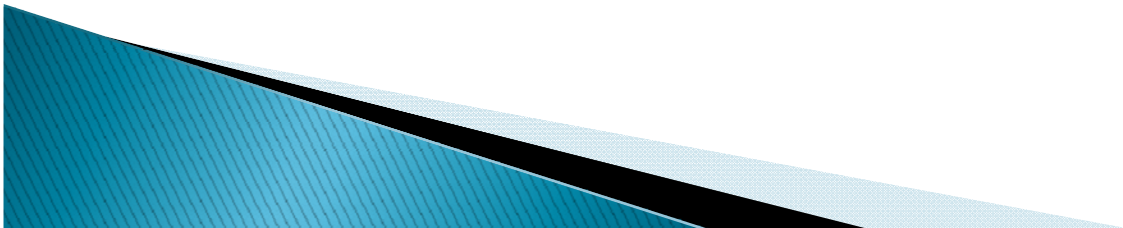
Алгоритм Якоби. MPI-версия

```
if(myrank!=0)
        MPI_Irecv(&A[0][0],L,MPI_DOUBLE, myrank-1, 1215,
            MPI_COMM_WORLD, &req[0]);
if(myrank!=ranksize-1)
        MPI_Isend(&A[nrow][0],L,MPI_DOUBLE, myrank+1, 1215,
            MPI_COMM_WORLD,&req[2]);
if(myrank!=ranksize-1)
        MPI_Irecv(&A[nrow+1][0],L,MPI_DOUBLE, myrank+1, 1216,
            MPI_COMM_WORLD, &req[3]);
if(myrank!=0)
        MPI_Isend(&A[1][0],L,MPI_DOUBLE, myrank-1, 1216,
            MPI_COMM_WORLD,&req[1]);
ll=4; shift=0;
if (myrank==0) {ll=2;shift=2;}
if (myrank==ranksize-1) {ll=2;}
MPI_Waitall(ll,&req[shift],&status[0]);
```



Алгоритм Якоби. MPI-версия

```
            for(i=1; i<=nrow; i++)
            {
                  if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1))) continue;
                  for(j=1; j<=L-2; j++)
                        B[i-1][j] = (A[i-1][j]+A[i+1][j]+
                              A[i][j-1]+A[i][j+1])/4.;
            }
      }/*DO it*/
printf("%d: Time of task=%lf\n",myrank,MPI_Wtime()-t1);
MPI_Finalize ();
return 0;
}
```



Проверка завершения любого числа обменов

- Проверка завершения любого числа обменов выполняется подпрограммой:

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

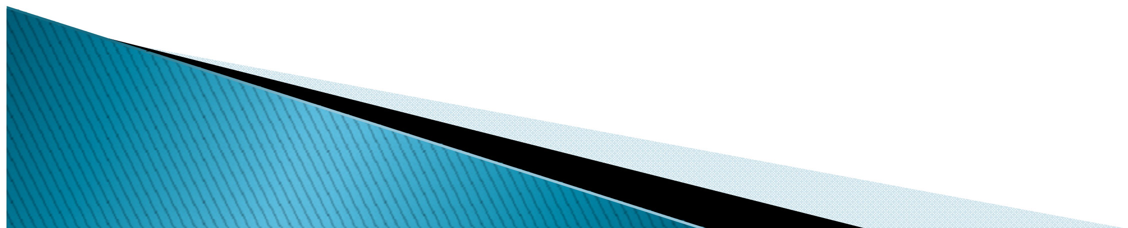
- Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.
- Входные параметры:
  - `requests` – запрос;
  - `count` – количество элементов в массиве `requests`.
- Выходные параметры:
  - `index` – индекс запроса (в языке С это целое число от 0 до `count` – 1) в массиве `requests`;
  - `status` – статус.

Неблокирующие процедуры проверки

- Подпрограмма `MPI_Test` выполняет неблокирующую проверку завершения приема или передачи сообщения:

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

- Входной параметр: идентификатор операции обмена `request`.
- Выходные параметры:
  - `flag` — «истина», если операция, заданная идентификатором `request`, выполнена;
  - `status` — статус выполненной операции.

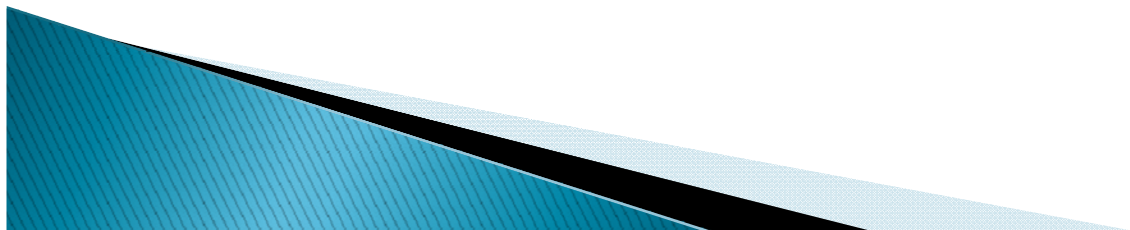


Неблокирующая проверка завершения всех обменов

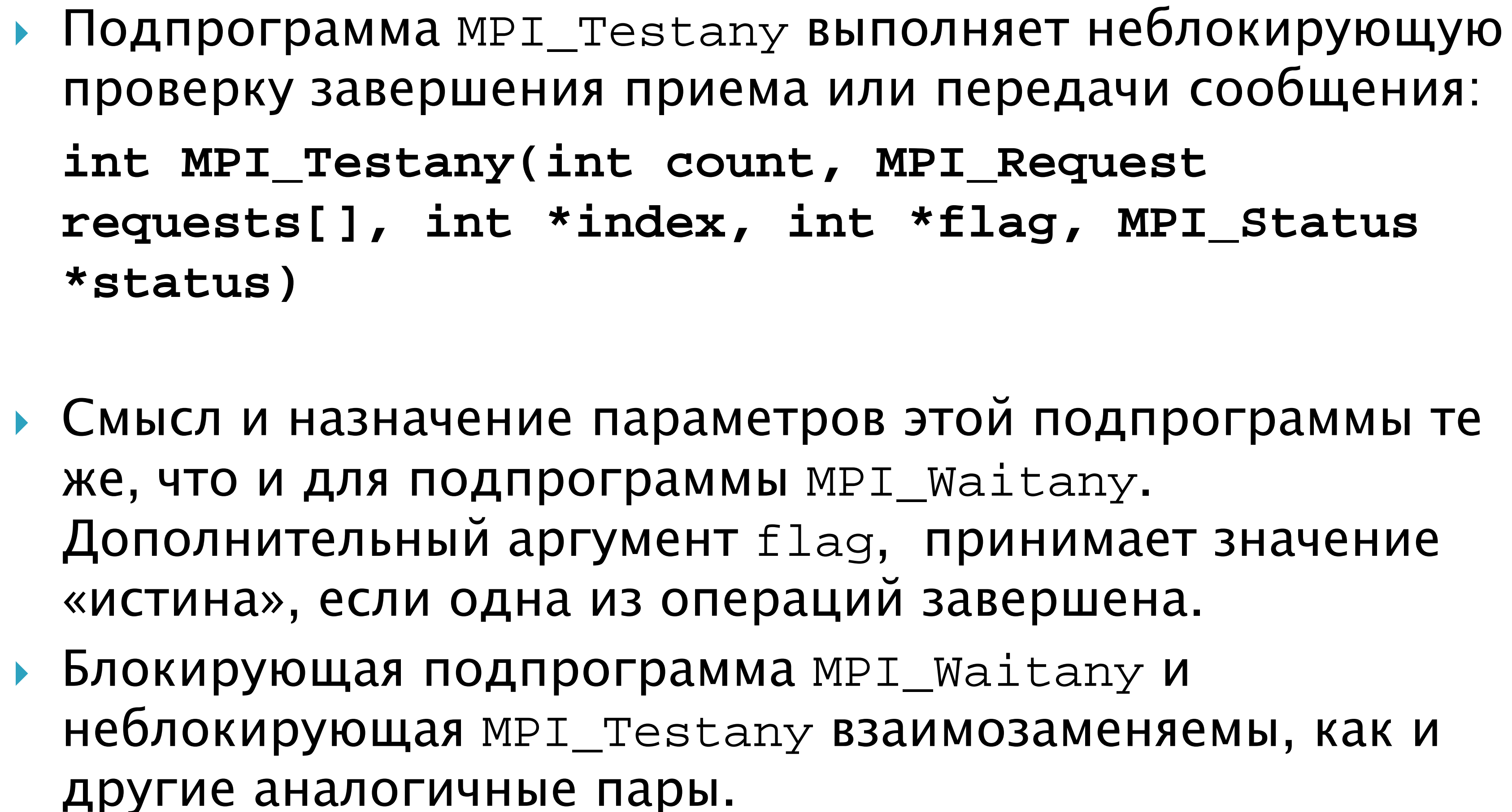
- Подпрограмма `MPI_Testall` выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

```
int MPI_Testall(int count, MPI_Request requests[],
int *flag, MPI_Status statuses[])
MPI_Testall(count, requests, flag, statuses, ierr)
```

- При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
- Параметр `count` – количество запросов.
- Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.



Неблокирующая проверка любого числа обменов



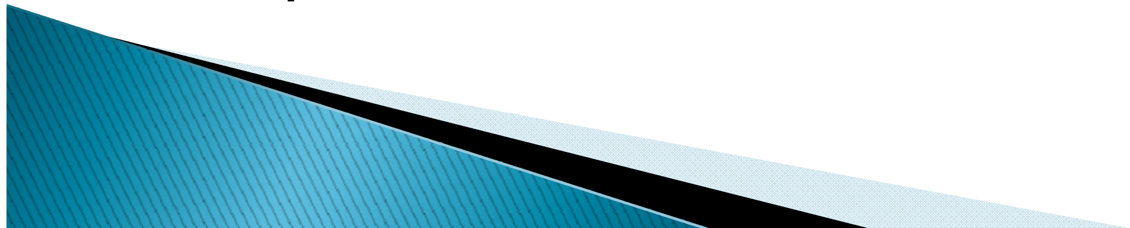
- Подпрограмма `MPI_Testany` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Testany(int count, MPI_Request
requests[], int *index, int *flag, MPI_Status
*status)
```

- Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы `MPI_Waitany`. Дополнительный аргумент `flag`, принимает значение «истина», если одна из операций завершена.
- Блокирующая подпрограмма `MPI_Waitany` и неблокирующая `MPI_Testany` взаимозаменяемы, как и другие аналогичные пары.

Другие операции проверки

- Подпрограммы MPI_Waitsome и MPI_Testsome действуют аналогично подпрограммам MPI_Waitany и MPI_Testany, кроме случая, когда завершается более одного обмена.
- В подпрограммах MPI_Waitany и MPI_Testany обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для MPI_Waitsome и MPI_Testsome статус возвращается для всех завершенных обменов.
- Эти подпрограммы можно использовать для определения, сколько обменов завершено.

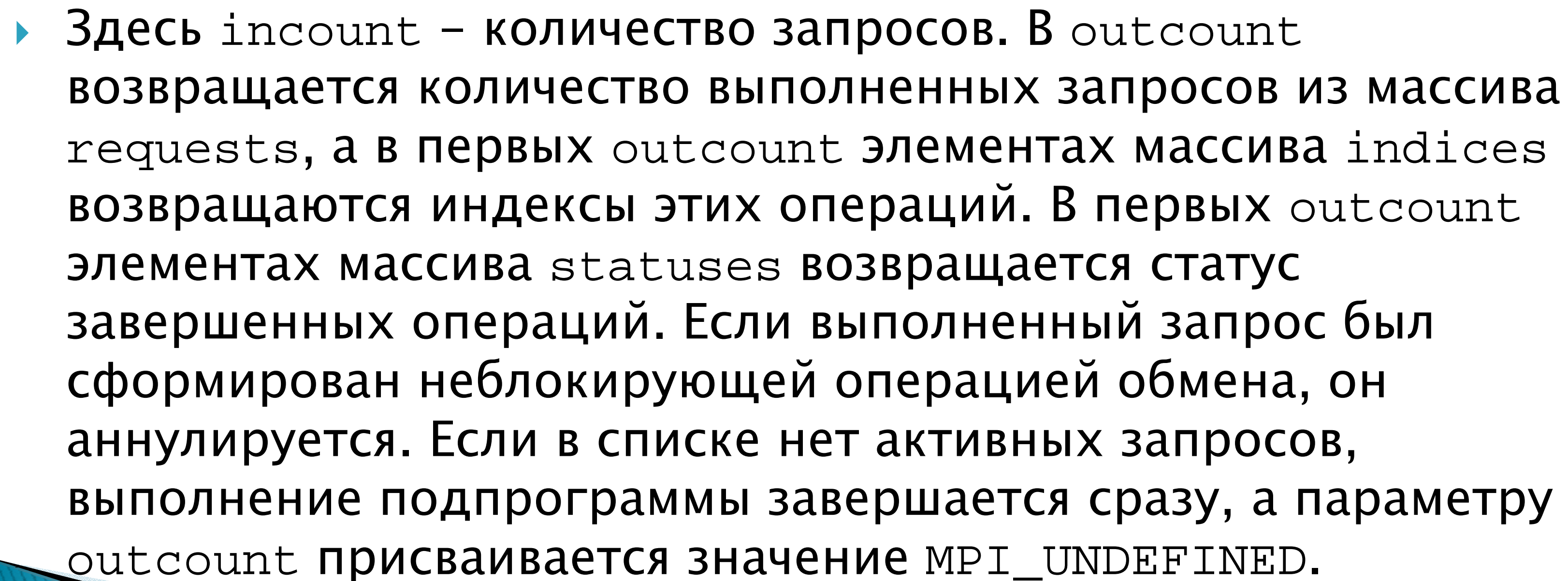


Другие операции проверки

- Блокирующая проверка выполнения обменов:

```
int MPI_Waitsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

- Здесь `incount` – количество запросов. В `outcount` возвращается количество выполненных запросов из массива `requests`, а в первых `outcount` элементах массива `indices` возвращаются индексы этих операций. В первых `outcount` элементах массива `statuses` возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру `outcount` присваивается значение `MPI_UNDEFINED`.

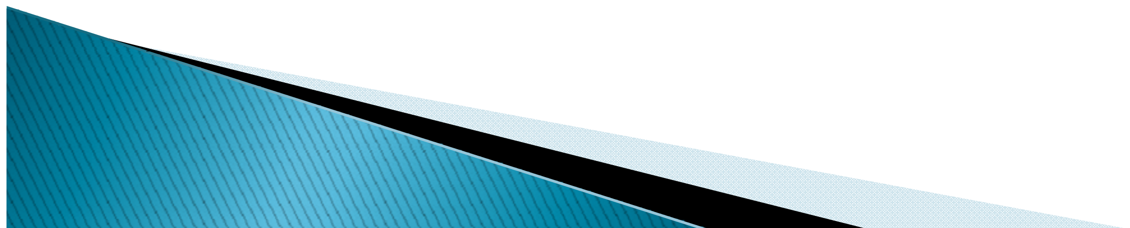


Другие операции проверки

- Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

- Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.



