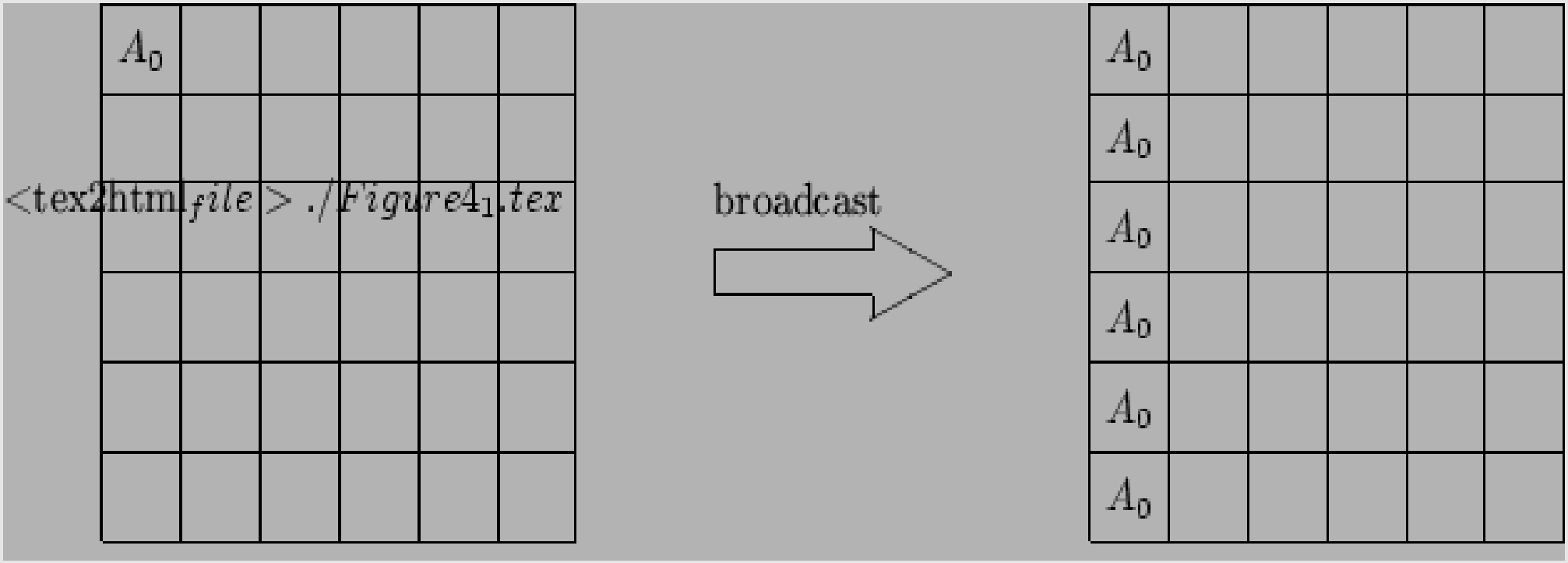
MPI: коллективные операции

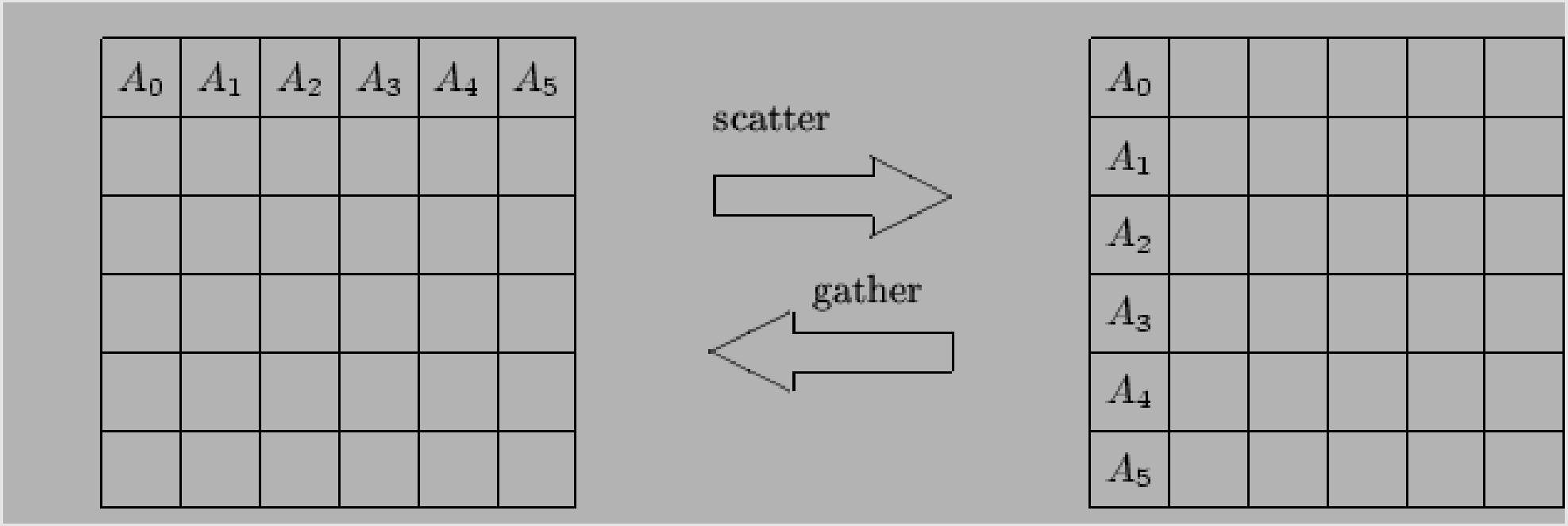
Коллективные операции

- Барьерная синхронизация всех процессов группы
- Широковещательная передача (broadcast) от одного процесса всем остальным процессам группы
- Сбор данных из всех процессов группы в один процесс
- Рассылка данных одним процессом группы всем процессам группы
- Вариант сбора данных, когда все процессы группы получают результат
- Раздача / сбор данных из всех процессов группы всем процессам группы (полный обмен или all-to-all)
- Вычислительные операции

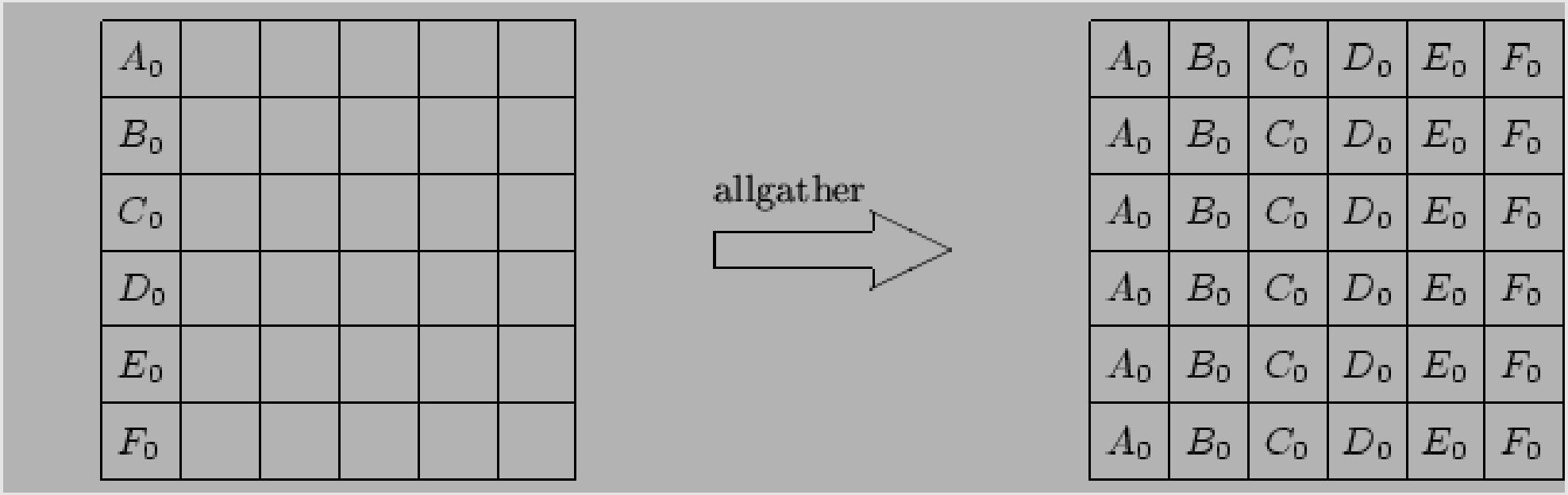
Broadcast



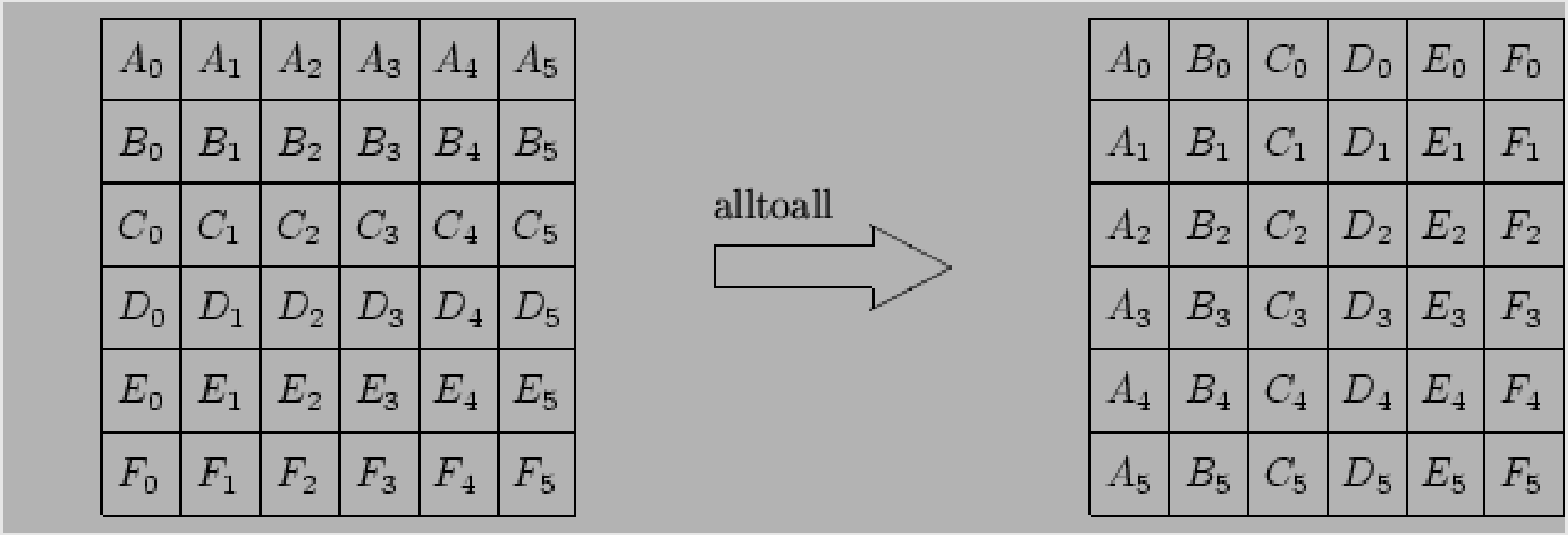
Scatter/gather



Allgather


A_0
B_0
C_0
D_0
E_0
F_0
allgather
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0
A_0 B_0 C_0 D_0 E_0 F_0

All-to-All


A_0 A_1 A_2 A_3 A_4 A_5
B_0 B_1 B_2 B_3 B_4 B_5
C_0 C_1 C_2 C_3 C_4 C_5
D_0 D_1 D_2 D_3 D_4 D_5
E_0 E_1 E_2 E_3 E_4 E_5
F_0 F_1 F_2 F_3 F_4 F_5
alltoall
A_0 B_0 C_0 D_0 E_0 F_0
A_1 B_1 C_1 D_1 E_1 F_1
A_2 B_2 C_2 D_2 E_2 F_2
A_3 B_3 C_3 D_3 E_3 F_3
A_4 B_4 C_4 D_4 E_4 F_4
A_5 B_5 C_5 D_5 E_5 F_5

Вызов коллективной функции может (но это не требуется) возвращать управление сразу, как только его участие в коллективной операции завершено.