Проверка статуса операции приема сообщения

- Блокирующая проверка:

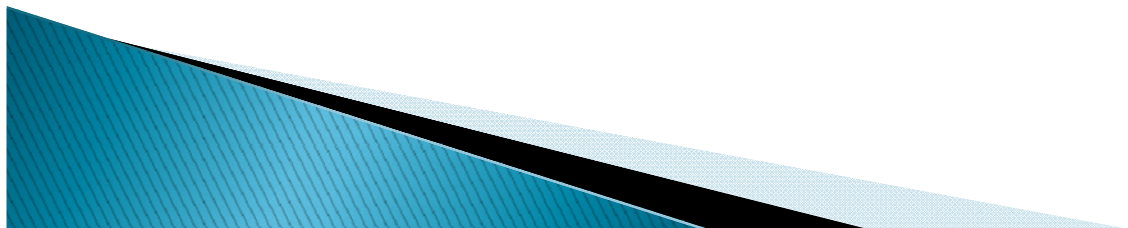
```
int MPI_Probe(int source, int tag, MPI_Comm comm,
  MPI_Status* status)
```

- Неблокирующая проверка:

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

- Входные параметры этой подпрограммы те же, что и у подпрограммы `MPI_Probe`.
- Выходные параметры:
  - `flag` – флаг;
  - `status` – статус.

- Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

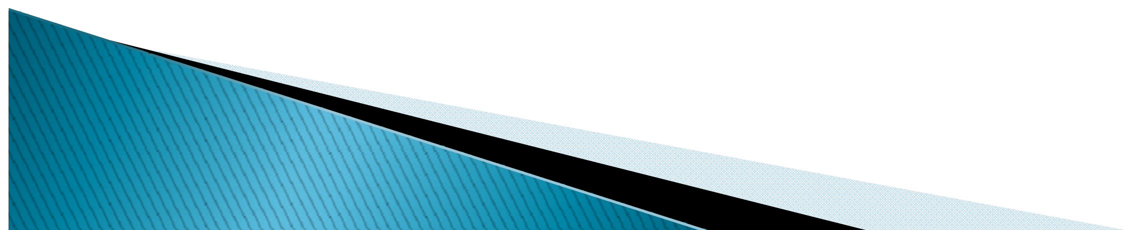


Проверка статуса операции приема сообщения (пример)

```
if (rank == 0) {
   MPI_Send(buf, size, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send to process 1
   printf("0 sent %d numbers to 1\n", size);
} else if (rank == 1) {
  MPI_Status status;
  MPI_Probe(0, 0, MPI_COMM_WORLD, &status); // Probe for an incoming  msg.
  // When probe returns, the status object has the size and other
  // attributes of the incoming message. Get the size of the message.
  MPI_Get_count(&status, MPI_INT, &size);
  // Allocate a buffer just big enough to hold the incoming numbers
  int* bufnumber = (int*)malloc(sizeof(int) * size);
  // Now receive the message with the allocated buffer
   MPI_Recv(bufnumber, size, MPI_INT, 0, 0,
    MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   printf("1 dynamically received %d numbers from 0.\n", size);
   free(bufnumber);
 }
```

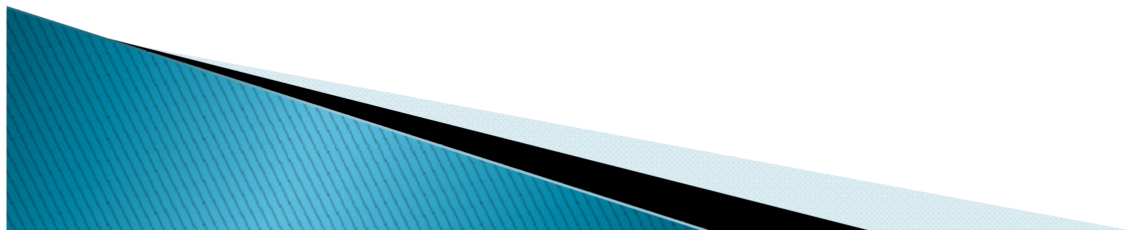
Коллективные операции

- Передача сообщений между группой процессов
- Вызываются ВСЕМИ процессами в коммуникаторе
- Примеры:
  - Broadcast, scatter, gather (рассылка данных)
  - Global sum, global maximum, и.т.д. (редукционные операции)
  - Барьерная синхронизаци



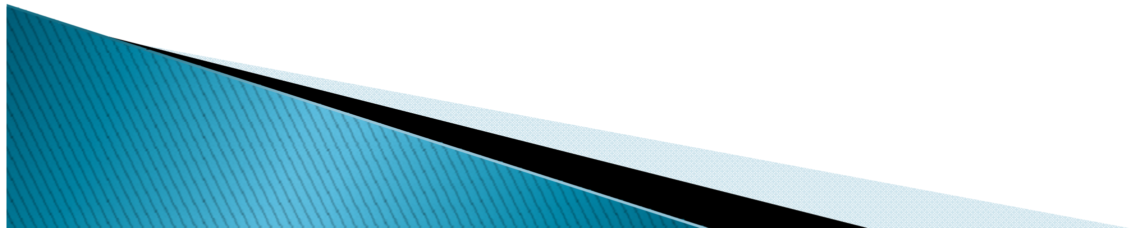
Характеристики коллективных передач

- Коллективные операции не являются помехой операциям типа точка-точка и наоборот
- Все процессы коммуникатора должны вызывать коллективную операцию
- Синхронизация не гарантируется (за исключением барьера)
- Нет тэгов
- Принимающий буфер должен точно соответствовать размеру отсылаемого буфера



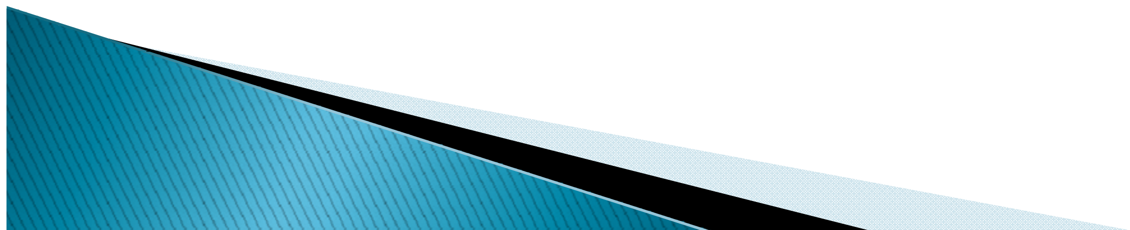
Особенности коллективных передач

```
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;
}
```



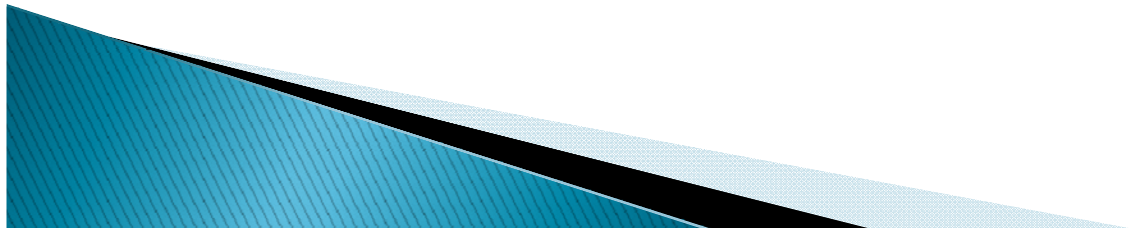
Особенности коллективных передач

```
switch(rank) {
   case 0:
       MPI_Bcast(buf1, count, type, 0, comm0);
       MPI_Bcast(buf2, count, type, 2, comm2);
       break;
   case 1:
       MPI_Bcast(buf1, count, type, 1, comm1);
       MPI_Bcast(buf2, count, type, 0, comm0);
       break;
   case 2:
       MPI_Bcast(buf1, count, type, 2, comm2);
       MPI_Bcast(buf2, count, type, 1, comm1);
       break;
   }
   /* comm0 is {0,1}, comm1 is {1, 2} comm2 is {2,0} */
```