Завершение вызова показывает, что процесс-отправитель уже может обращаться к буферу обмена.

Это однако не означает, что другие процессы в группе завершили операцию.

Вызов операции коллективного обмена *может* иметь эффект синхронизации всех процессов в группе.

С другой стороны, корректная переносимая программа должна учитывать факт, что коллективная операция может быть синхронизирована. Нельзя полагаться на этот побочный эффект синхронизации, но необходимо учитывать, что он может иметь место.

Барьерная синхронизация

MPI_BARRIER(comm)
IN comm коммуникатор(дескриптор)

Функция барьерной синхронизации MPI_BARRIER блокирует вызывающий процесс, пока все процессы группы не вызовут её.

Барьерная синхронизация

```
int MPI_Barrier(MPI_Comm comm

void MPI::Intracomm::Barrier() const
```

Барьерная синхронизация

MPI_BARRIER(comm)
IN comm коммуникатор(дескриптор)

Функция барьерной синхронизации MPI_BARRIER блокирует вызывающий процесс, пока все процессы группы не вызовут её.

Широковещательный обмен

Функция широковещательной передачи MPI_BCAST посылает сообщение из корневого процесса всем процессам группы, включая себя.

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT buffer адрес начала буфера
IN count количество записей в буфере (целое)
IN datatype тип данных в буфере (дескриптор)
IN root номер корневого процесса (целое)
IN comm коммуникатор (дескриптор)

Широковещательный обмен

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

MPI_BCAST(BUFFER, COUNT, DATATYPE,
          ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM,
IERROR

void MPI::Intracomm::Bcast(void* buffer, int count,
const Datatype& datatype, int root) const
```

Пример использования MPI_BCAST

Широковещательная передача 100 целых чисел от процесса 0 каждому процессу в группе.

```
MPI_Comm comm;
int array[100];
int root = 0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

Сборка данных

При выполнении операции сборки данных MPI_GATHER каждый процесс, включая корневой, посылает содержимое своего буфера в корневой процесс. Корневой процесс получает сообщения, располагая их в порядке возрастания номеров процессов.

Поведение, как если бы все процессы выполнили:

```
MPI_Send(sendbuf, sendcount, sendtype, root, ...)
```

А корневой процесс:

```
MPI_Recv(recvbuf + i * recvcount * extent(recvtype),
         recvcount, recvtype, i, ...)
```

Сборка данных

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

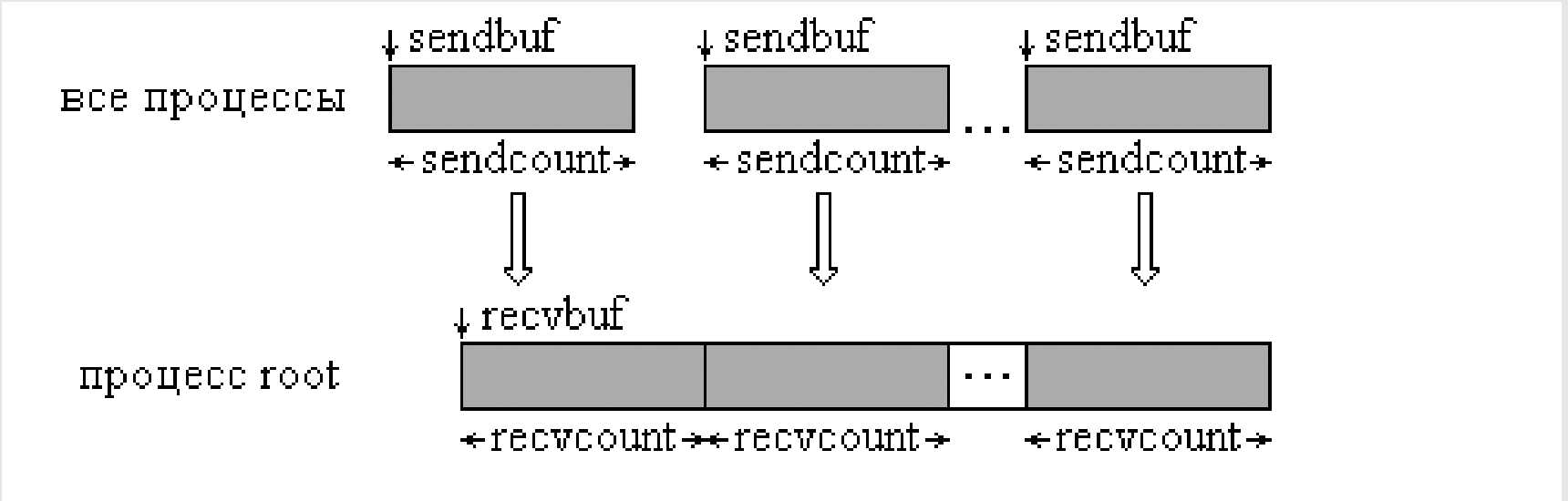
IN sendbuf адрес буфера процесса-отправителя
IN sendcount количество элементов в отсылаемом сообщении
IN sendtype тип элементов в отсылаемом сообщении
OUT recvbuf адрес буфера процесса сборки данных
IN recvcount количество элементов в принимаемом сообщении
IN recvtype тип данных элементов в буфере процесса-получателя
IN root номер процесса-получателя

Сборка данных

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

void MPI::Intracomm::Gather(const void* sendbuf,
 int sendcount, const Datatype& sendtype,
 void* recvbuf, int recvcount, const Datatype& recvtype,
 int root) const
```

Сборка данных



MPI_GATHERV

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

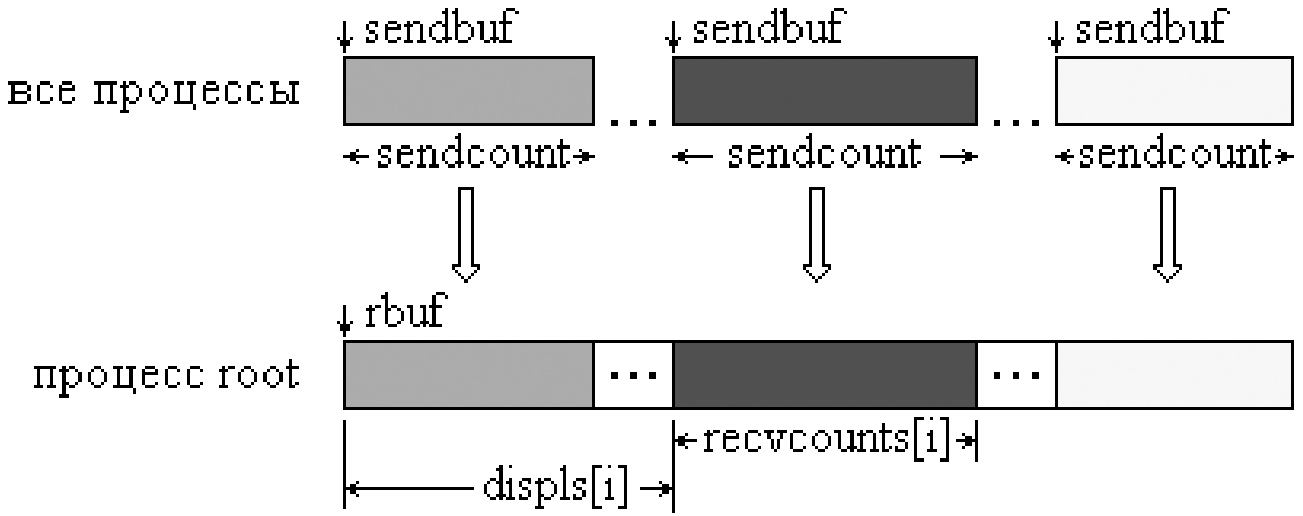
…..

IN recvcounts массив целых чисел (по размеру группы), содержащий количества элементов, которые получены от каждого из процессов (используется только корневым процессом)

IN displs массив целых чисел (по размеру группы). Элемент i определяет смещение относительно recvbuf, в котором размещаются данные из процесса i (используется только корневым процессом)

……

MPI_GATHERV


sendbuf
sendbuf
sendbuf
все процессы
sendcount
sendcount
sendcount
rbuf
процесс root
recvcounts[i]
displs[i]

Пример сбора данных

Сбор 100 целых чисел с каждого процесса группы в корневой процесс

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100,
    MPI_INT, root, comm);
```

Рассылка данных

Операция MPI_SCATTER обратна операции MPI_GATHER. Результат ее выполнения таков, как если бы корневой процесс выполнил n операций посылки

```
MPI_Send(senbuf + i * extent(sendtype), sendcount,
                    sendtype, i,...),
```

а каждый процесс выполнит приём

```
MPI_Recv(recvbuf, recvcount, recvtype, i,...)
```

Буфер отправки игнорируется всеми некорневыми процессами.

Рассылка данных

MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

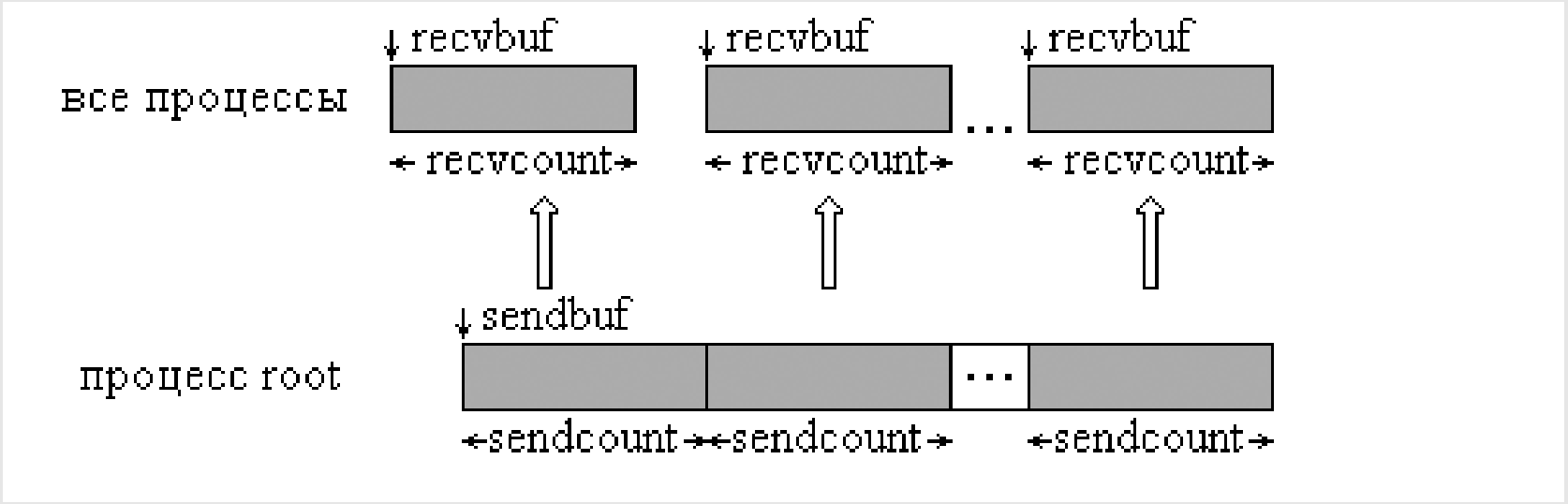
IN sendbuf начальный адрес буфера рассылки
IN sendcount количество элементов, посылаемых каждому процессу
IN sendtype тип данных элементов в буфере посылки
OUT recvbuf адрес буфера процесса-получателя
IN recvcount количество элементов в буфере корневого
IN recvtype тип данных элементов приемного буфера
IN root номер процесса-получателя
IN comm коммуникатор

Рассылка данных

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

void MPI::Intracomm::Scatter(const void* sendbuf,
    int sendcount, const Datatype& sendtype,
    void* recvbuf, int recvcount, const Datatype& recvtype,
    int root) const
```

Рассылка данных

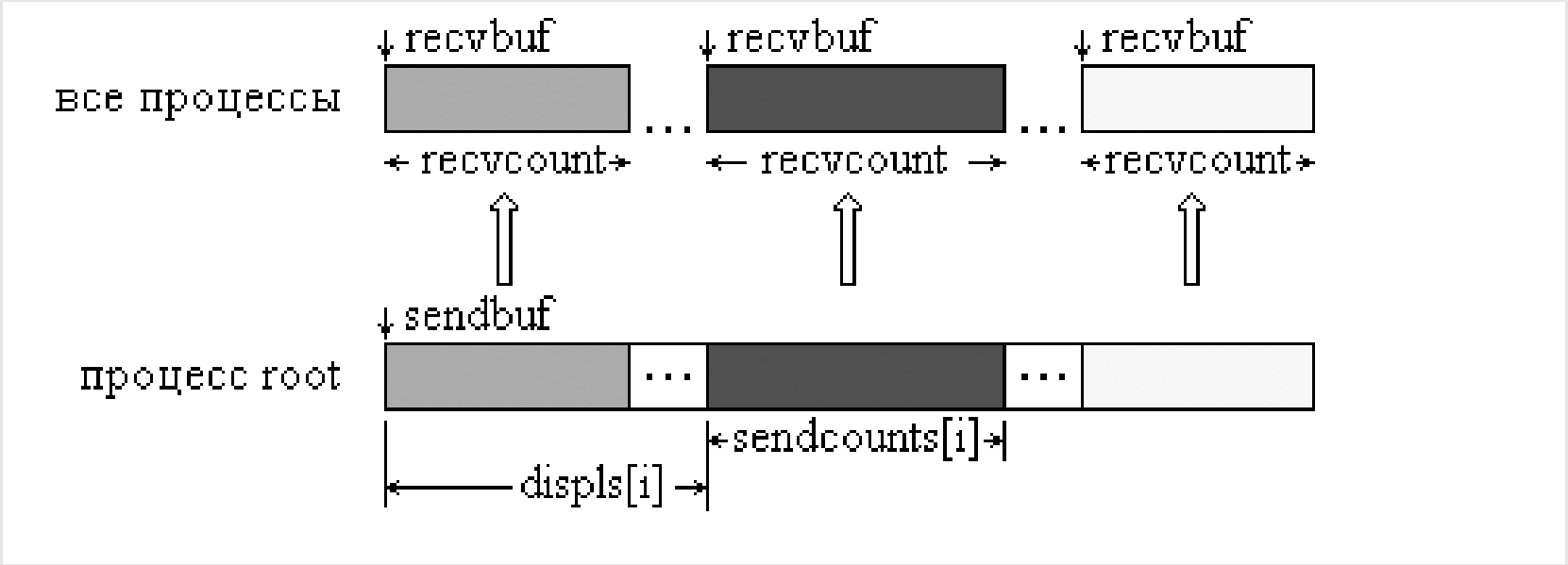


MPI_SCATTERV

MPI_SCATTERV(
sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

…..
IN sendcounts целочисленный массив (размера группы), определяющий число элементов, для отправки каждому процессу
IN displs целочисленный массив (размера группы). Элемент i указывает смещение (относительно sendbuf, из которого берутся данные для процесса)
…..

MPI_SCATTERV



Пример рассылки данных

MPI_SCATTER рассылает 100 чисел типа int из корневого процесса каждому процессу в группе

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100,
    MPI_INT, root, comm);
```

Сборка для всех процессов

Функцию MPI_ALLGATHER можно представить как MPI_GATHER, где результат принимают все процессы, а не только главный.

Результат выполнения вызова MPI_ALLGATHER(...) такой же, как если бы все процессы выполнили n вызовов

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

для `root = 0, ..., n-1.`

Сборка для всех процессов

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

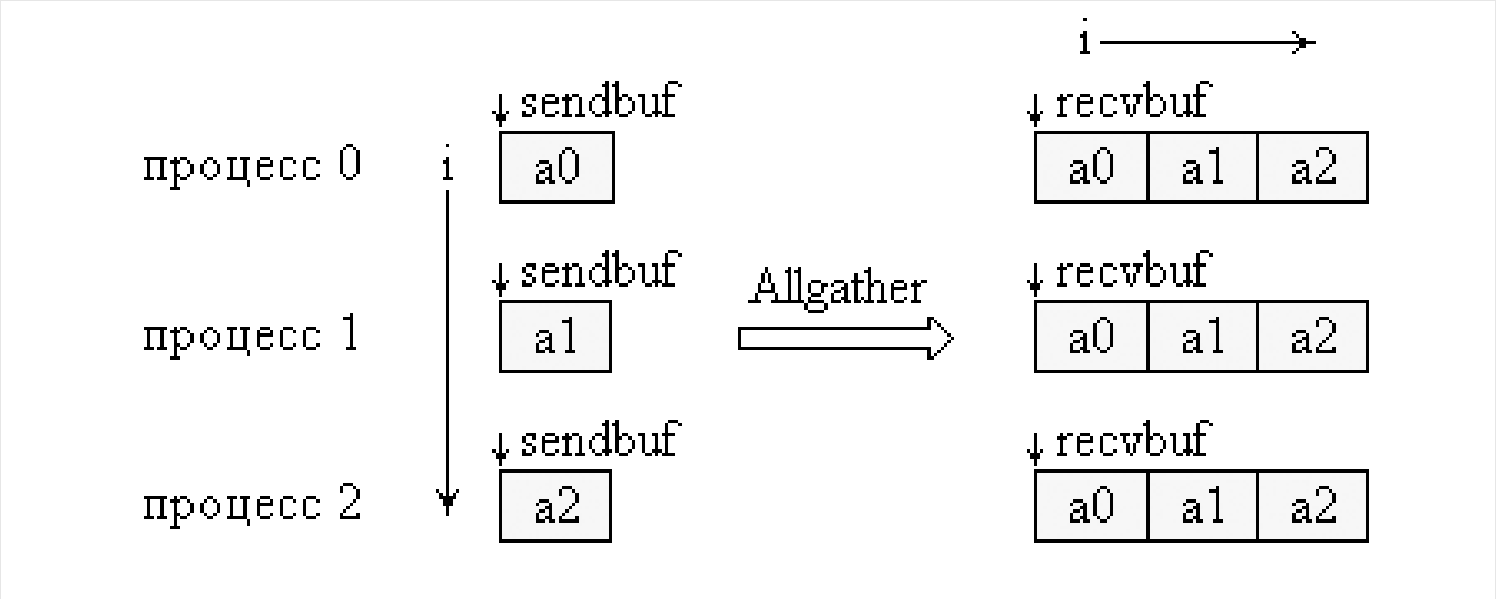
IN sendbuf начальный адрес посылающего буфера
IN sendcount количество элементов в буфере
IN sendtype тип данных элементов в посылающем буфере
OUT recvbuf адрес принимающего буфера
IN recvcount количество элементов, полученных от любого процесса (целое)
IN recvtype тип данных элементов принимающего буфера (дескриптор)
IN comm коммуникатор (дескриптор)

Сборка для всех процессов

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

void MPI::Intracomm::Allgather(const void* sendbuf,
    int sendcount, const Datatype& sendtype,
    void* recvbuf, int recvcount, const Datatype& recvtype)
    const
```

Сборка для всех процессов



MPI_ALLGATHERV

MPI_ALLGATHERV(sendbuf,sendcount,sendtype,recvbuf,recvcounts, displs,recvtype,comm)

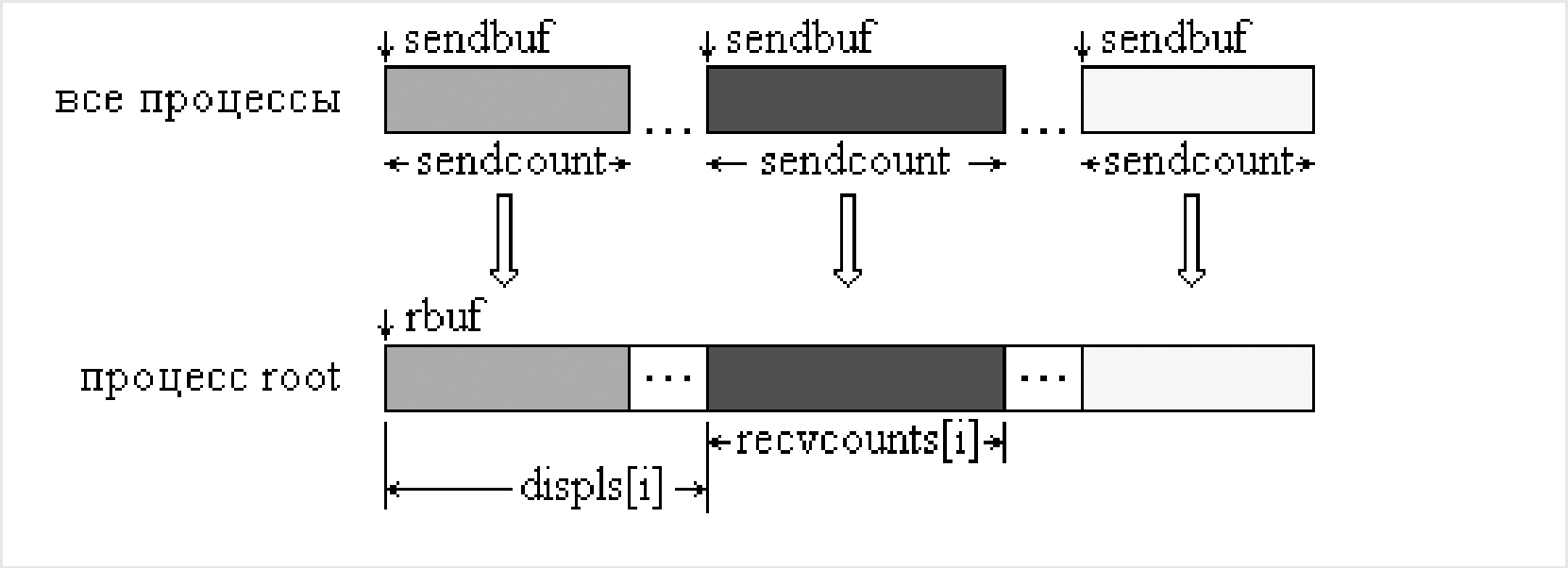
…..

IN recvcounts целочисленный массив (размера группы) содержащий количество элементов, полученых от каждого процесса

IN displs целочисленный массив (размера группы). Элемент i представляет смещение области (относительно recvbuf), где на помещаются принимаемые данные от процесса i

…..

MPI_ALLGATHERV



Пример сборки для всех процессов

Сбор 100 чисел типа int от каждого процесса в группе для каждого процесса.

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100,
    MPI_INT, comm);
```