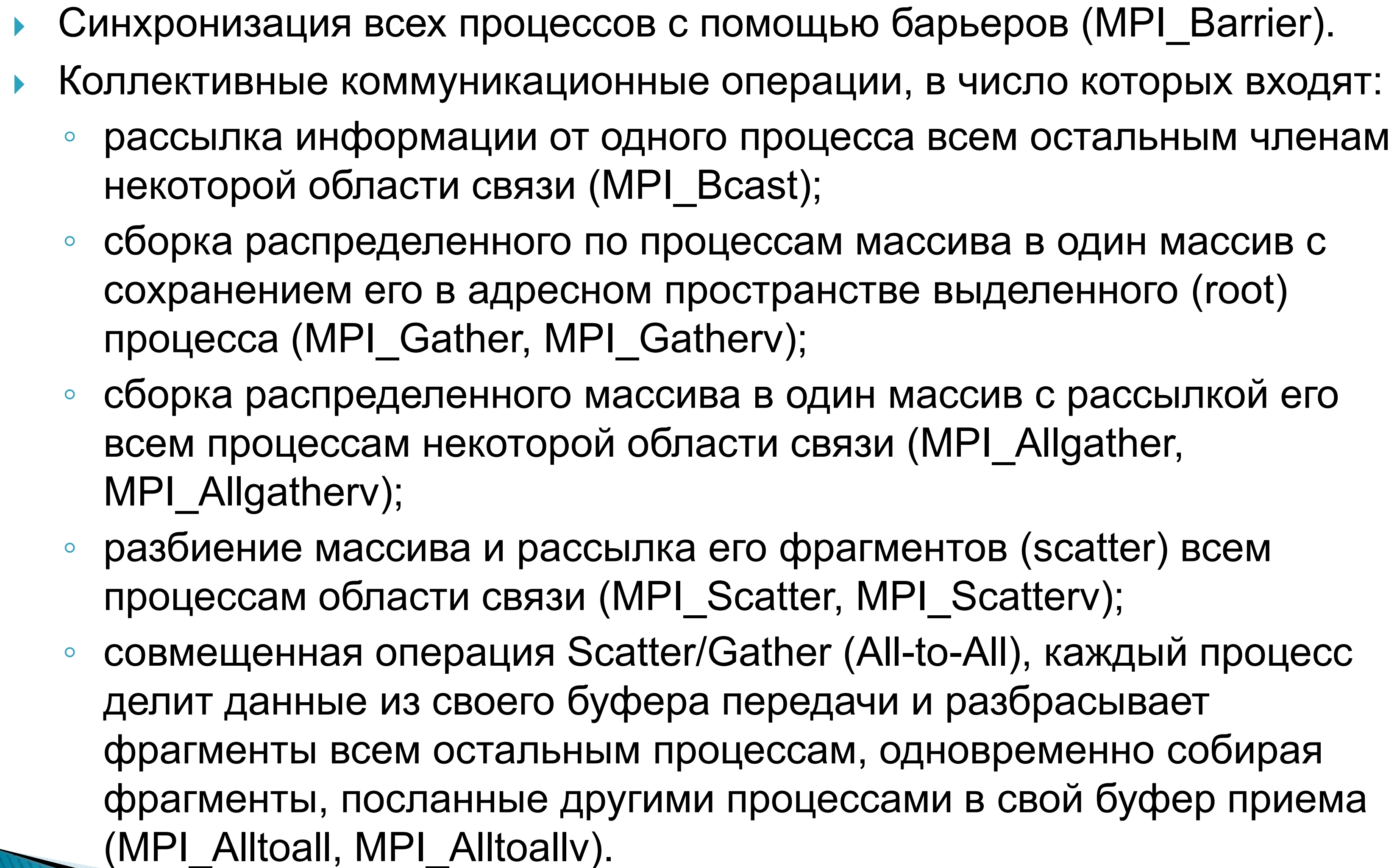
Особенности коллективных передач

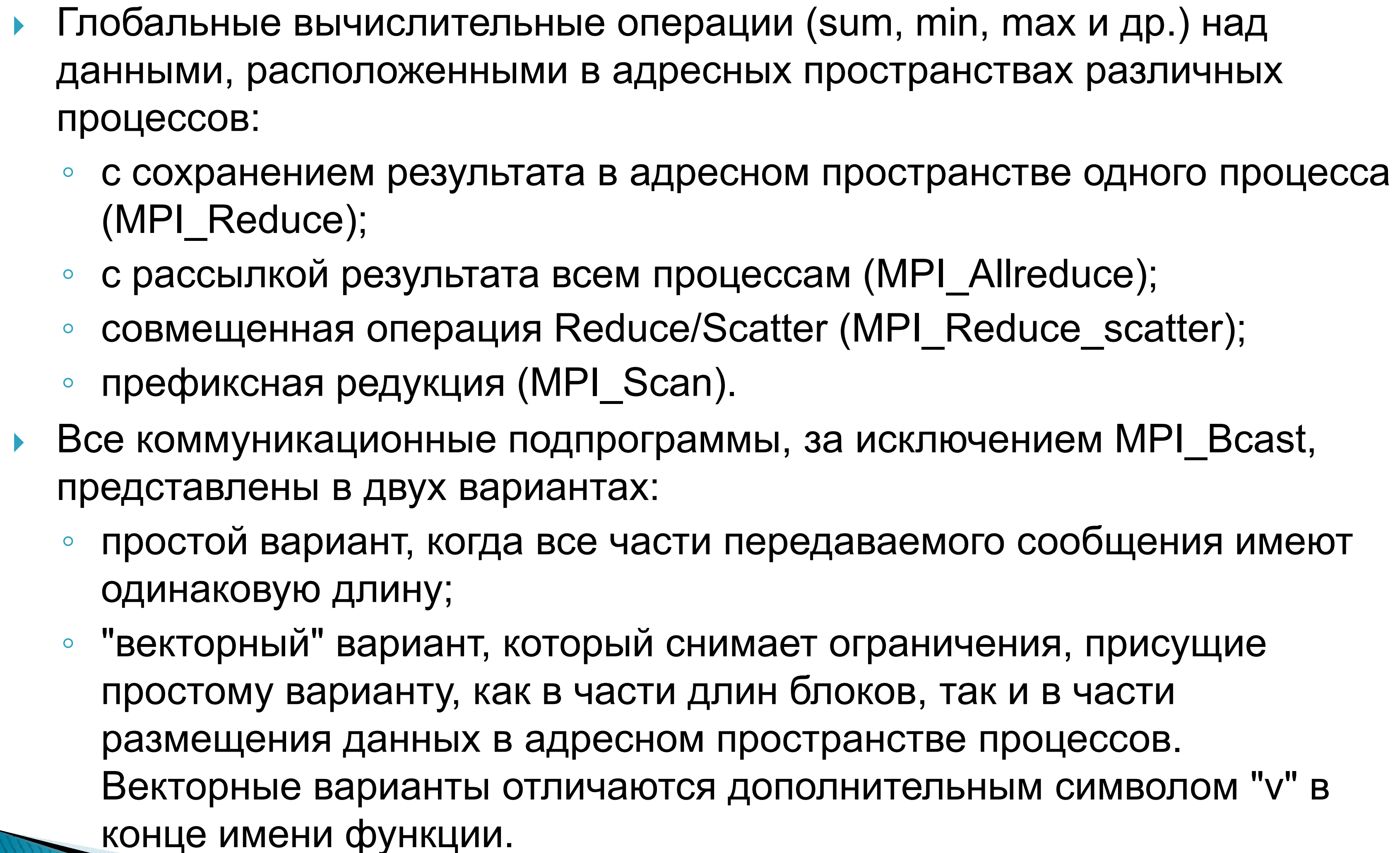
```
switch(rank) {
   case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
   case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```



Обзор коллективных операций

- Синхронизация всех процессов с помощью барьеров (MPI_Barrier).
- Коллективные коммуникационные операции, в число которых входят:
  - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
  - сборка распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
  - сборка распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
  - разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
  - совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).



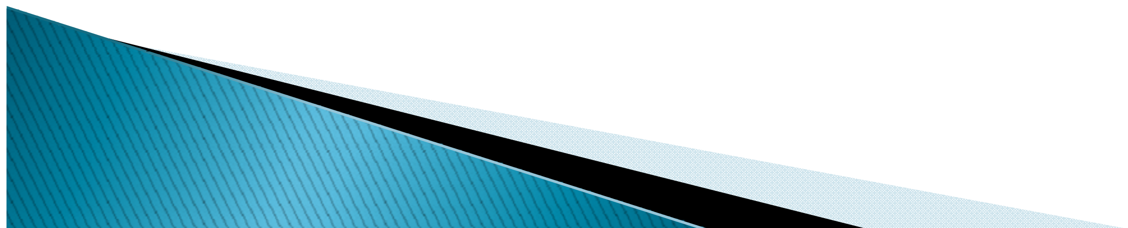

Обзор коллективных операций
Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
с рассылкой результата всем процессам (MPI_Allreduce);
совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
префиксная редукция (MPI_Scan).
Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:
простой вариант, когда все части передаваемого сообщения имеют одинаковую длину;
"векторный" вариант, который снимает ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

Широковещательная рассылка

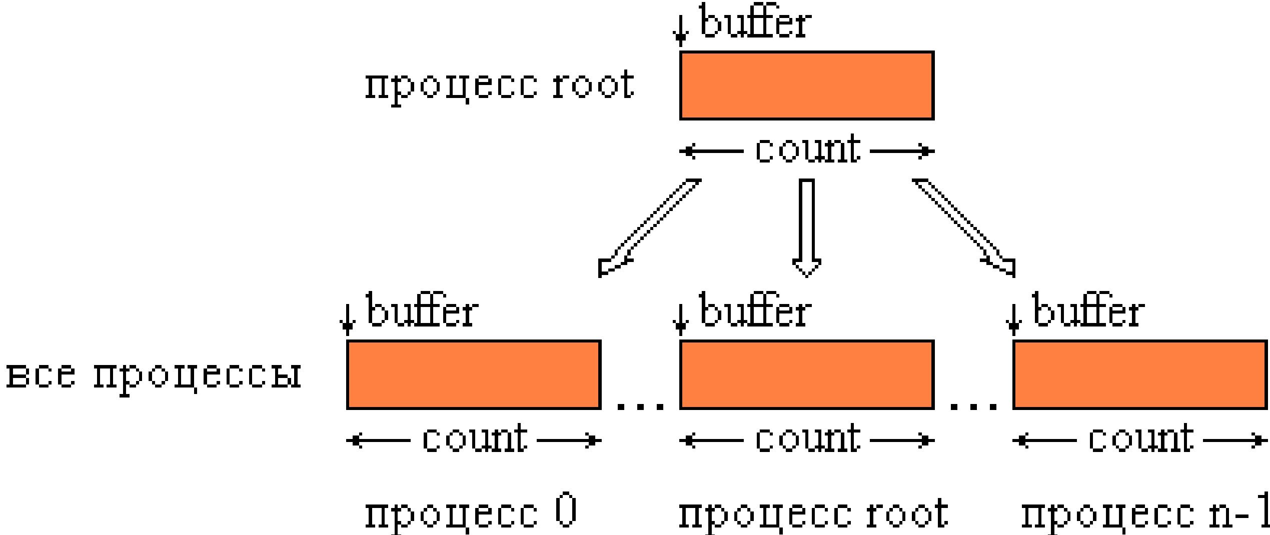
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

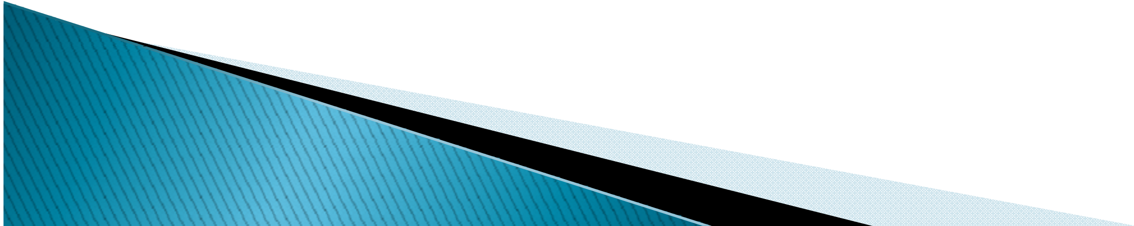
INOUT	buffer	- адрес начала расположения в памяти рассылаемых данных;
IN	count	- число посылаемых элементов;
IN	datatype	- тип посылаемых элементов;
IN	root	- номер процесса-отправителя;
IN	comm	- коммуникатор.

Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.



Широковещательная рассылка





Сбор блоков данных от всех процессов группы

Сборка блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root:

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

IN sendbuf - адрес начала размещения посылаемых данных;

IN sendcount - число посылаемых элементов;

IN sendtype - тип посылаемых элементов;

OUT recvbuf - адрес начала буфера приема (используется только в процессе-получателе root);

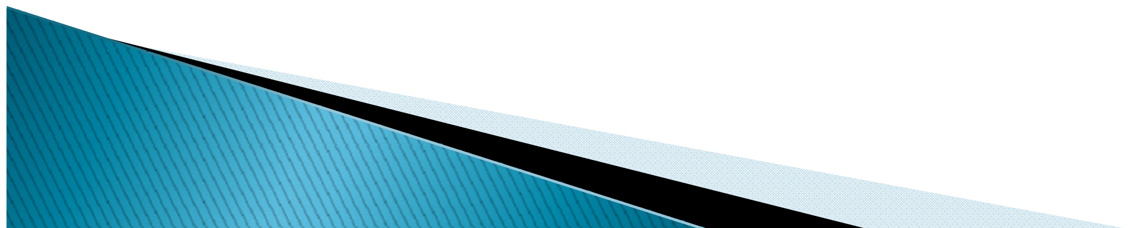
IN recvcount - число элементов, получаемых от каждого процесса (используется только в процессе-получателе root);

IN recvtype - тип получаемых элементов;

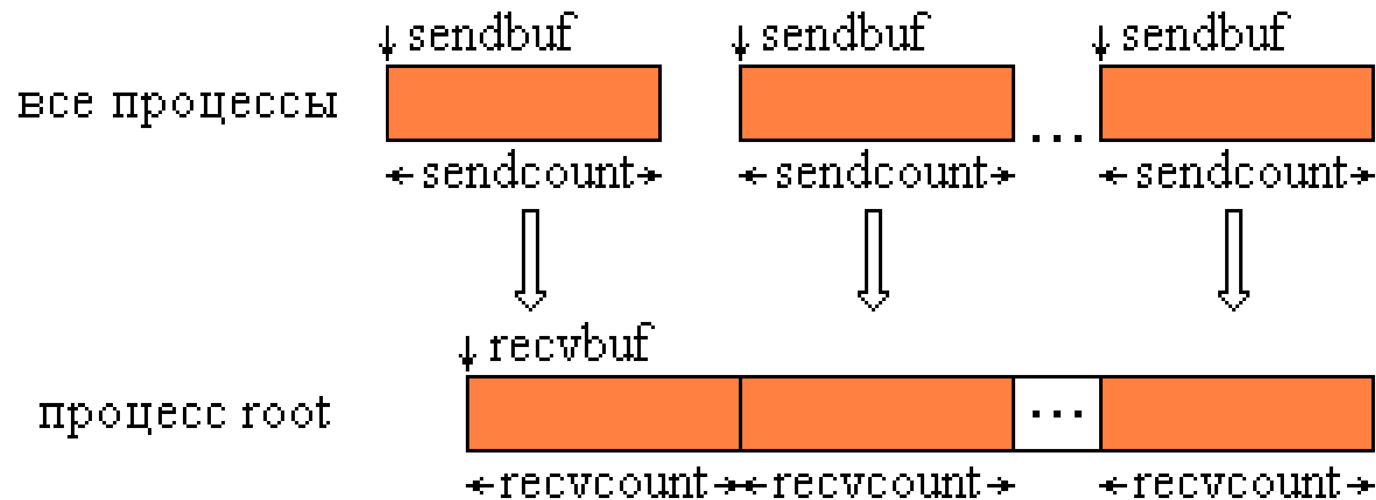
IN root - номер процесса-получателя; IN comm - коммуникатор.

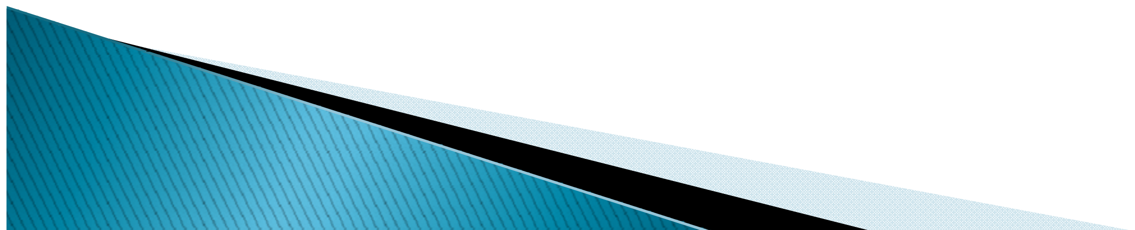
Сбор блоков данных от всех процессов группы

- Функция **MPI_Gather** производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root.
- Длина блоков предполагается одинаковой.
- Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть 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.



Сбор блоков данных от всех процессов группы





Сбор блоков данных от всех процессов группы

Функция **MPI_Gatherv** позволяет собирать блоки с разным числом элементов:

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

IN sendbuf - адрес начала буфера передачи;

IN sendcount - число посылаемых элементов;

IN sendtype - тип посылаемых элементов;

OUT rbuf - адрес начала буфера приема;

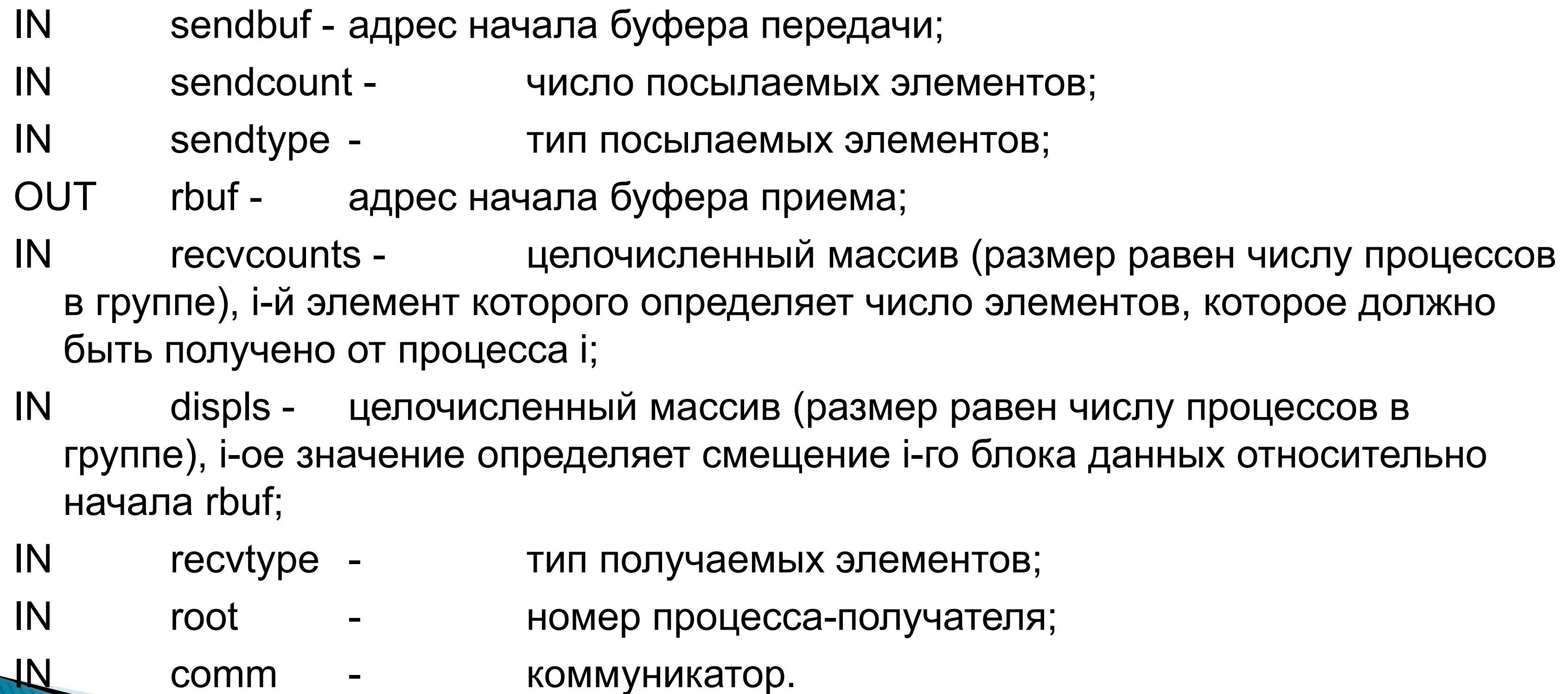
IN recvcounts - целочисленный массив (размер равен числу процессов в группе), i-й элемент которого определяет число элементов, которое должно быть получено от процесса i;

IN displs - целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение i-го блока данных относительно начала rbuf;

IN recvtype - тип получаемых элементов;

IN root - номер процесса-получателя;

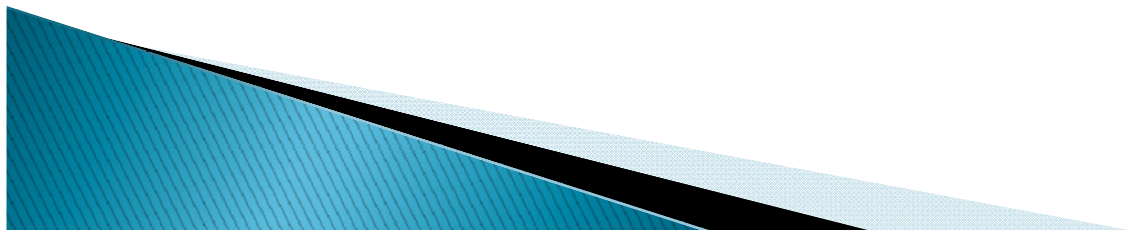
IN comm - коммуникатор.



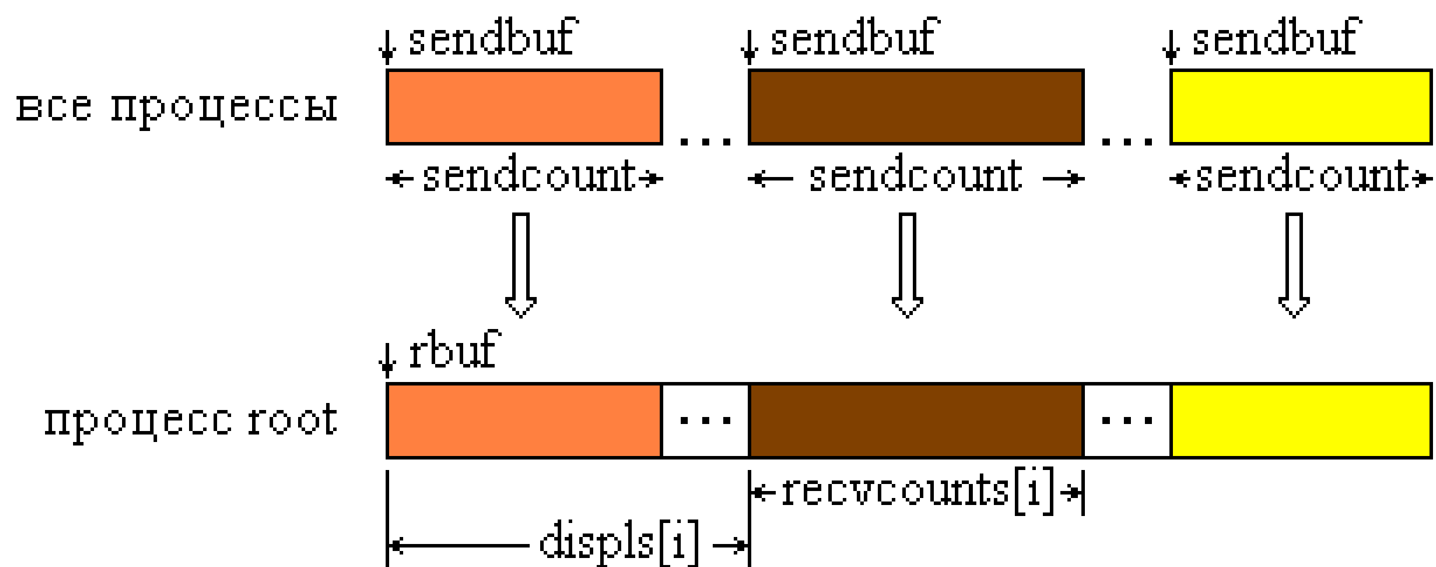
Сбор блоков данных от всех процессов группы

Функция **MPI_Gatherv** позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива recvcounts. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений displs.

Сообщения помещаются в буфер приема процесса root в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом i, размещаются в адресном пространстве процесса root, начиная с адреса rbuf + displs[i]



Сбор блоков данных от всех процессов группы



Сбор блоков данных от всех процессов группы

150 150 150

100 100 100

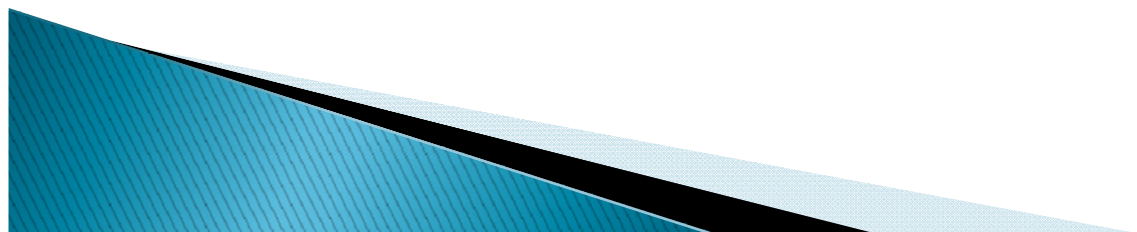
all processes

100 100 100

at root

stride

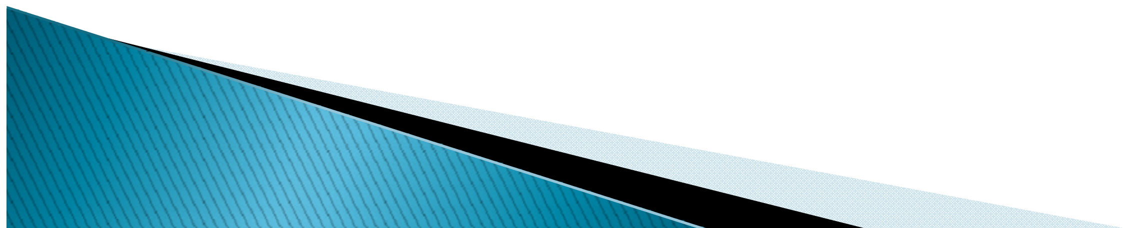
rbuf



Сбор блоков данных от всех процессов группы