all-to-all Scatter/Gather

MPI_ALLTOALL - это расширение функции MPI_ALLGATHER для случая, когда каждый процесс посылает различные данные каждому получателю.

Результат выполнения функции MPI_ALLTOALL такой же, как если бы каждый процесс выполнил посылку данных каждому процессу (включая себя) вызовом

```
MPI_Send(sendbuf + i * sendcount * extent(sendtype),
         sendcount, sendtype, i, ...),
```

и принял данные от всех остальных процессов путем вызова

```
MPI_Recv(recvbuf + i* recvcount* extent(recvtype),
         recvcount, i, ...).
```

all-to-all Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

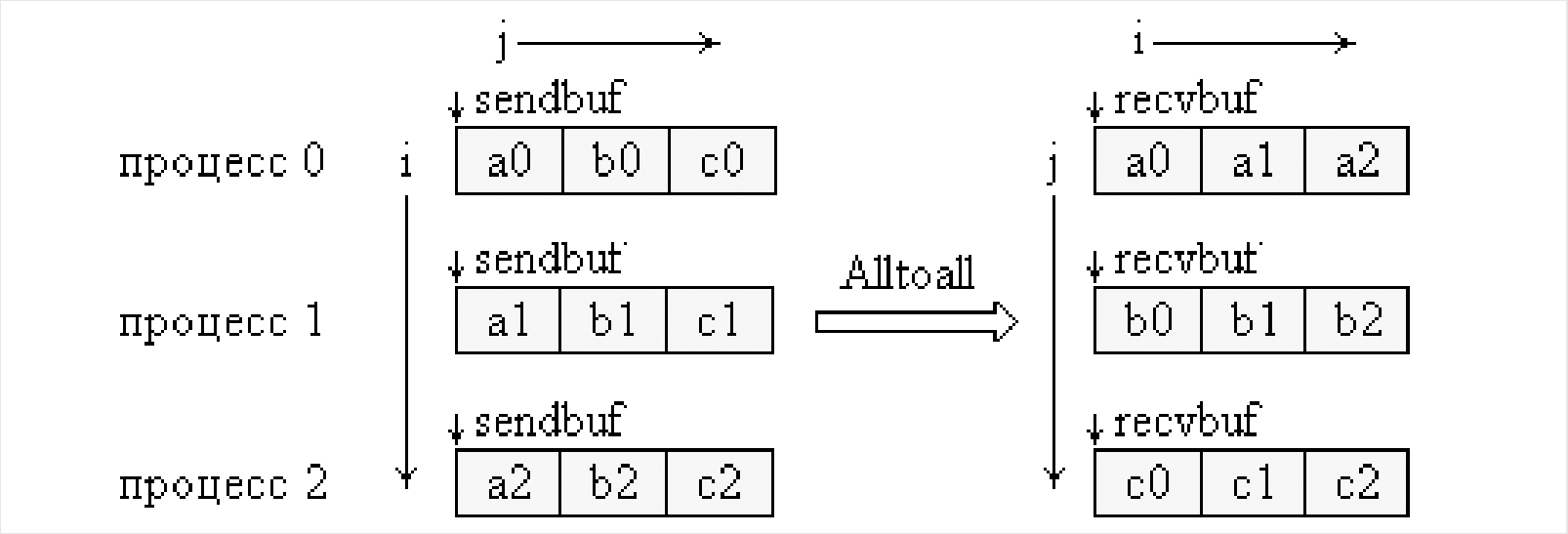
IN sendbuf начальный адрес посылающего буфера
IN sendcount количество элементов, посылаемых в каждый процесс
IN sendtype тип данных элементов посылающего буфера
OUT recvbuf адрес принимающего буфера
IN recvcount количество элементов, принятых от какого-либо процесса
IN recvtype тип данных элементов принимающего буфера
IN comm коммуникатор

all-to-all Scatter/Gather

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

void MPI::Intracomm::Alltoall(const void* sendbuf,
    int sendcount, const Datatype& sendtype,
    void* recvbuf, int recvcount, const Datatype& recvtype)
    const
```

all-to-all Scatter/Gather



MPI_ALLTOALLV(sendbuf,sendcounts,sdispls,sendtype, recvbuf,recvcounts,rdispls,recvtype,comm)

…..

IN sendcounts целочисленный массив (размера группы), определяющий количество посылаемых каждому процессу элементов

IN sdispls целочисленный массив(размера группы). Элемент j содержит смещение области (относительно sendbuf), из которой берутся данные для процесса j

…..

IN recvcounts целочисленный массив (размера группы), содержащий число элементов, которые могут быть приняты от каждого процессса

IN rdispls целочисленный массив (размера группы). Элемент i определяет смещение области (относительно recvbuf), в которой размещаются данные, получаемые из процесса i

Вычислительные операции над распределенными данными

- 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).

Операции редукции

Функции предназначены для выполнения операций глобальной редукции (суммирование, нахождение максимума, логическое И, и т.д.) для всех элементов группы.

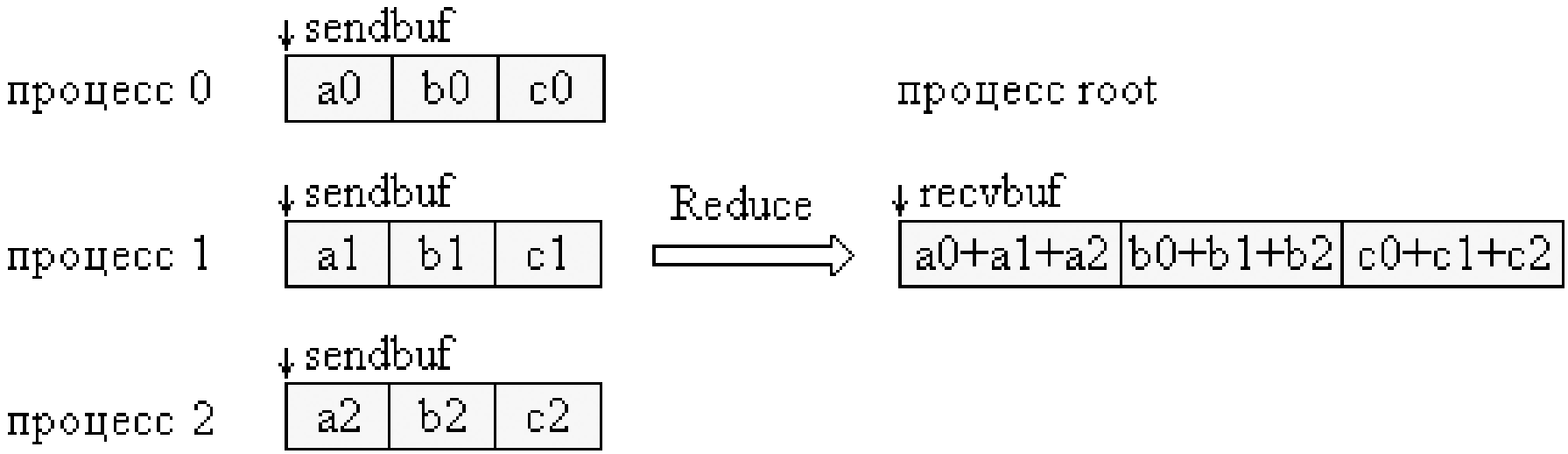
Операция редукции может быть выбрана из предопределенного списка операций, или определяться пользователем.

Функции глобальной редукции имеют несколько разновидностей: операции, возвращающие результат в один узел; функции (all-reduce), возвращающие результат во все узлы; операция просмотра.

MPI_REDUCE

Функция MPI_REDUCE объединяет элементы входного буфера каждого процесса в группе, используя операцию, и возвращает объединенное значение в выходной буфер процесса с номером root.

MPI_REDUCE


sendbuf
процесс 0
a0
b0
c0
процесс root
sendbuf
процесс 1
a1
b1
c1
Reduce
recvbuf
a0+a1+a2
b0+b1+b2
c0+c1+c2
sendbuf
процесс 2
a2
b2
c2

MPI_REDUCE

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN sendbuf адрес посылающего буфера
OUT recvbuf адрес принимающего буфера
IN count количество элементов в посылающем буфере
IN datatype тип данных элементов посылающего буфера
IN op операция редукции (дескриптор)
IN root номер главного процесса (целое)
IN comm коммуникатор (дескриптор)

MPI_REDUCE

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

void MPI::Intracomm::Reduce(const void* sendbuf,
    void* recvbuf, int count, const Datatype& datatype,
    const Op& op, int root) const
```

Предопределенные операции редукции

- MPI_MAX, MPI_MIN - определение максимального и минимального значения;
- MPI_MINLOC, MPI_MAXLOC- определение максимального и минимального значения и их местоположения;
- MPI_SUM, MPI_PROD - вычисление глобальной суммы и глобального произведения;
- MPI_LAND, MPI_LOR, MPI_LXOR - логические "И", "ИЛИ", исключающее "ИЛИ";
- MPI_BAND, MPI_BOR, MPI_BXOR - побитовые "И", "ИЛИ", исключающее "ИЛИ"

Пример использования MPI_REDUCE

```
// Инициализация

for( i=0; i<4; i++)
   outbuf[i] = rand()%100;

printf("Rank %.2d: %.2d, %.2d, %.2d, %.2d\n",
   rank,
   outbuf[0], outbuf[1],outbuf[2], outbuf[3]);

MPI_Reduce( outbuf, inbuf, PROC, MPI_INT,
   MPI_MAX, 0, MPI_COMM_WORLD);

if( rank == 0)
   printf("Maximum: %.2d, %.2d, %.2d, %.2d\n",
      inbuf[0], inbuf[1], inbuf[2], inbuf[3]);

MPI_Finalize();
```

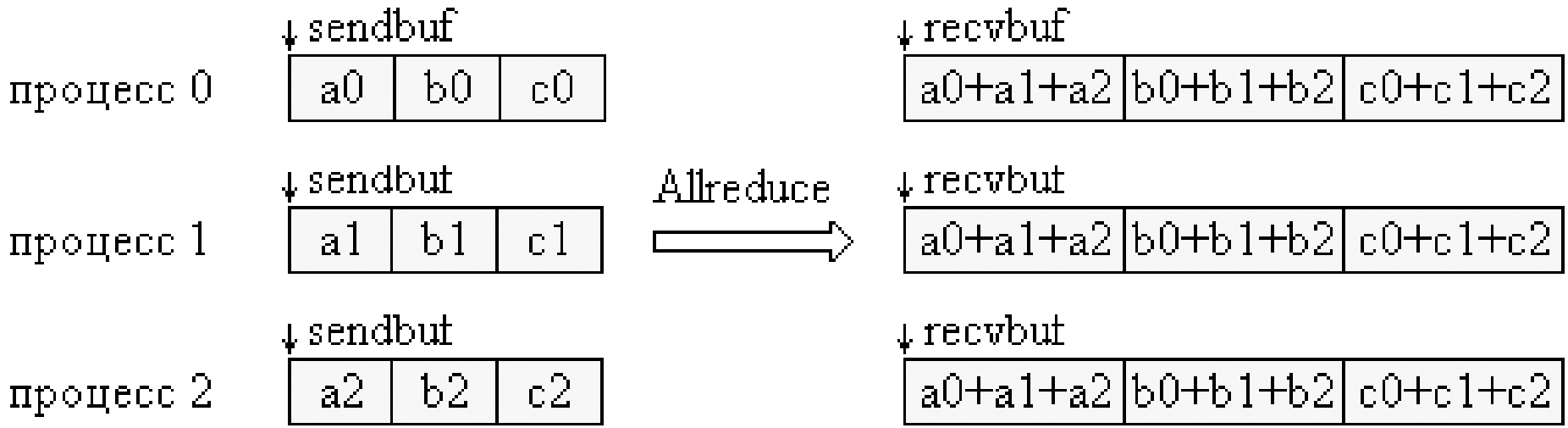
Результат работы

```
Rank 00: 87, 62, 65, 66
Rank 03: 38, 33, 10, 41
Rank 01: 25, 41, 34, 23
Maximum: 87, 62, 65, 80
Rank 02: 68, 49, 08, 80
```

MPI_ALLREDUCE

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 MPI_REDUCE.

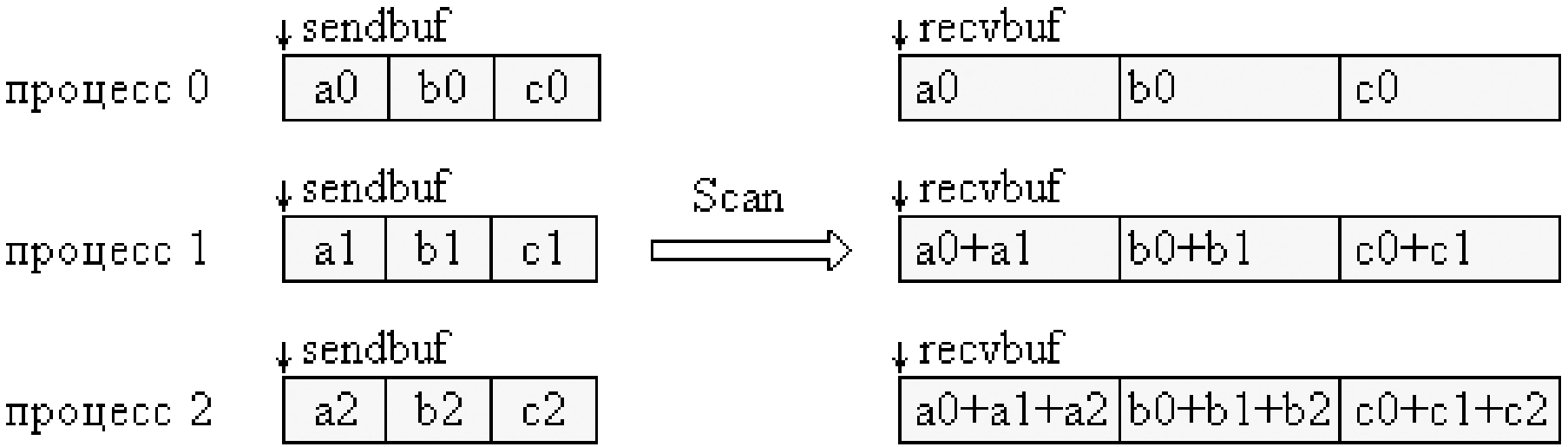
MPI_ALLREDUCE



MPI_Scan

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.

MPI_SCAN


sendbuf
процесс 0
a0
b0
c0
recvbuf
a0
b0
c0
sendbuf
процесс 1
a1
b1
c1
Scan
recvbuf
a0+a1
b0+b1
c0+c1
sendbuf
процесс 2
a2
b2
c2
recvbuf
a0+a1+a2
b0+b1+b2
c0+c1+c2

Группы, контексты и коммуникаторы

Группы

Группа — упорядоченный набор идентификаторов процессов. Каждый процесс в группе связан с целочисленным номером. Нумерация является непрерывной и начинается с нуля.

Существует две предопределенных группы:
MPI_GROUP_EMPTY — группа, не содержащая ни одного процесса;
MPI_GROUP_NULL — значение возвращаемое, когда группа не может быть создана.

Контексты

Контекст — свойство коммуникаторов, которое позволяет разделять пространство обмена. Сообщение, посланное в одном контексте, не может быть получено в другом контексте. Более того, там, где это разрешено, коллективные операции независимы от ждущих операций парного обмена.

Коммуникаторы

Коммуникаторы объединяют концепции группы и контекста.

Операции обмена в MPI используют коммуникаторы для определения области, в которой должны выполняться парная или коллективная операция. Для коллективной связи коммуникатор определяет набор процессов, которые участвуют в коллективной операции.

Таким образом, коммуникатор ограничивает «пространственную» область коммуникации, и обеспечивает машинно-независимую адресацию процессов их номерами.

Коммуникаторы

В MPI существует два типа коммуникаторов:

Intracommunicator — описывает область связи некоторой группы процессов;

Intercommunicator — служит для связи между процессами двух различных групп.

Предопределенные коммуникаторы

MPI_COMM_WORLD — описывает область связи, содержащую все процессы;

MPI_COMM_SELF — описывает область связи, состоящую из одного процесса

Конструкторы групп

Конструкторы групп применяются к подмножеству и расширенному множеству существующих групп. Эти конструкторы создают новые группы на основе существующих групп.

Данные операции являются локальными и различные группы могут быть определены на различных процессах; процесс может также определять группу, которая не включает саму себя.

MPI не имеет механизма для формирования группы с нуля, группа может формироваться только на основе другой, предварительно определенной группы.

Конструкторы групп

Функция MPI_COMM_GROUP возвращает в group дескриптор группы из comm.

MPI_COMM_GROUP(comm, group)

IN comm коммуникатор
OUT group группа, соответствующая comm

int MPI_Comm_group(MPI_Comm comm,
MPI_Group *group)

MPI::Group MPI::Comm::Get_group() const

Конструкторы групп

Обьединение (union) - содержит все элементы первой группы (group1) и следующие за ними элементы второй группы (group2), не входящие в первую группу.

MPI_GROUP_UNION(group1, group2, newgroup)

IN group1 первая группа
IN group2 вторая группа
OUT newgroup объединенная группа

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

Конструкторы групп

Пересечение (intersect) - содержит все элементы первой группы, которые также находятся во второй группе, упорядоченные как в первой группе.

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN group1 первая группа
IN group2 вторая группа
OUT newgroup группа, образованная пересечением

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

Конструкторы групп

Разность (difference) - содержит все элементы первой группы, которые не находятся во второй группе, упорядоченные как в первой группе.

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN group1 первая группа
IN group2 вторая группа
OUT newgroup исключенная группа

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

Конструкторы групп

Функция MPI_GROUP_INCL создает группу newgroup, которая состоит из n процессов из group с номерами ranks[0],..., ranks[n-1]; процесс с номером i в newgroup есть процесс с номером ranks[i] в group. Каждый из n элементов ranks должен быть правильным номером в group, и все элементы должны быть различными, иначе программа будет неверна. Если n = 0, то newgroup имеет значение MPI_GROUP_EMPTY. Эта функция может использоваться, например, для переупорядочения элементов группы

Конструкторы групп

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN group группа
IN n количество элементов в массиве номеров (и размер newgroup, целое)
IN ranks номера процессов в group, перешедших в новую группу (массив целых)
OUT newgroup новая группа, полученная из прежней, упорядоченная согласно ranks

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

Конструкторы групп

Функция MPI_GROUP_EXCL создает группу процессов newgroup, которая получена путем удаления из group процессов с номерами ranks[0],..., ranks[n-1]. Упорядочивание процессов в newgroup идентично упорядочиванию в group. Каждый из n элементов ranks должен быть правильным номером в group, и все элементы должны быть различными

Конструкторы групп

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN group группа
IN n количество элементов в массиве номеров
IN ranks массив целочисленных номеров в group, не входящих в newgroup
OUT newgroup новая группа, полученная из прежней, сохраняющая порядок, определенный group

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

Деструктор групп

Эта операция маркирует объект группы для удаления. Дескриптор group установливается вызовом в состояние MPI_GROUP_NULL. Любая выполняющаяся операция, использующая эту группу, завершится нормально.

MPI_GROUP_FREE(group)

INOUT group идентификатор группы

int MPI_Group_free(MPI_Group *group)

Определение размера группы

MPI_GROUP_SIZE (group, size)

IN group группа
OUT size количество процессов в группе

int MPI_Group_size(MPI_Group group, int *size)

Определение номера процесса в группе

MPI_GROUP_RANK (group, rank)

IN group группа
OUT rank номер процесса в группе или
MPI_UNDEFINED, если процесс не является
членом группы

int MPI_Group_rank(MPI_Group group, int *rank)

Конструкторы коммуникаторов

Функция создания нового коммуникатора. Функция возвращает MPI_COMM_NULL для процессов, не входящих в group.

MPI_COMM_CREATE(comm, group, newcomm)

IN comm коммуникатор
IN group группа, являющаяся подмножеством группы comm
OUT newcomm новый коммуникатор

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

Конструкторы коммуникаторов

Функция MPI_COMM_SPLIT делит группу, связанную с comm на непересекающиеся подгруппы, по одной для каждого значения цвета color.