```
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
   displs[i] = i*stride;
   rcounts[i] = 100;
}
/* Create datatype for 1 column of array */
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);

MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```



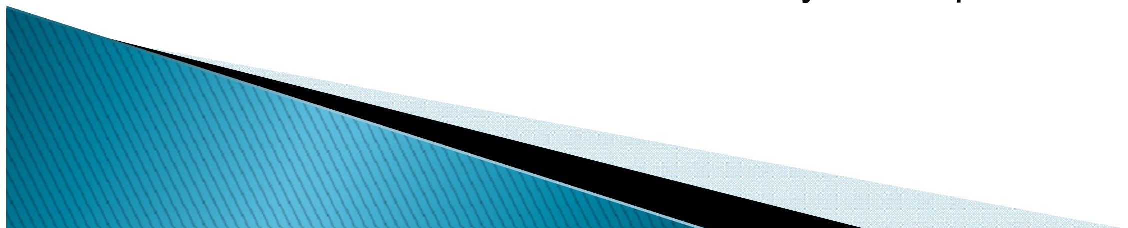
Сбор блоков данных от всех процессов группы

Функция **MPI_Allgather** выполняется так же, как **MPI_Gather**, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.

C:

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

IN	sendbuf	-	адрес начала буфера посылки;
IN	sendcount	-	число посылаемых элементов;
IN	sendtype	-	тип посылаемых элементов;
OUT	recvbuf	-	адрес начала буфера приема;
IN	recvcount	-	число элементов, получаемых от каждого процесса;
IN	recvtype	-	тип получаемых элементов;
IN	comm	-	коммуникатор.



Сбор блоков данных от всех процессов группы

i

		sendbuf		recvbuf		
процесс 0	i	a0		a0	a1	a2
		sendbuf	Allgather	recvbuf		
процесс 1		a1		a0	a1	a2
		sendbuf		recvbuf		
процесс 2		a2		a0	a1	a2

Сбор блоков данных от всех процессов группы

Функция **MPI_Allgatherv** является аналогом функции **MPI_Gatherv**, но сборка выполняется всеми процессами группы. Поэтому в списке параметров отсутствует параметр root.

int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,

void* rbuf, int *recvcounts, int *displs,

MPI_Datatype recvtype, MPI_Comm comm)

IN sendbuf - адрес начала буфера передачи;

IN sendcount - число посылаемых элементов;

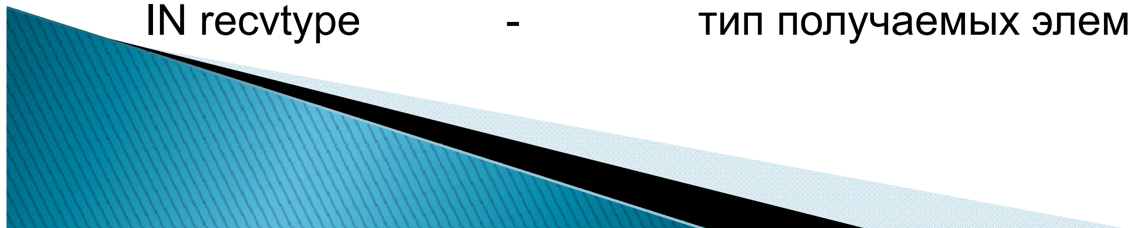
IN sendtype - тип посылаемых элементов;

OUT rbuf - адрес начала буфера приема;

IN recvcounts - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса;

IN displs - целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение относительно начала rbuf i-го блока данных;

IN recvtype - тип получаемых элементов; IN comm - коммуникатор.



Рассылка блоков данных по всем процессам группы

Функция **MPI_Scatter** разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,

void* recvbuf, int recvcount, MPI_Datatype recvtype,

int root, MPI_Comm comm)

IN sendbuf - адрес начала размещения блоков распределяемых данных (используется только в процессе-отправителе root);

IN sendcount - число элементов, посылаемых каждому процессу;

IN sendtype - тип посылаемых элементов;

OUT recvbuf - адрес начала буфера приема;

IN recvcount - число получаемых элементов;

IN recvtype - тип получаемых элементов;

IN root - номер процесса-отправителя;

IN comm - коммуникатор.

Рассылка блоков данных по всем процессам группы

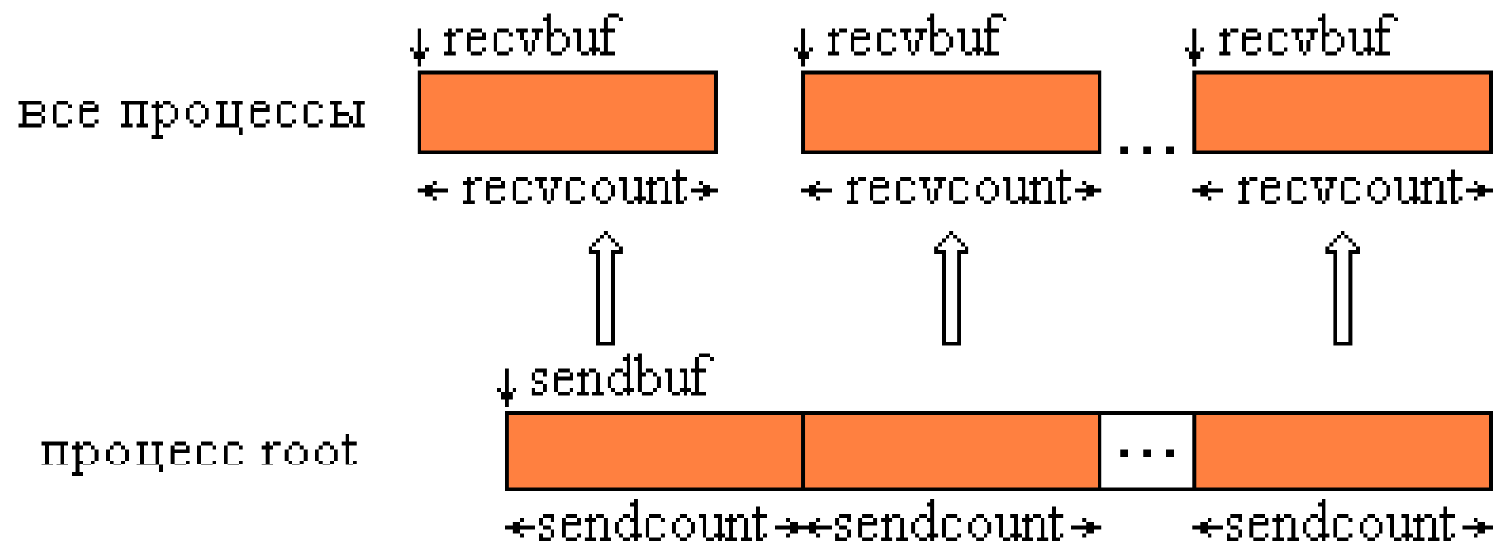
Функция **MPI_Scatter** разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посылает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

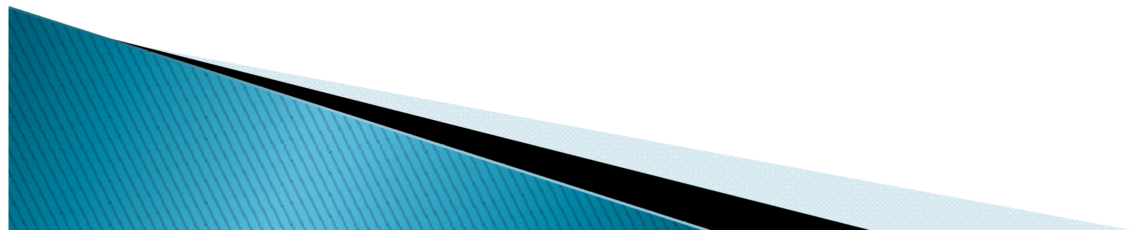
Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны.

Число посылаемых элементов sendcount должно равняться числу принимаемых recvcount.

Следует обратить внимание, что значение sendcount в вызове из процесса root - это число посылаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather.

Рассылка блоков данных по всем процессам группы





Рассылка блоков данных по всем процессам группы

Функция **MPI_Scaterv** является векторным вариантом функции **MPI_Scatter**, позволяющим посылать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого i-му процессу, задается в массиве смещений displs, а число посылаемых элементов - в массиве sendcounts.