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN comm коммуникатор
IN color управление соданием подмножества
IN key управление назначением номеров
OUT newcomm новый коммуникатор

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

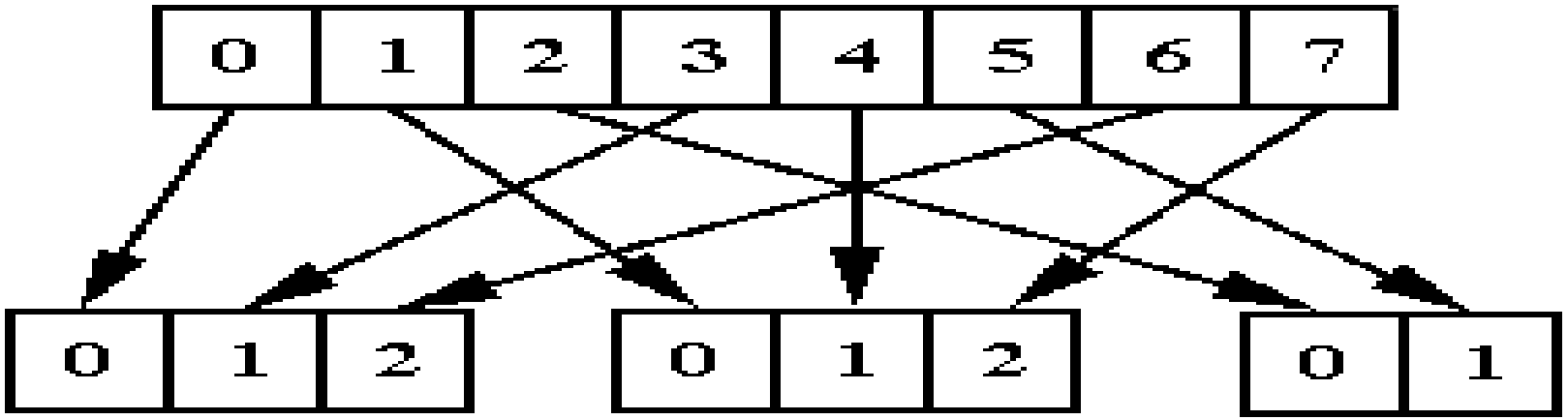
Пример расщепления на подгруппы

```
MPI_Comm comm;

 . . . . .

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

color= rank%3;
MPI_Comm_split( MPI_COMM_WORLD, color, rank,
 &comm);
```



Деструктор коммуникаторов

MPI_COMM_FREE(comm)

INOUT comm удаляемый коммуникатор

int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR) INTEGER COMM, IERROR

void MPI::Comm:Free()

Определение размера коммуникатора

MPI_COMM_SIZE(comm, size)

IN comm коммуникатор
OUT size количество процессов в группе comm

int MPI_Comm_size(MPI_Comm comm, int *size)

Получение номера процесса

Функция MPI_COMM_RANK возвращает номер процесса в частной группе коммуникатора

MPI_COMM_RANK(comm, rank)

IN comm коммуникатор
OUT rank номер вызывающего процесса в группе comm

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Интер-коммуникация

Интер-коммуникация является обменом типа «точка-точка» между процессами в различных группах. Группа, содержащая процесс, который инициализирует операцию интеркоммуникации, называется «локальной группой», а группа, содержащая целевой процесс, называется «удаленной группой».

Как и в интра-коммуникации, целевой процесс определяется парой (communicator, rank). В отличие от интра-коммуникации, номер указывается относительно второй, отдаленной группы.

Создание интер-коммуникатора

MPI_INTERCOMM_CREATE(local_comm,local_leader,peer_comm,remote_leader,tag,newintercomm)

IN local_comm локальный интракоммуникатор
IN local_leader номер лидера локальной группы в local_comm
IN peer_comm «уравнивающий коммуникатор», используется только на local_leader
IN remote_leader номер лидера удаленной группы в peer_comm, используется только на local_leader
IN tag тэг
OUT newintercomm новый интеркоммуникатор

```
int MPI_Intercomm_create(MPI_Comm local_comm,
    int local_leader, MPI_Comm peer_comm,
    int remote_leader, int tag, MPI_Comm *newintercomm)
```

MPI: топологии процессов

Топология является необязательным атрибутом, который дополняет систему интра-коммуникаторов и не применяется в интер-коммуникаторах.

Топология обеспечивает удобный способ обозначения процессов в группе (внутри коммуникатора) и оказывает помощь исполнительной системе при размещении процессов в аппаратной среде.

Виртуальные топологии

Взаимосвязь процессов может быть представлена графом. Узлы такого графа представляют процессы, а ребра соответствуют связям между процессами.

Ребра в графе не взвешены, поэтому процесс может изображаться только подключенным или не подключенным.

Во многих приложениях графовая структура является регулярной и довольно простой, поэтому использование всех деталей графового представления будет неудобным и малоэффективным в использовании.

Типы топологий

- Декартова топология
- Графовая топология

Как узнать тип топологии

```
MPI_TOPO_TEST(comm, status)

IN   comm    коммуникатор
OUT  status    тип топологии коммуникатора comm

int MPI_Topo_test(MPI_Comm comm, int *status)

MPI_GRAPH   топология графа
MPI_CART декартова топология
MPI_UNDEFINED   топология не определена
```

Декартова топология

В случае декартовой топологии (Cartesian topology) все процессы интерпретируются, как узлы некоторой n-мерной решетки размера

$k1 \times k2 \times \ldots \times kn$

(если n = 2, то процессы можно рассматривать как элементы прямоугольной матрицы размера $k1 \times k2$).

```
MPI_CART_CREATE(comm_old, ndims, dims, periods,
reorder, comm_cart)

IN   comm_old исходный коммуникатор
IN   ndims  размерность создаваемой
     декартовой решетки
IN   dims   целочисленный массив размера ndims,
     хранящий количество процессов по каждой
     координате
IN   periods  массив логических элементов размера
     ndims, определяющий, периодична (true) или нет
     (false) решетка в каждой размерности
IN   reorder  нумерация может быть сохранена (false
     или переупорядочена (true)
OUT  comm_cart  коммуникатор новой декартовой
     топологии
int MPI_Cart_create(MPI_Comm comm_old,
   int ndims, int *dims, int *periods,
   int reorder, MPI_Comm *comm_cart)
```

Определение размерности декартовой топологии

```
MPI_CARTDIM_GET(comm, ndims)

IN   comm   коммуникатор с декартовой топологией
OUT  ndims  число размерностей в декартовой
     топологии системы

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

Определение информации по декартовой топологии

```
MPI_CART_GET(comm, maxdims, dims, periods,
coords)

IN   comm    коммуникатор с декартовой топологией
IN   maxdims  длина векторов dims, periods и coords
OUT  dims     число процессов по каждой декартовой
     размерности
OUT  periods  периодичность (true / false) для
     каждой декартовой размерности
OUT  coords   координаты вызываемых процессов в
     декартовой системе координат

int MPI_Cart_get(MPI_Comm comm, int maxdims,
   int *dims, int *periods, int *coords)
```

Перевод логических координат процессов в номера, которые используются в процедурах парного обмена

```
MPI_CART_RANK(comm, coords, rank)

IN   comm    коммуникатор с декартовой топологией
IN   coords    целочисленный массив (размера ndims),
     описывающий декартовы координаты процесса
OUT  rank    номер указанного процесса

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

Перевод номера процесса в координату

```
MPI_CART_COORDS(comm, rank, maxdims, coords)

IN   comm    коммуникатор с декартовой топологией
IN   rank    номер процесса внутри группы comm
IN   maxdims  длина вектора coord (целое)
OUT  coords    целочисленный массив (размера ndims),
     содержащий декартовы координаты указанного
     процесса

int MPI_Cart_coords(MPI_Comm comm, int rank,
   int maxdims, int *coords)
```

Если используется декартова топология, то операцию MPI_SENDRECV можно выполнить путем сдвига данных вдоль направления координаты.

В качестве входного параметра MPI_SENDRECV использует номер процесса-отправителя для приема и номер процесса-получателя - для передачи.

Если функция MPI_CART_SHIFT выполняется для декартовой группы процессов, то она передает вызывающему процессу эти номера, которые затем могут быть использованы для MPI_SENDRECV.

Сдвиг в декартовых координатах

```
MPI_CART_SHIFT(comm, direction, disp,
rank_source,
   rank_dest)

IN   comm   коммуникатор с декартовой топологией
IN   direction   координата сдвига
IN   disp   направление смещения
     (> 0: смещение вверх, < 0: смещение вниз)
OUT  rank_source   номер процесса-отправителя
OUT  rank_dest   номер процесса-получателя

int MPI_Cart_shift(MPI_Comm comm, int direction,
   int disp, int *rank_source, int *rank_dest)
```

Декомпозиция декартовых структур

```
MPI_CART_SUB(comm, remain_dims, newcomm)

IN   comm   коммуникатор с декартовой топологией
IN   remain_dims   i-ый элемент в remain_dims
     показывает, содержится ли i-ая размерность в
     подрешетке (true) или нет (false)
OUT  newcomm   коммуникатор, содержащий подрешетку,
     которая включает вызываемый процесс

int MPI_Cart_sub(MPI_Comm comm,
     int *remain_dims, MPI_Comm *newcomm)
```

Пример: создание матрицы процессов

```
int rank, size;
int value;
MPI_Status status;
MPI_Comm comm;

int dims[] = { 3, 3};
int periods[] = { 0, 0 };
int reorder = 1;
int coords[2];
int src, dst;
```

```
MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
        reorder, &comm);

    MPI_Comm_rank( comm, &rank);
    MPI_Cart_coords( comm, rank, 2, coords);

    value = (rank+1)*10;

    printf("Process before %d[%d,%d] = %d\n", rank,
        coords[0], coords[1], value);
```

```
MPI_Cart_shift( comm, 1, 1, &src, &dst);
MPI_Sendrecv_replace( &value, 1, MPI_INT, dst,
    99, src, 99, comm, &status);

MPI_Barrier(comm);
printf("Process after %d[%d,%d] = %d\n", rank,
    coords[0], coords[1], value);

MPI_Finalize();
```

Результат работы матричного обмена

Before 0[0,0] = 10
Before 1[0,1] = 20
Before 2[0,2] = 30
Before 3[1,0] = 40
Before 4[1,1] = 50
Before 5[1,2] = 60
Before 6[2,0] = 70
Before 7[2,1] = 80
Before 8[2,2] = 90

After 0[0,0] = 10
After 1[0,1] = 10
After 2[0,2] = 20
After 3[1,0] = 40
After 4[1,1] = 40
After 5[1,2] = 50
After 6[2,0] = 70
After 7[2,1] = 70
After 8[2,2] = 80

Пример: создание «трубы» из процессов

```
int rank, size;
int value;
MPI_Status status;
MPI_Comm comm;

int dims[] = { 3, 3};
int periods[] = { 0, 1 };
int reorder = 1;
int coords[2];
int src, dst;
```

Результат работы «трубчатого» обмена

Before 0[0,0] = 10
Before 1[0,1] = 20
Before 2[0,2] = 30
Before 3[1,0] = 40
Before 4[1,1] = 50
Before 5[1,2] = 60
Before 6[2,0] = 70
Before 7[2,1] = 80
Before 8[2,2] = 90

After 0[0,0] = 30
After 1[0,1] = 10
After 2[0,2] = 20
After 3[1,0] = 60
After 4[1,1] = 40
After 5[1,2] = 50
After 6[2,0] = 90
After 7[2,1] = 70
After 8[2,2] = 80

Топология графов

В случае топологии графа (graph topology) процессы интерпретируются как вершины некоторого графа

Связи между процессами определяются посредством задания набора ребер (дуг) для этого графа.