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

IN sendbuf - адрес начала буфера посылки (используется только в процессе-отправителе root);

IN sendcounts - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу;

IN displs - целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение относительно начала sendbuf для данных, посылаемых процессу i;

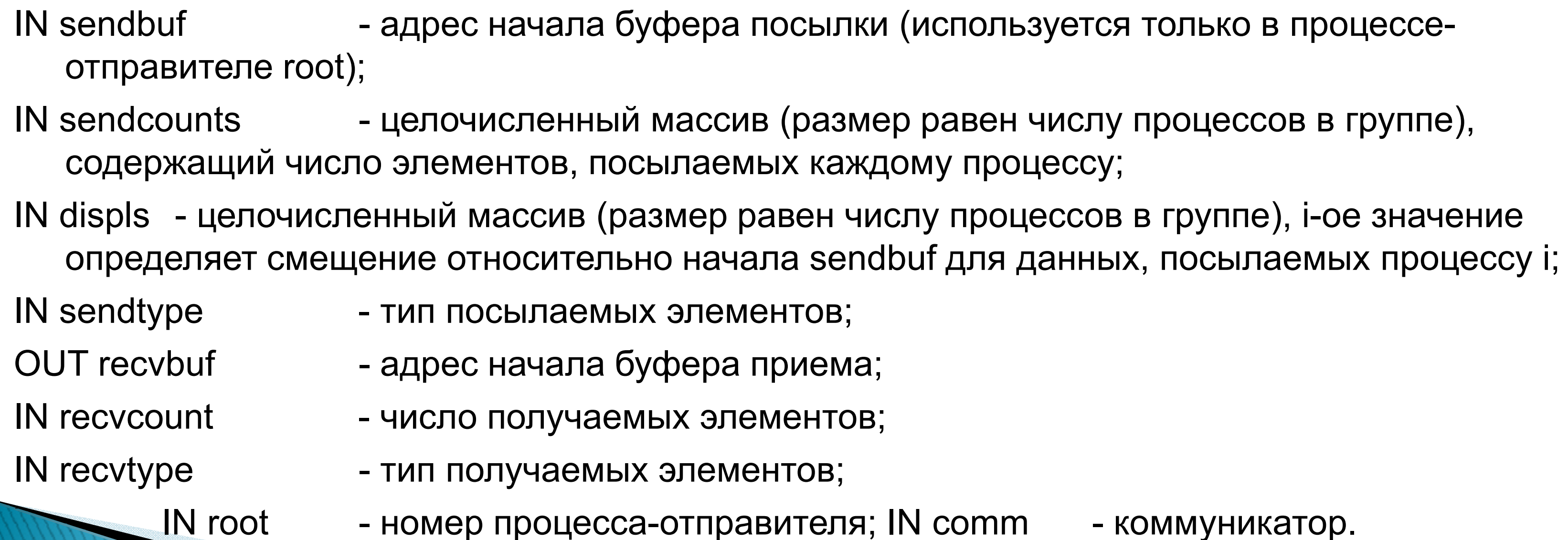
IN sendtype - тип посылаемых элементов;

OUT recvbuf - адрес начала буфера приема;

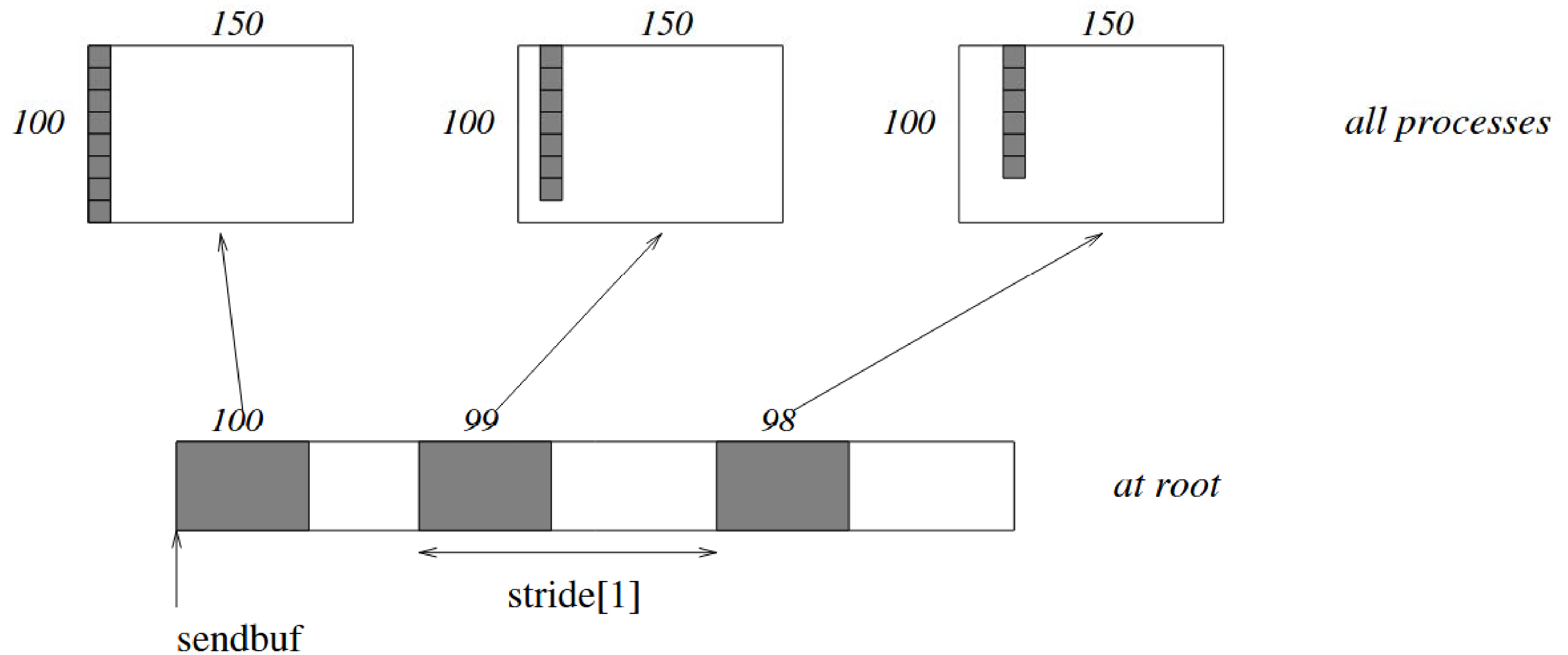
IN recvcount - число получаемых элементов;

IN recvtype - тип получаемых элементов;

IN root - номер процесса-отправителя; IN comm - коммуникатор.

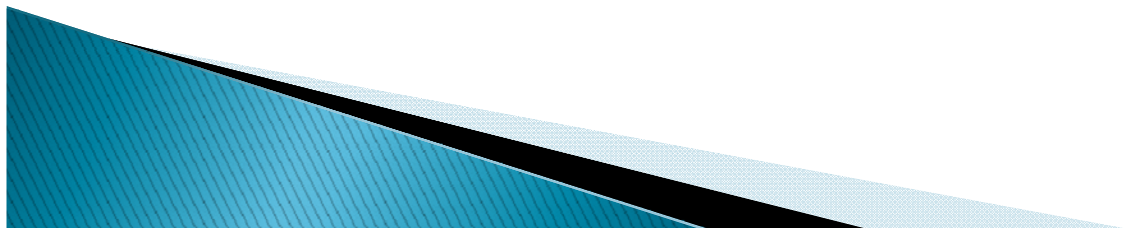


Рассылка блоков данных по всем процессам группы



Рассылка блоков данных по всем процессам группы

```
stride  = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow sendbuf comes from elsewhere */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype, root, comm);
```



Рассылка данных от всех процессов всем процессам

Функция **MPI_Alltoall** совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям. Процесс i посылает j-ый блок своего буфера sendbuf процессу j, который помещает его в i-ый блок своего буфера recvbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.

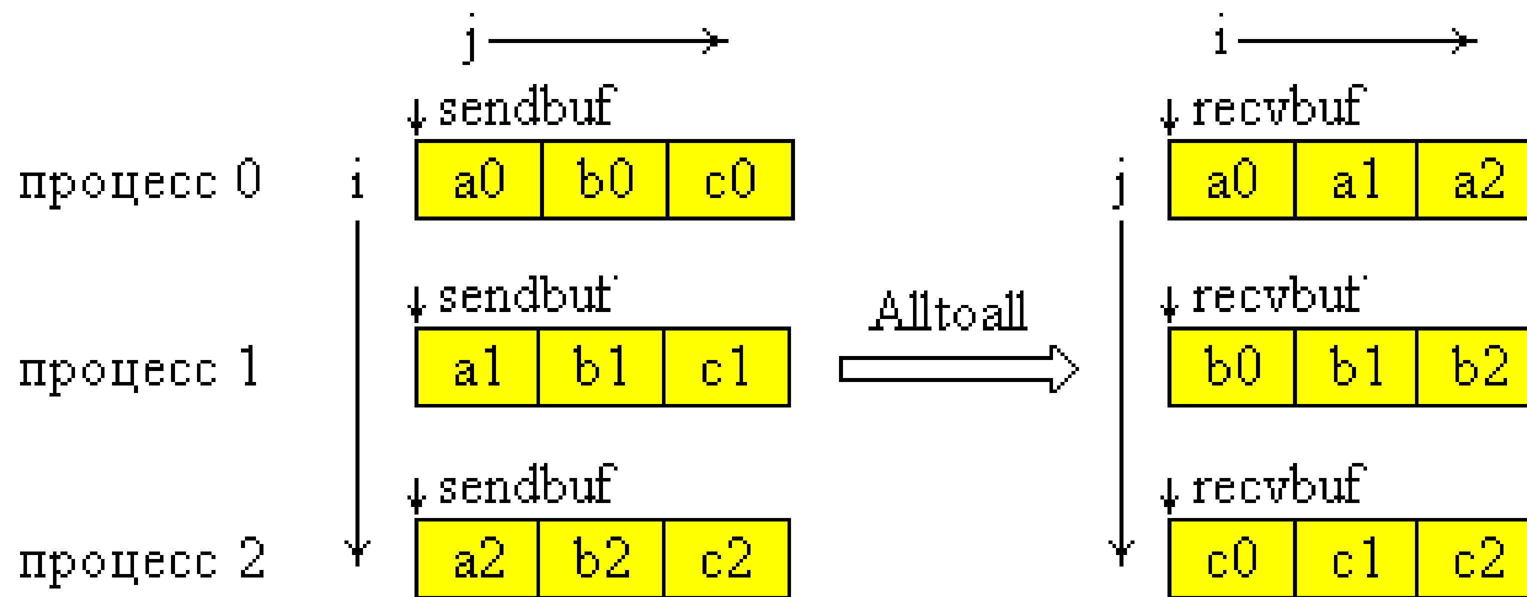
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,

void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

IN sendbuf	- адрес начала буфера посылки;
IN sendcount	- число посылаемых элементов;
IN sendtype	- тип посылаемых элементов;
OUT recvbuf	- адрес начала буфера приема;
IN recvcount	- число элементов, получаемых от каждого процесса;
IN recvtype	- тип получаемых элементов;
IN comm	- коммуникатор.

MPI_Alltoallv реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

Рассылка данных от всех процессов всем процессам

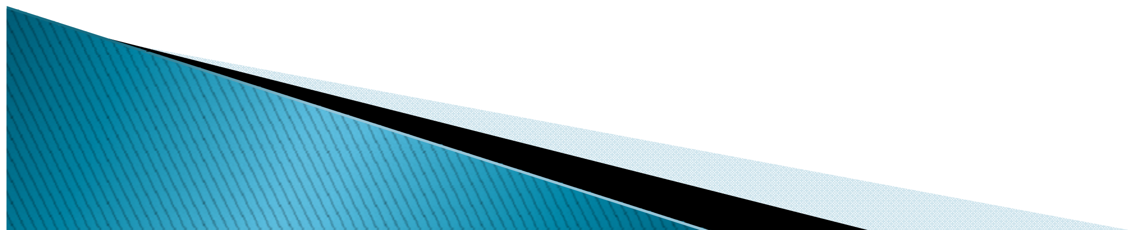


Глобальные вычислительные операции над распределенными данными

Операцией редукции называется операция, аргументом которой является вектор, а результатом - скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора распределенными по процессам коммуникатора.

Операции редукции в MPI представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce).
- с сохранением результата в адресном пространстве всех процессов (MPI_Allreduce).
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. i-я компонента этого вектора является результатом редукции первых i компонент распределенного вектора (MPI_Scan).
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter).



Глобальные вычислительные операции над распределенными данными

Функция **MPI_Reduce** выполняется следующим образом. Операция глобальной редукции, указанная параметром op, выполняется над первыми элементами входного буфера, и результат посылается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т.д.

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

IN sendbuf - адрес начала входного буфера;

OUT recvbuf - адрес начала буфера результатов (используется только в процессе-получателе root);

IN count - число элементов во входном буфере;

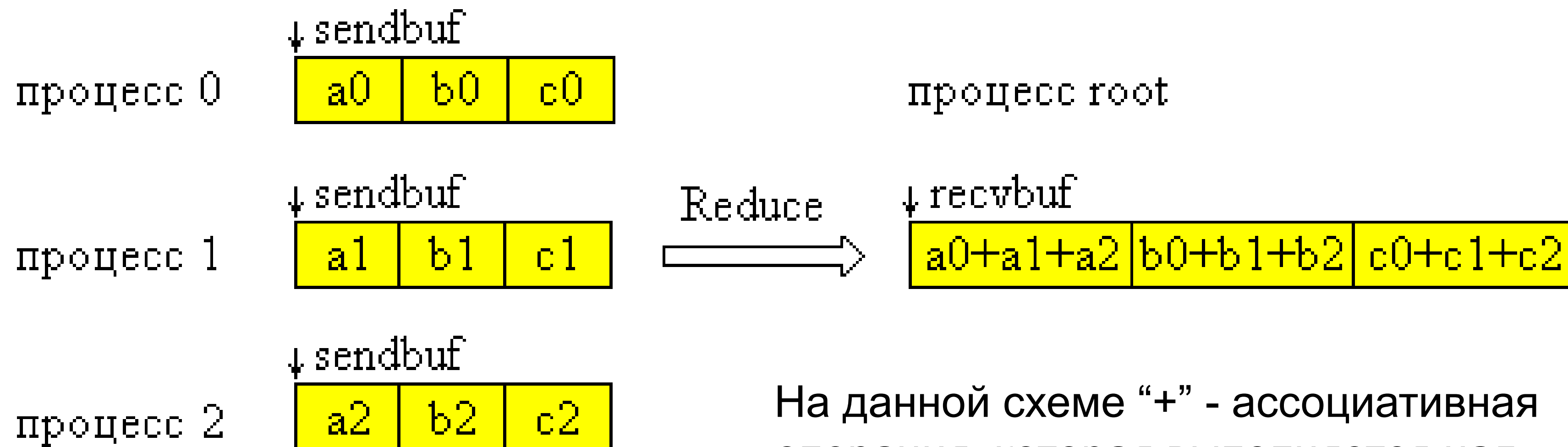
IN datatype - тип элементов во входном буфере;

IN op - операция, по которой выполняется редукция;

IN root - номер процесса-получателя результата операции;

IN comm - коммуникатор.

Глобальные вычислительные операции над распределенными данными



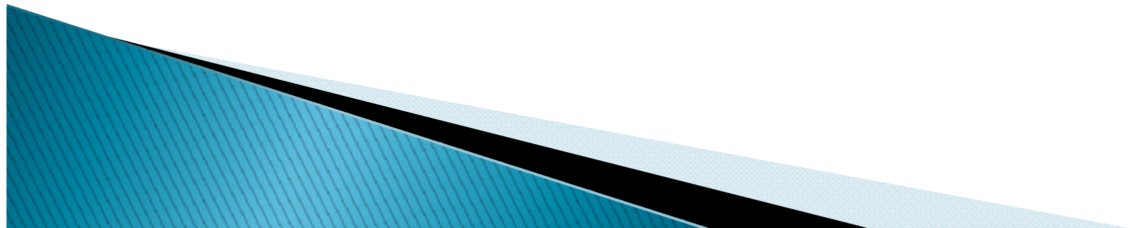
На данной схеме "+" - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Глобальные вычислительные операции над распределенными данными

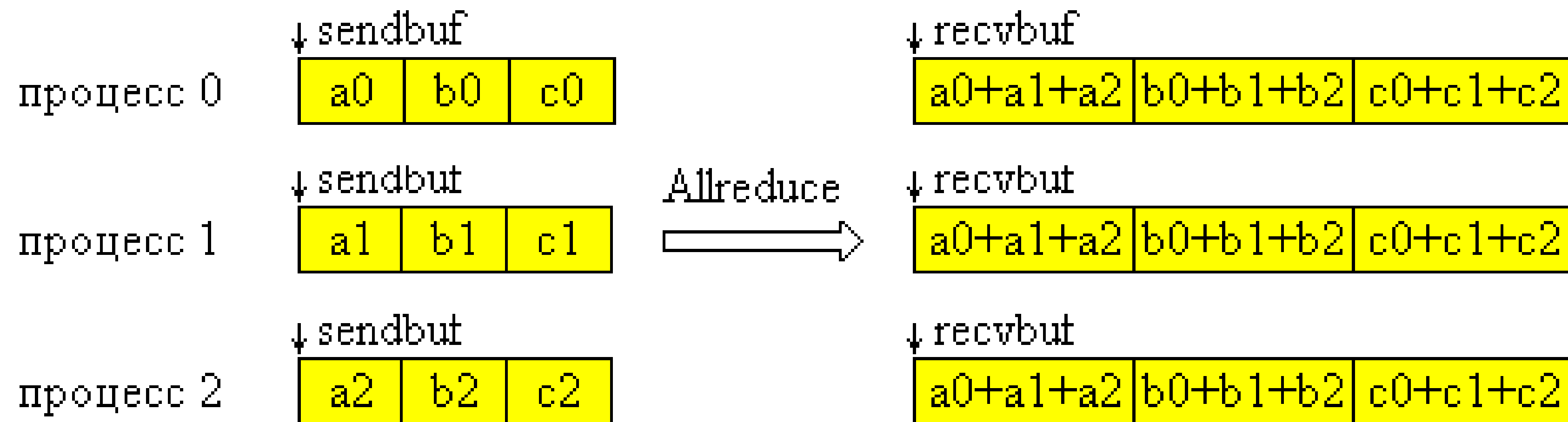
Функция **MPI_Allreduce** сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса root. В остальном, набор параметров такой же, как и в предыдущей функции.

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

IN sendbuf - адрес начала входного буфера;
OUT recvbuf - адрес начала буфера приема;
IN count - число элементов во входном буфере;
IN datatype - тип элементов во входном буфере;
IN op - операция, по которой выполняется редукция;
IN comm - коммуникатор.



Глобальные вычислительные операции над распределенными данными



На данной схеме "+" - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Глобальные вычислительные операции над распределенными данными

Функция **MPI_Reduce_scatter** совмещает в себе операции редукции и распределения результата по процессам.

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

IN sendbuf - адрес начала входного буфера;

OUT recvbuf - адрес начала буфера приема;

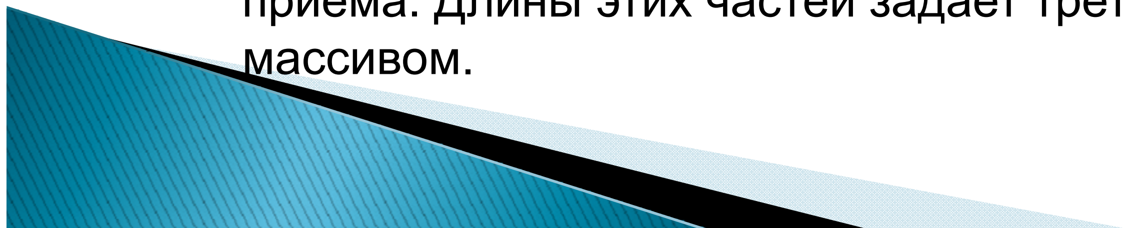
IN revcount - массив, в котором задаются размеры блоков, посылаемых процессам;

IN datatype - тип элементов во входном буфере;

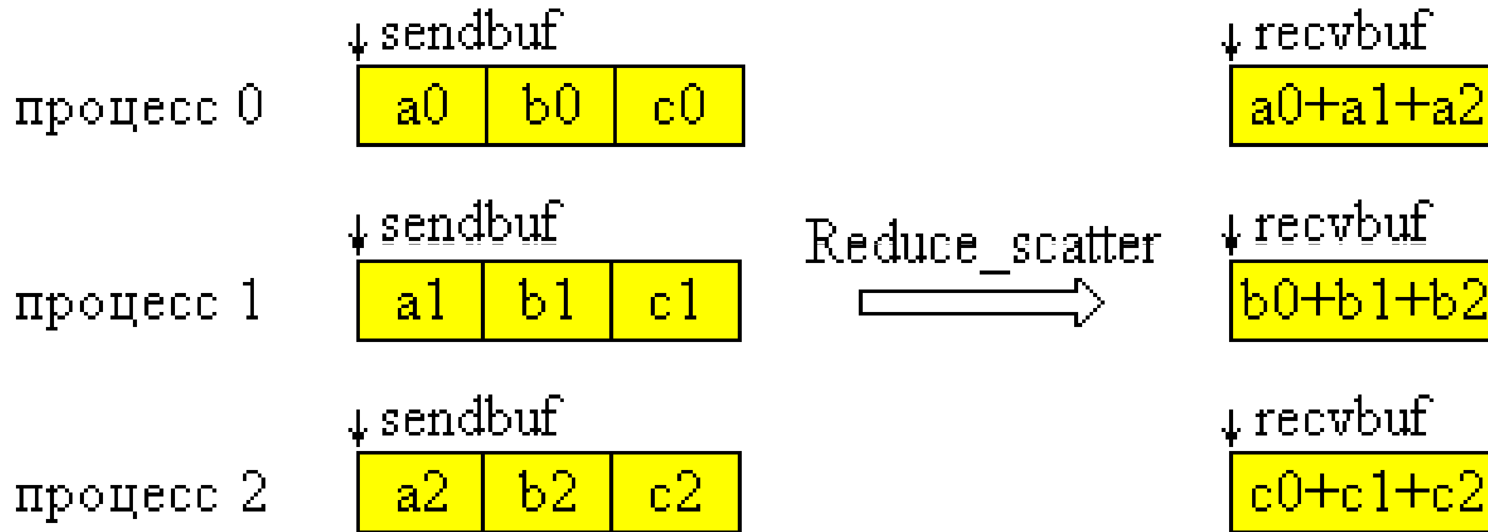
IN op - операция, по которой выполняется редукция;

IN comm - коммуникатор.

Функция **MPI_Reduce_scatter** отличается от **MPI_Allreduce** тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе, i-ая часть посылается i-ому процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом.



Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

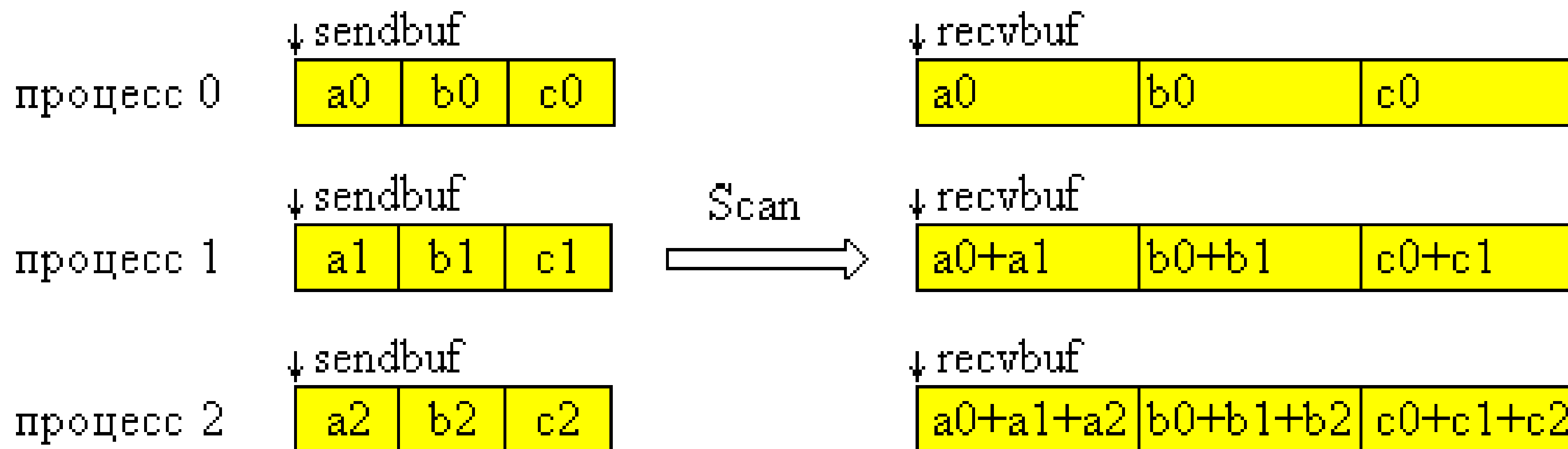
Глобальные вычислительные операции над распределенными данными

Функция **MPI_Scan** выполняет префиксную редукцию. Параметры такие же, как в **MPI_Allreduce**, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0, ... i включительно.

int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

IN sendbuf - адрес начала входного буфера
OUT recvbuf - адрес начала буфера приема
IN count - число элементов во входном буфере
IN datatype - тип элементов во входном буфере
IN op - операция, по которой выполняется редукция
IN comm - коммуникатор

Глобальные вычислительные операции над распределенными данными



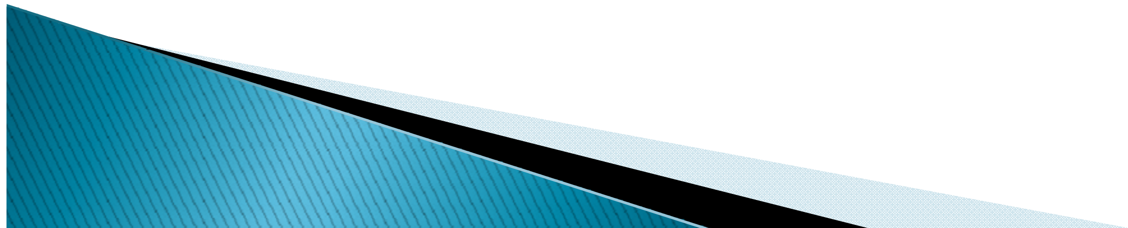
На данной схеме “+” - ассоциативная операция, которая выполняется над парой операндов типа datatype и возвращает результат того же типа: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

Коллективные асинхроные операции

- **MPI_Ibarrier**
- **MPI_Ibcast**
- **MPI_Igather**
- **MPI_Iscatter**
- **MPI_Iallgather**
- **MPI_Ialltoall**

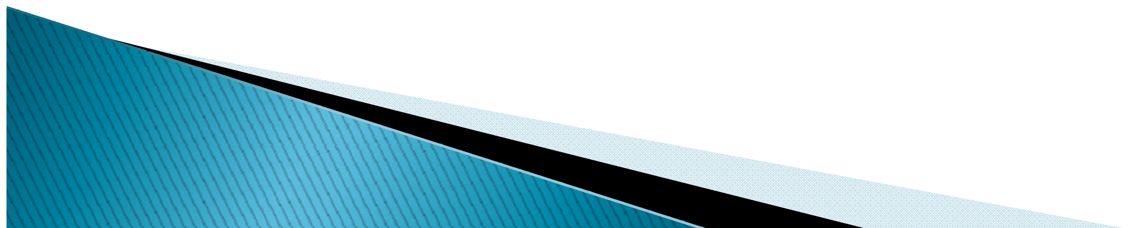
Редукционные операции

- **MPI_Ireduce**
- **MPI_Iallreduce**
- **MPI_Ireduce_scatter**
- **MPI_Iscan**
- ....

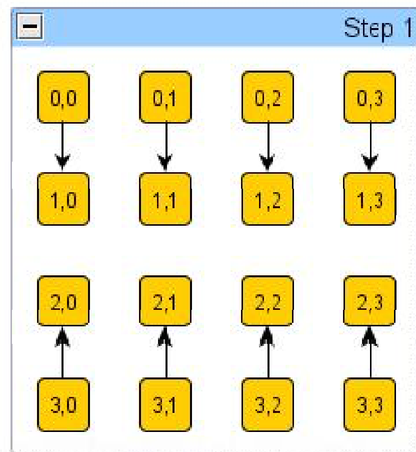


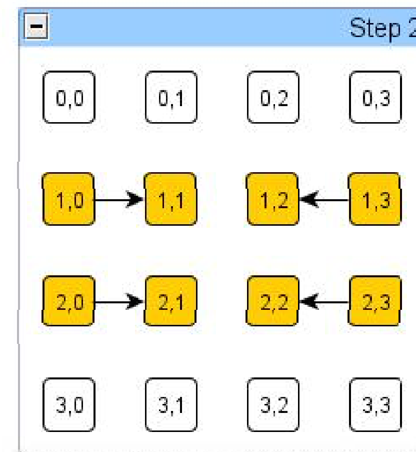
Коллективные асинхроные операции

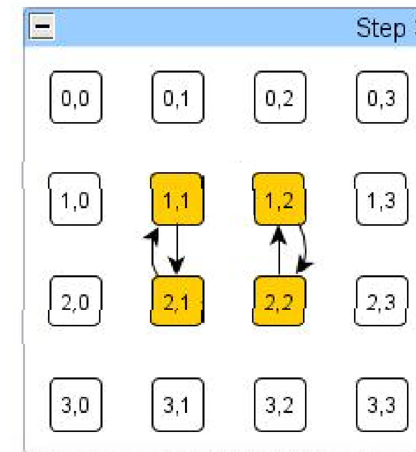
```
MPI_Request req;
switch(rank) {
  case 0:
    MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    break;
  case 1:
    MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
    break;
}
/* erroneous false matching of Alltoall and Ialltoall */
```