Создание графовой топологии

```
MPI_GRAPH_CREATE(comm_old, nnodes, index, edges,
reorder, comm_graph)

IN   comm_old входной коммуникатор
IN   nnodes   количество узлов графа
IN   index  массив целочисленных значений,
     описывающий степени вершин(i-ый элемент массива
index хранит общее число соседей первых i вершин графа)
IN   edges  массив целочисленных значений,
     описывающий ребра графа – списки соседей вершин
IN   reorder  номера могут быть переупорядочены
     (true) или нет (false)
OUT  comm_graph  построенный коммуникатор с графовой
     топологией (дескриптор)

int MPI_Graph_create(MPI_Comm comm_old,
   int nnodes, int *index, int *edges,
   int reorder, MPI_Comm *comm_graph)
```

Получение размера графа

```
MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN   comm   коммуникатор группы с графовой
     топологией
OUT  nnodes   число вершин графа
     (целое, равно числу процессов в группе)
OUT  nedges   число ребер графа

int MPI_Graphdims_get(MPI_Comm comm,
   int *nnodes, int *nedges)
```

Получение топологии графа

```
MPI_GRAPH_GET(comm, maxindex, maxedges, index,
edges)

IN   comm    коммуникатор с графовой топологией
IN   maxindex длина вектора index
IN   maxedges длина вектора edges
OUT  index   целочисленный массив, содержащий
     структуру графа
OUT  edges   целочисленный массив, содержащий
     структуру графа

int MPI_Graph_get(MPI_Comm comm,
   int maxindex, int maxedges,
   int *index, int *edges)
```

Получение количества соседей процесса

```
MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN   comm   коммуникатор с графовой топологией
IN   rank   номер процесса в группе comm (целое)
OUT  nneighbors  номера процессов, являющихся
     соседними указанному процессу

int MPI_Graph_neighbors_count(MPI_Comm comm,
int rank, int *nneighbors)
```

Получение списка соседей

```
MPI_GRAPH_NEIGHBORS(comm, rank,
  maxneighbors, neighbors)

IN   comm       коммуникатор с графовой топологией
IN   rank       номер процесса в группе comm
IN   maxneighbors  размер массива neighbors
OUT  neighbors  номера процессов, соседних данному

int MPI_Graph_neighbors(MPI_Comm comm,
  int rank, int maxneighbors, int *neighbors)
```

Пример: создание графа

Вершина	Соседи
0	1,3
1	0
2	3
3	0,2

```
int rank, size;
MPI_Status status;
MPI_Comm comm;
int reorder = 1;
int nnodes = 4;
int index[] = {2,3,4,6};
int edges[] = {1,3,0,3,0,2};
int nneigh;

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

MPI_Graph_create(MPI_COMM_WORLD, nnodes, index,
    edges, reorder, &comm);

MPI_Comm_rank( comm, &rank);
MPI_Graph_neighbors_count( comm, rank, &nneigh);

printf("Process %d have %d neighbors\n",
    rank, nneigh);

MPI_Finalize();
```

Результат работы

Process 0 have 2 neighbors
Process 1 have 1 neighbors
Process 2 have 1 neighbors
Process 3 have 2 neighbors

Расширение MPI-2

Объект Info

Многие из подпрограмм MPI-2 берут аргумент info. info - скрытый объект с указателем типа MPI_Info в , MPI::Info в и INTEGER в . Он состоит из (key, value) пар (и key и value - строки).

Ключ может иметь только одно значение. MPI резервирует несколько ключей и требует, чтобы, если реализация использовала зарезервированный ключ, она должна обеспечить указанные функциональные возможности.

Функции для работы с ключами

```
int MPI_info_create (MPI_Info *info);
int MPI_info_set(MPI_Info info,
     char *key, char *value)
int MPI_Info_delete(MPI_Info info, char *key)
int MPI_info_get(MPI_Info info,
     char *key, int valuelen, char *value, int *flag)
int MPI_Info_get_valuelen (MPI_Info info,
     char *key, int *valuelen, int *flag)
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
int MPI_Info_get_nthkey (MPI_Info info,
     int n, char *key)
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
int MPI_info_free(MPI_Info *info)
```

Пример создания объекта Info

```
int errs = 0;
MPI_Info info;
int nkeys, i, vallen, flag, rank;
char key[MPI_MAX_INFO_KEY];
char value[MPI_MAX_INFO_VAL];

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

MPI_Info_create( &info );
```

```
if( rank == 0){
     MPI_Info_set( info, "host", "myhost" );
     MPI_Info_set( info, "file", "runfile.txt" );
     MPI_Info_set( info, "test", "2.15.234.5" );
   } else {
     MPI_Info_set( info, "host2", "myhost" );
     MPI_Info_set( info, "file3", "runfile.txt" );
   }

    MPI_Info_get_nkeys( info, &nkeys );
    vallen = MPI_MAX_INFO_VAL;

    for (i=0; i<nkeys; i++) {
        MPI_Info_get_nthkey( info, i, key );
        MPI_Info_get( info,key,vallen,value,&flag );
        printf( "Process %d: %s = %s\n",
             rank, key, value );
    }
    MPI_Info_free( &info );
    MPI_Finalize();
```

Результат выполнения

Process 0: host = myhost
Process 0: file = runfile.txt
Process 0: test = 2.15.234.5
Process 1: host2 = myhost
Process 1: file3 = runfile.txt

Создание и управление процессами

Приложения MPI могут запустить новые процессы через интерфейс внешнего менеджера процессов.

MPI_COMM_SPAWN запускает процессы MPI и устанавливает с ними соединение, возвращая интеркоммуникатор.

MPI_COMM_SPAWN_MULTIPLE запускает несколько различных файлов (или один двоичный файл с разными аргументами), помещая их в единый MPI_COMM_WORLD и возвращая интеркоммуникатор. MPI использует существующую абстракцию группы для представления процессов.

```
MPI_COMM_SPAWN(command, argv, maxprocs, info,
root, comm, intercomm, array_of_errcodes)

IN   command  Имя порождаемой программы
IN   argv     Аргументы команды
IN   maxprocs Максимальное число процессов для
     запуска
IN   info     Набор пар ключ-значение, сообщающий
     системе выполнения, где и как запускать процессы
IN   root     Ранг процесса, для которого
     анализируются предыдущие аргументы
IN   comm     Интеркоммуникатор, содержащий группу
     порожденных процессов
OUT  intercomm  Интеркоммуникатор между первичной и
     вновь порожденной группой
OUT  array_of_errcodes  Один код на процесс

int MPI_Comm_spawn(char *command, char **argv,
   int maxprocs, MPI_Info info, int root, MPI_Comm
   comm, MPI_Comm *intercomm, int
*array_of_errcodes)
```

Создание множества различных процессов

```
MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands,
   array_of_argv, array_of_maxprocs, array_of_info,
   root, comm, intercomm, array_of_errcodes)
```

IN **count** Количество команд
IN **array_of_commands** Выполняемые программы
IN **array_of_argv** Аргументы для **commands**
IN **array_of_maxprocs** Максимальное количество процессов, запускаемых для каждой команды
IN **array_of_info** Объекты **info**, сообщающие системе выполнения, где и как запускать процессы
IN **root** Ранг процесса, в котором проверяются предыдущие аргументы
IN **comm** Внутренний коммуникатор, содержащий группу порожденных процессов
OUT **intercomm** Интеркоммуникатор между оригинальной и вновь порожденной группами
OUT **array_of_errcodes** Массив кодов ошибок

Получение интеркоммуникатора для связи с родителем

```
MPI_COMM_GET_PARENT(parent)

OUT  parent    Коммуникатор родителя

int MPI_Comm_get_parent (MPI_Comm *parent)
```

Примеры запуска новых процессов

```
char command= "myprog";
char *argv= "-gridfile", "my.grd", NULL;
MPI_Comm_spawn(command, argv, ...);

. . . . .

char *array_of_commands = "myprog","yourprog";
char **array_of_argv;
char *argv0= "-gridfile", "my.grd", (char *)0;
char *argv1= "testparam", (char *)0;
array_of_argv0= argv0;
array_of_argv1= argv1;
MPI_Comm_spawn_multiple(2, array_of_commands,
   array_of_argv, ...);
```

Установка соединения

Некоторыми ситуациями, в которых эти функции могут быть полезны, являются следующие:

• Две части приложения, запущенные независимо друг от друга, должны взаимодействовать.
• Инструмент визуализации желает присоединиться к работающему процессу.
• Сервер желает принять соединения от нескольких клиентов. И клиенты, и сервер могут быть параллельными программами.

~~Явки, пароли~~
Имена, адреса, порты

MPI может приспособиться к широкому кругу систем выполнения при сохранении возможности написания простого переносимого кода.

• Сервер расположен по хорошо известному адресу Internet на host:port.
• Сервер выводит адрес на терминал, а пользователь указывает этот адрес программе-клиенту.
• Сервер размещает информацию об адресе на сервере имен.
• Сервер, с которым клиент соединяется в действительности, является брокером, работающим как посредник между клиентом и действительным сервером.

Процедуры сервера

```
MPI_OPEN_PORT (info, port_name)

IN   info   Информация, специфичная для реализации
     об установке адреса
OUT  port_name   Новый установленный порт (строка)

int MPI_Open_port(MPI_Info info, char *port_name)



int MPI_Close_port (char *port_name)

int MPI_Comm_accept(char *port_name, MPI_Info info,
   int root, MPI_Comm comm, MPI_Comm *newcomm)
```

Процедура клиента

```
MPI_COMM_CONNECT(port_name,info,root,comm,newcomm)

IN   port_name   Сетевой адрес
IN   info    Информация, зависящая от реализации
IN   root    Ранг в comm для узла root
IN   comm    Интракоммуникатор, внутри которого
     выполняется коллективный вызов
OUT  newcomm  Интеркоммуникатор с сервером в
     качестве удаленной группы

int MPI_Comm_connect(char port_name, MPI_Info
info, int root, MPI_Comm comm, MPI_Comm *newcomm)
```

Опубликование имен

```
MPI_PUBLISH_NAME (service_name, info, port_name)

IN   service_name  Имя сервиса для ассоциации с
     портом
IN   info   Информация, зависящая от реализации
IN   port_name   Имя порта

int MPI_Publish_name(char *service_name,
   MPI_Info info, char *port_name)

int MPI_Unpublish_name(char *service_name,
   MPI_Info info, char *port_name)

int MPI_Lookup_name (char *service_name,
   MPI_Info info, char *port_name)
   OUT  port_name    Имя порта
```

Зарезервированные значения ключей

Зарезервировано только для MPI_OPEN_PORT

ip_port: Значение, содержащее номер порта IP, на котором устанавливается port

ip_address: Значение, содержащее IP-адрес, на котором устанавливается port.