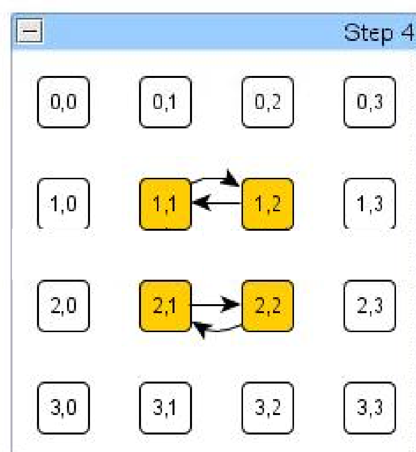


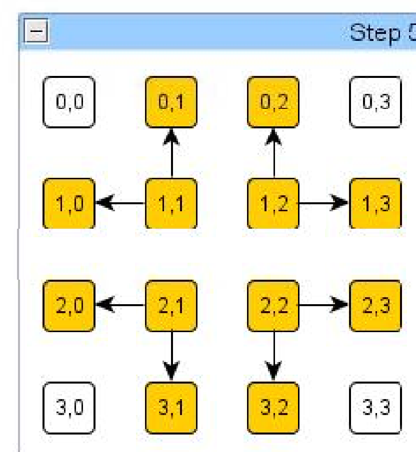
Реализация MPI_Allreduce

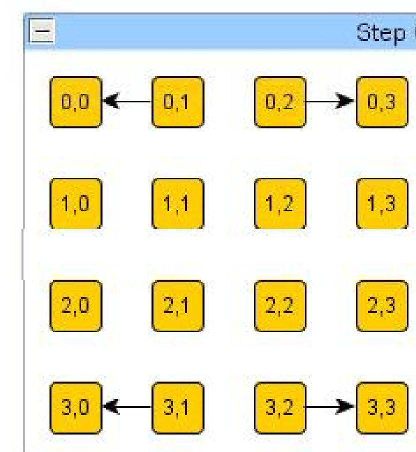


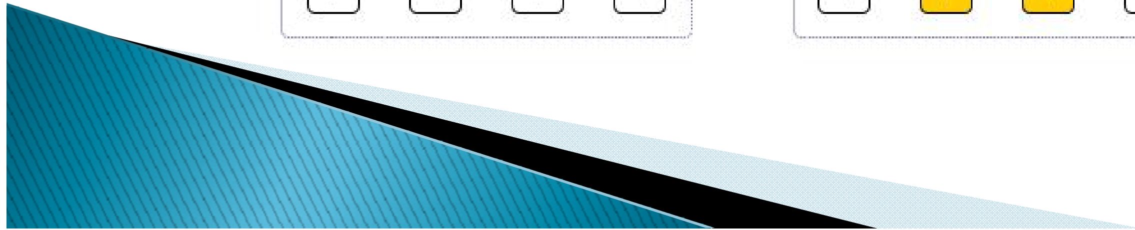




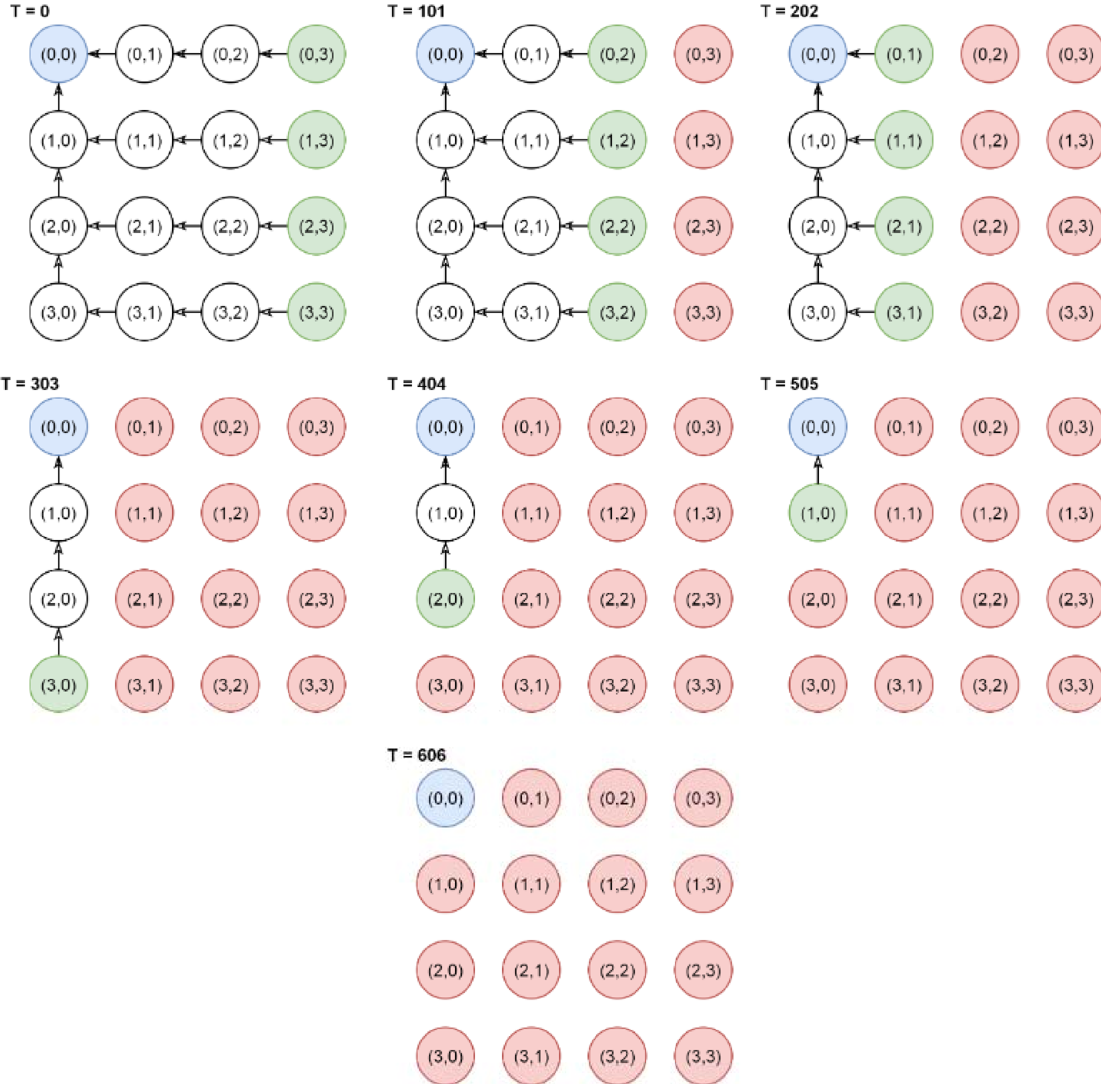


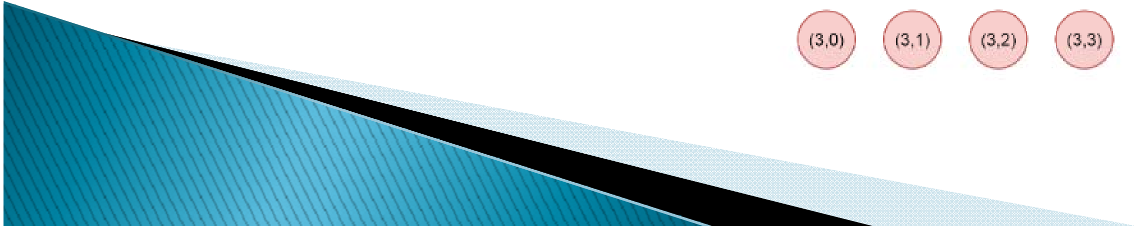




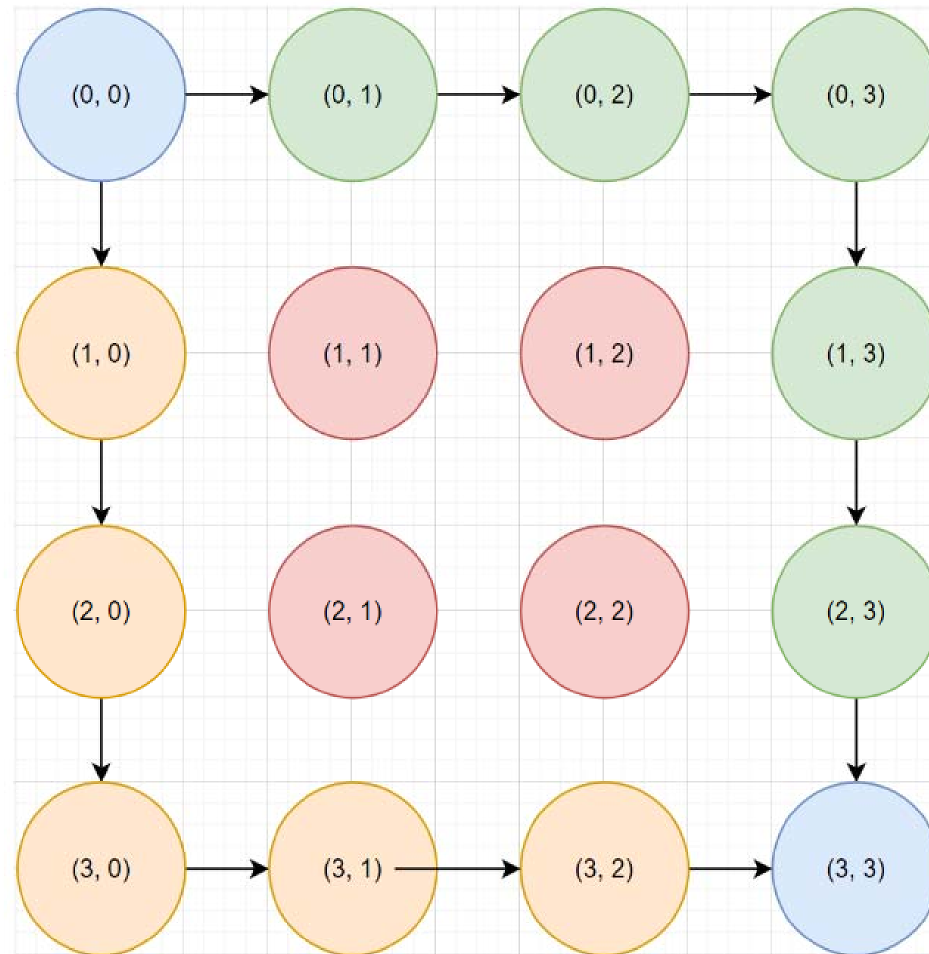


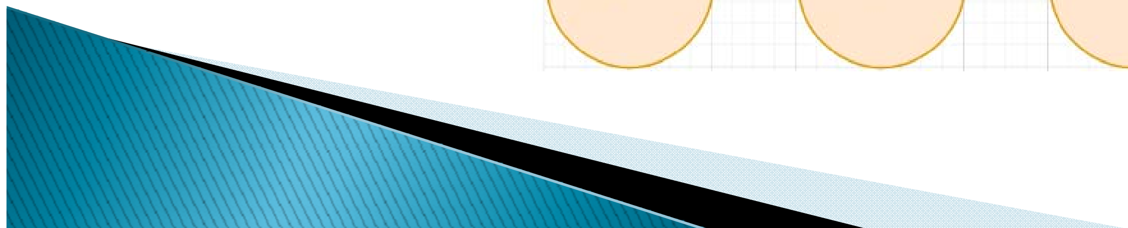
Реализация MPI_Gather





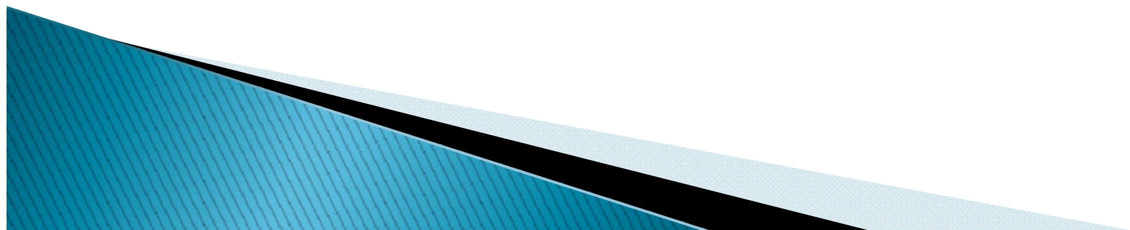
Передача длинного сообщения





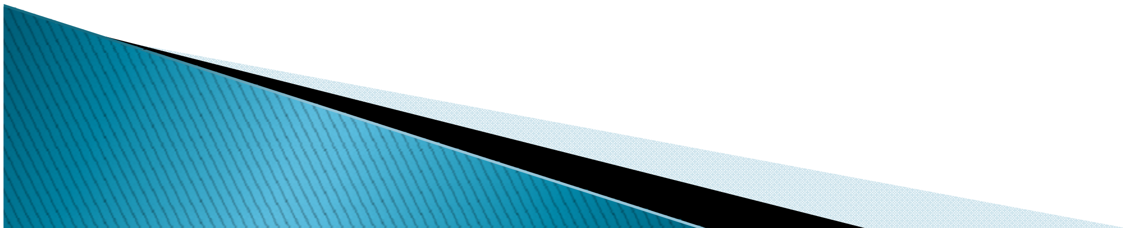
Partitioned Point-to-Point Communication

```
#include "mpi.h"
#define PARTITIONS 8
#define COUNT 5
int main(int argc, char *argv[])
{
double message[PARTITIONS*COUNT];
MPI_Count partitions = PARTITIONS;
int source = 0, dest = 1, tag = 1, flag = 0;
int myrank, i;
int provided;
MPI_Request request;
MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
if (provided < MPI_THREAD_SERIALIZED)
	MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```



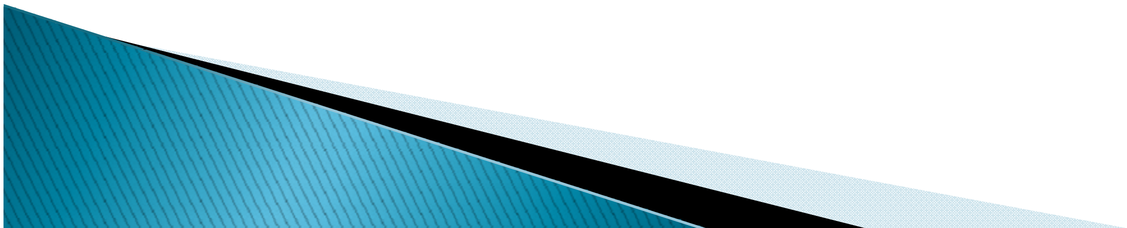
Partitioned Point-to-Point Communication

```
if (myrank == 0)
  {
    MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
        MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    for(i = 0; i < partitions; ++i)
    {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
    }
    while(!flag)
    {
        /* do useful work #1 */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
    }
    MPI_Request_free(&request);
 }
```

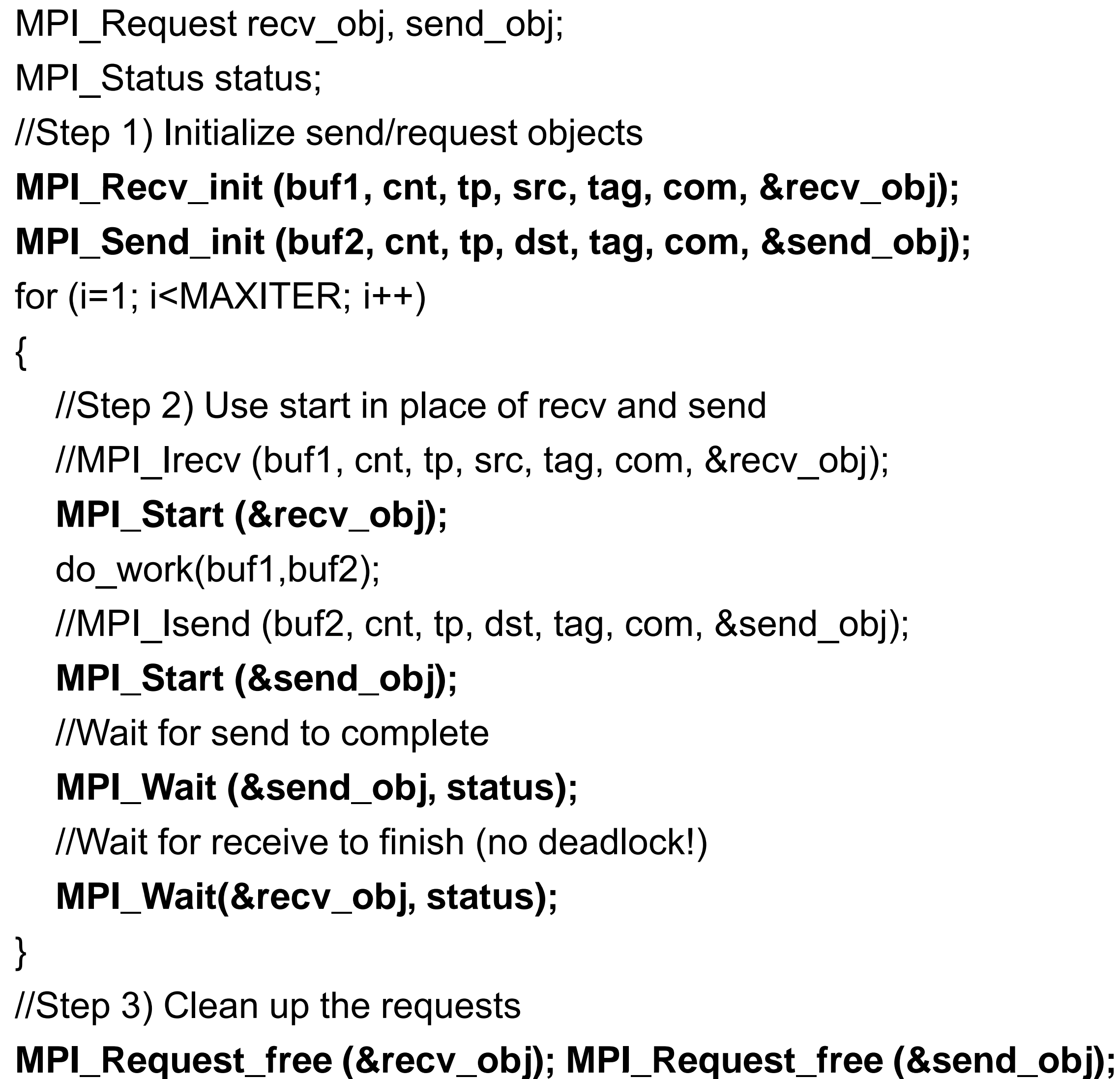


Partitioned Point-to-Point Communication

```
    else if (myrank == 1)
    {
        MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
            MPI_COMM_WORLD, MPI_INFO_NULL, &request);
        MPI_Start(&request);
        while(!flag)
        {
            /* do useful work #1 */
           MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
           /* do useful work #2 */
        }
        MPI_Request_free(&request);
    }
MPI_Finalize();
return 0;
}
```

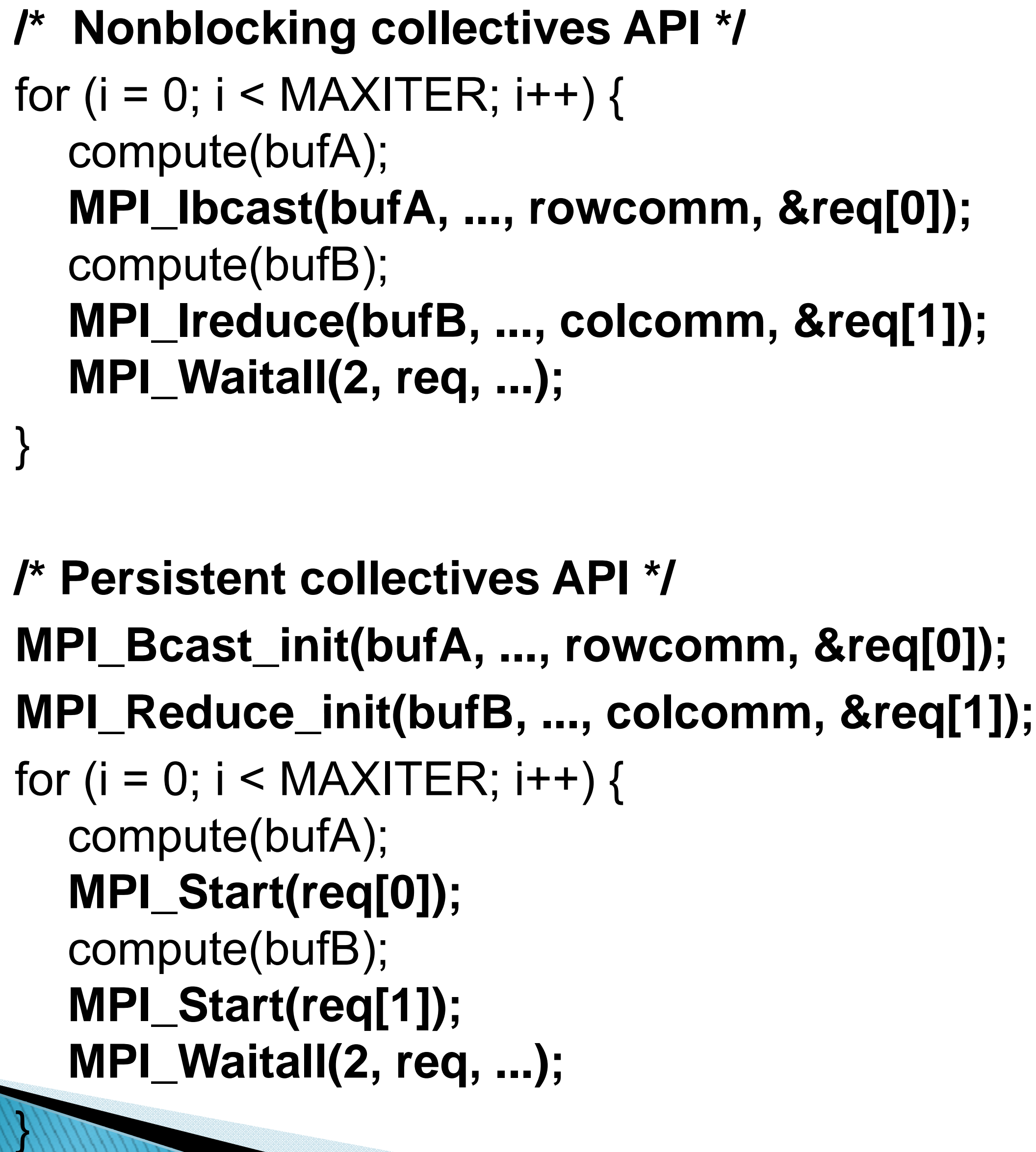


Point-to-Point Persistent Communication



```
MPI_Request recv_obj, send_obj;
MPI_Status status;
//Step 1) Initialize send/request objects
MPI_Recv_init (buf1, cnt, tp, src, tag, com, &recv_obj);
MPI_Send_init (buf2, cnt, tp, dst, tag, com, &send_obj);
for (i=1; i<MAXITER; i++)
{
   //Step 2) Use start in place of recv and send
   //MPI_Irecv (buf1, cnt, tp, src, tag, com, &recv_obj);
   MPI_Start (&recv_obj);
   do_work(buf1,buf2);
   //MPI_Isend (buf2, cnt, tp, dst, tag, com, &send_obj);
   MPI_Start (&send_obj);
   //Wait for send to complete
   MPI_Wait (&send_obj, status);
   //Wait for receive to finish (no deadlock!)
   MPI_Wait(&recv_obj, status);
}
//Step 3) Clean up the requests
MPI_Request_free (&recv_obj); MPI_Request_free (&send_obj);
```

Collective Persistent Communication

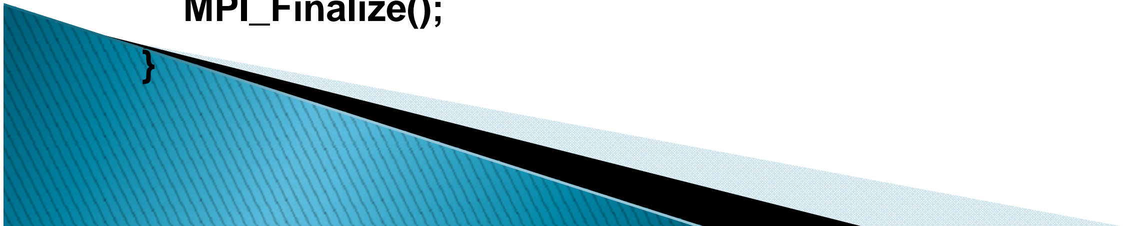


```
/*  Nonblocking collectives API */
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Ibcast(bufA, …, rowcomm, &req[0]);
    compute(bufB);
    MPI_Ireduce(bufB, …, colcomm, &req[1]);
    MPI_Waitall(2, req, …);
}

/* Persistent collectives API */
MPI_Bcast_init(bufA, …, rowcomm, &req[0]);
MPI_Reduce_init(bufB, …, colcomm, &req[1]);
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Start(req[0]);
    compute(bufB);
    MPI_Start(req[1]);
    MPI_Waitall(2, req, …);
}
```

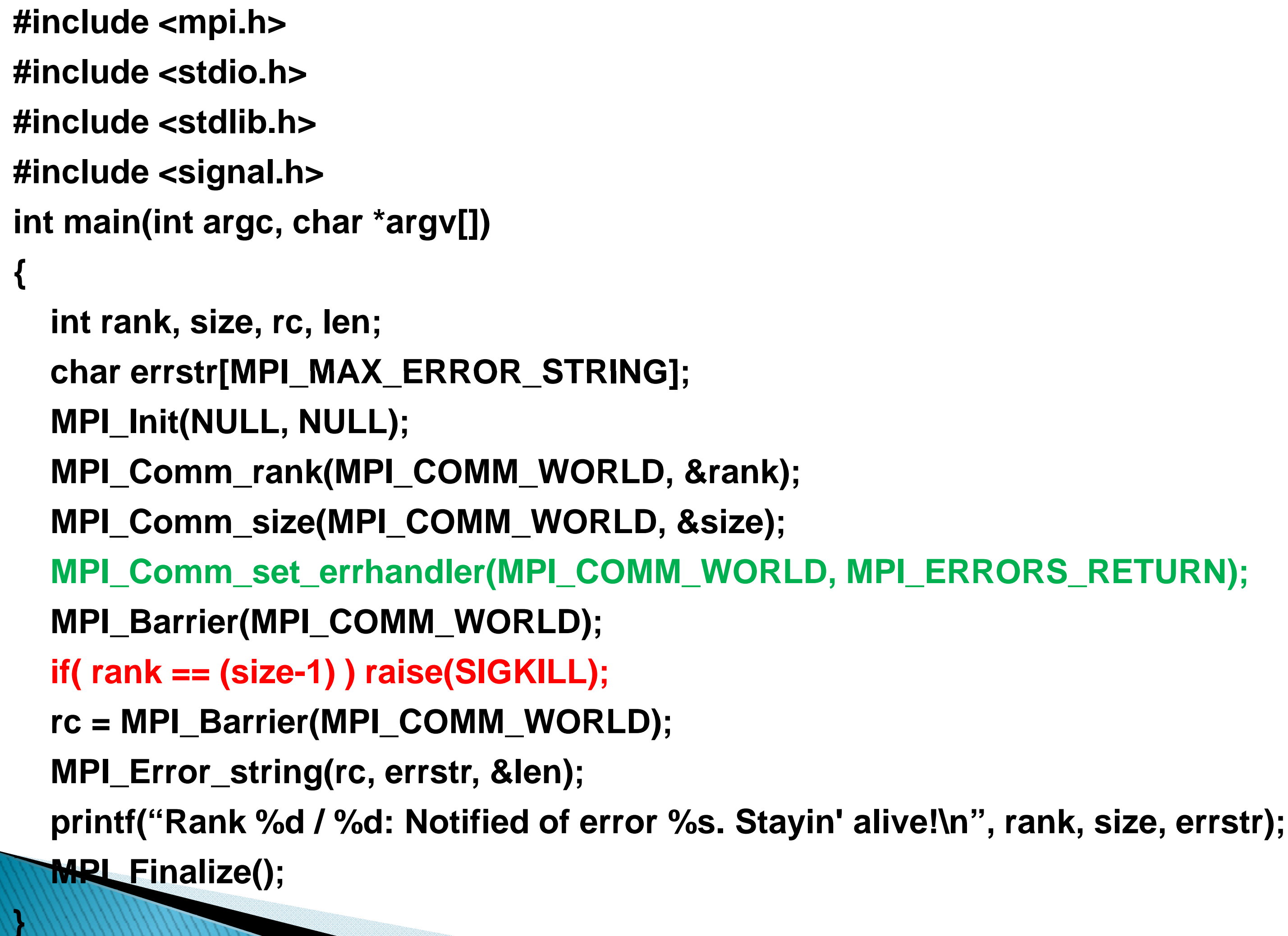
Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf(“Rank %d / %d: Notified of error %s. Stayin' alive!\n”, rank, size, errstr);
    MPI_Finalize();
}
```



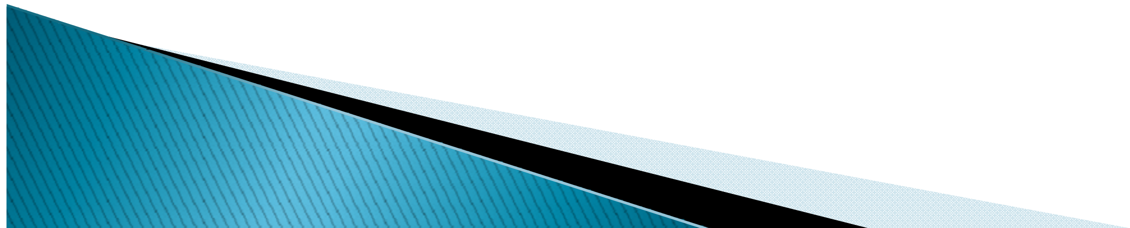
Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```



Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
   int rank, size, len;
   char errstr[MPI_MAX_ERROR_STRING];
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Error_string( *err, errstr, &len );
   printf("Rank %d / %d: Notified of error %s\n",
          rank, size, errstr);
}
```



Моделирование сбоя во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
    MPI_Finalize();
}
```