Односторонние взаимодействия

Работа с памятью

В некоторых системах операции передачи сообщений и удаленный доступ к памяти (RMA) выполняются быстрее при доступе к специально распределенной памяти (например, память, которая разделена другими процессами в группе связи на SMP).

MPI обеспечивает механизм для распределения и освобождения такой специальной памяти. Использование такой памяти для передачи сообщений или RMA не обязательно, и эта память может использоваться без ограничений, как любая другая динамически распределенная память.

Удаленный доступ к памяти (RMA) расширяет механизмы взаимодействий MPI, позволяя одному процессу определить все коммуникационные параметры как для посылающей стороны, так и для получающей.

Каждый процесс может вычислить, к каким данным других процессов ему потребуется обратиться или какие данные модифицировать.

В то же время, процессы могут не знать, к каким данным в их собственной памяти потребуется обратиться удаленным (remote) процессам или что им потребуется модифицировать, мало того, они могут даже не знать, что это за процессы. Таким образом, параметры передачи оказываются доступными полностью только на одной стороне.

Управление памятью

```
MPI_ALLOC_MEM(size, info, baseptr)

IN   size    размер сегмента памяти в байтах
IN   info    аргумент информации
OUT  baseptr  указатель на начало распределенного
     сегмента памяти

int MPI_Alloc_mem(MPI_Aint size,
     MPI_Info info, void *baseptr)


MPI_FREE_MEM(base)

IN   base    начальный адрес сегмента памяти,
распределенного MPI_ALLOC_MEM

int MPI_Free_mem(void *base)
```

Создание окна

Операция инициализации позволяет каждому процессу из группы интракоммуникаторов определить, используя коллективную операцию, «окно» в своей памяти, которое становится доступным для удаленных процессов.

MPI_WIN_CREATE(base, size, disp_unit, info, comm,win)

IN **base** начальный адрес окна
IN **size** размер окна в байтах
IN **disp_unit** размер локальной единицы смещения в байтах
IN **info** аргумент info
IN **comm** коммуникатор
OUT **win** оконный объект, вызвращаемый вызовом

```
int MPI_Win_create(void *base, MPI_Aint size,
   int disp_unit, MPI_Info info, MPI_Comm comm,
   MPI_Win *win)
```

MPI_WIN_FREE(win)

INOUT **win** оконный объект (дескриптор)

```
int MPI_Win_free(MPI_Win *win)
```

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN **origin_addr** начальный адрес буфера инициатора
IN **origin_count** число записей в буфере инициатора
IN **origin_datatype** тип данных каждой записи в буфере инициатора
IN **target_rank** номер получателя
IN **taget_disp** смещение от начала окна до буфера получателя
IN **target_count** число записей в буфере получателя
IN **target_datatype** тип данных каждой записи в буфере получателя
IN **win** оконный объект, используемый для коммуникации

```
int MPI_Put(void *origin_addr, int origin_count,
   MPI_Datatype origin_datatype, int target_rank,
   MPI_Aint target_disp, int target_count,
   MPI_Datatype target_datatype, MPI_Win win)
```

```
MPI_GET(origin_addr, origin_count, origin_datatype,
   target_rank, target_disp, target_count,
   target_datatype, win)
```

OUT **origin_addr** начальный адрес буфера инициатора
IN **origin_count** число записей в буфере инициатора
IN **origin_datatype** тип данных каждой записи в буфере инициатора
IN **target_rank** ранк получателя
IN **target_disp** смещение от начала окна до буфера адресата
IN **target_count** число записей в буфере адресата
IN **target_datatype** тип данных каждой записи в буфере адресата
IN **win** оконный объект, используемый для коммуникации

```
int MPI_Get(void *origin_addr, int origin_count,
   MPI_Datatype origin_datatype, int target_rank,
   MPI_Aint target_disp, int target_count,
   MPI_Datatype target_datatype, MPI_Win win)
```

Синхронизация 1.

Коллективный синхронизационный MPI_WIN_FENCE вызов обеспечивает простую модель синхронизации, которая часто используется при параллельных вычислениях: а именно, слабосинхронную модель (loosely synchronous model), когда общие вычислительные фазы перемежаются с фазами общих комуникаций.

Синхронизация 2.

Четыре функции MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST и MPI_WIN_WAIT могут использоваться чтобы свести синхронизацию к минимуму: синхронизируются только пары взаимодействующих процессов, и это происходит только тогда, когда синхронизация необходима, чтобы корректно упорядочить RMA обращения к окну принимая во внимание локальные обращеня к этому же окну.

Синхронизация 3.

Общие и эксклюзивные блокировки обеспечиваются двумя вызовами
MPI_WIN_LOCK и MPI_WIN_UNLOCK.
Синхронизация с блокировками полезна для MPI приложений, которые эмулируют модель с общей памятью через MPI вызовы

Ввод/вывод

Открытие файла

MPI_FILE_OPEN открывает файл с именем filename для всех процессов из группы коммуникатора comm.

```
MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN   comm     коммуникатор
IN   filename имя открываемого файла (строка)
IN   amode    тип доступа к файлу
IN   info     информационный объект
OUT  fh     новый дескриптор файла

int MPI_File_open(MPI_Comm comm, char *filename,
   int amode, MPI_Info info, MPI_File *fh)
```

Параметры открытия

- MPI_MODE_RDONLY -- только чтение,
- MPI_MODE_RDWR -- чтение и запись,
- MPI_MODE_WRONLY -- только запись,
- MPI_MODE_CREATE -- создавать файл, если он не существует,
- MPI_MODE_EXCL -- ошибка, если создаваемый файл уже существует,
- MPI_MODE_DELETE_ON_CLOSE -- удалять файл при закрытии,
- MPI_MODE_UNIQUE_OPEN -- файл не будет параллельно открыт где-либо еще,
- MPI_MODE_SEQUENTIAL -- файл будет доступен лишь последовательно,
- MPI_MODE_APPEND -- установить начальную позицию всех файловых указателей на конец файла

Файловые операции

MPI_FILE_CLOSE(fh)
INOUT fh дескриптор файла
int **MPI_File_close**(MPI_File *fh)

MPI_FILE_DELETE(filename, info)
IN filename имя удаляемого файла
IN info информационный объект
int **MPI_File_delete**(char *filename, MPI_Info info)

MPI_FILE_SET_SIZE(fh, size)
INOUT fh дескриптор файла (дескриптор)
IN size размер, до которого необходимо расширить или урезать файл (целое)
int **MPI_File_set_size**(MPI_File fh, MPI_Offset size)

MPI_FILE_GET_SIZE(fh, size)
IN fh дескриптор файла (дескриптор)
OUT size размер файла в байтах (целое)
int **MPI_File_get_size**(MPI_File fh, MPI_Offset *size)

Файловые операции

возвращает дубликат группы коммуникатора, использованной для открытия файла

```
MPI_FILE_GET_GROUP(fh, group)
IN   fh   дескриптор файла
OUT  group  группа, открывшая файл
int MPI_File_get_group(MPI_File fh, MPI_Group
*group)
```

возвращает тип доступа к файлу

```
MPI_FILE_GET_AMODE(fh, amode)
IN   fh   дескриптор файла
OUT  amode  тип доступа, использованный при открытии
файла
int MPI_File_get_amode(MPI_File fh, int *amode)
```

Info для файлов

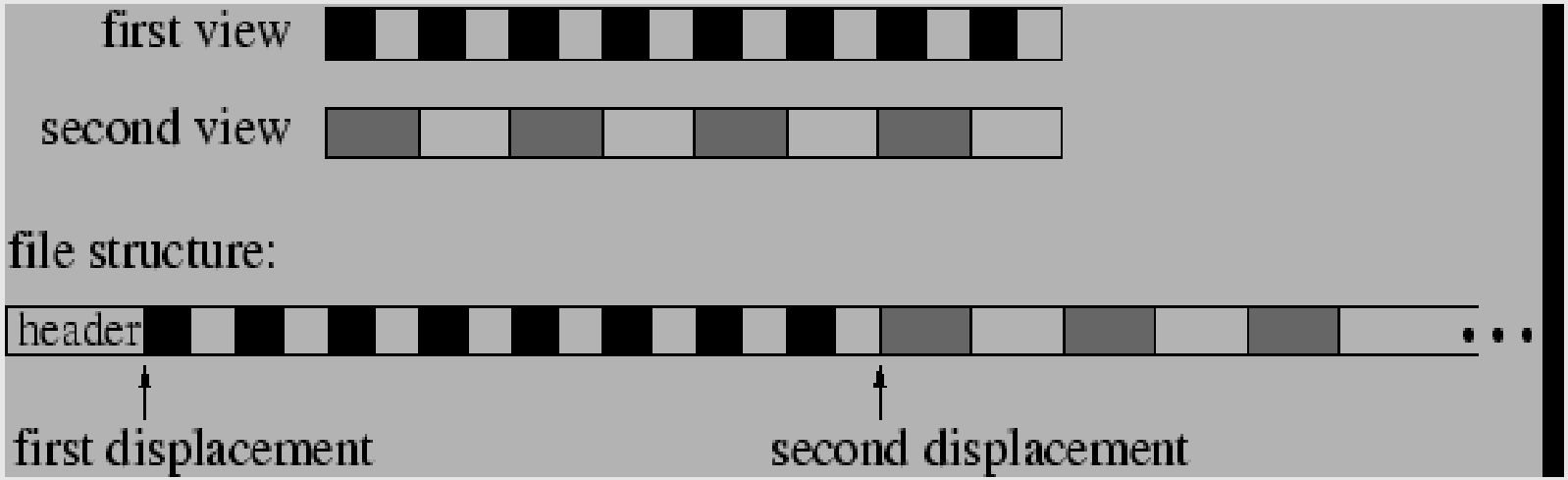
```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
int MPI_File_get_info(MPI_File fh, MPI_Info *info)
```

Некоторые зарезервированные ключи:
access_style, collective_buffering, cb_block_size, cb_buffer_size, cb_nodes, chunked, chunked_item, chunked_size, filename, file_perm, io_node_list, nb_proc, num_io_nodes, striping_factor, striping_unit

Файловые виды

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)



Позиционирование

MPI поддерживает три вида позиционирования для подпрограмм доступа к данным:

- точное смещение
- индивидуальные файловые указатели
- общие файловые указатели

Различные методы позиционирования могут быть смешаны в одной программе и не влияют друг на друга.

Синхронизация

MPI поддерживает блокирующие и неблокирующие вызовы.

При неблокирующих вызовах необходим отдельный вызов завершения запроса (MPI_WAIT, MPI_TEST или любой из их вариантов) для завершения запроса ввода-вывода, то есть для того, чтобы убедиться в том, что данные были записаны или прочитаны, и использовать буфер снова безопасно для пользователя.