Лекция 3. ИСПОЛЬЗОВАНИЕ СРЕДСТВ MPI ДЛЯ РЕАЛИЗАЦИИ ШИРОКОВЕЩАТЕЛЬНЫХ СООБЩЕНИЙ И ГРУППОВОЙ ПЕРЕСЫЛКИ ПРИ ОБМЕНЕ ДАННЫМИ.

Любую задачу в MPI можно решить, используя только связь типа «точка-точка», например, если необходимо передать значение вектора **x** всем процессам параллельной программы, то можно реализовать следующий код:

```
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
for(int i = 1; i < ProcNum+1; i++)
   MPI_Send(&x, n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

где **x** – массив вещественных чисел, **n**– размер массива. Однако такое решение неэффективно, поскольку повторение операций передачи приведет к суммированию затрат на подготовку передаваемых сообщений. Кроме того данная операция может быть выполнена за (ProcNum-1) итераций передачи данных. Более целесообразно использовать специальную MPI-функцию широковещательной рассылки, которая будет описана далее.

рПод коллективными операциями в MPI понимаются операции над данными, в которых принимают участие все процессы используемого коммуникатора. Из этого следует, что одним из ключевых аргументов в вызове коллективной функции является коммуникатор, который определяет группу (или группы) участвующих процессов, между которыми выполняется широковещательная передача данных. Несколько коллективных функций, таких как широковещательная рассылка (**broadcast**), сбор данных (**gather**) имеют единственный инициирующий или принимающий процесс, этот процесс называется корнем (**root**). Некоторые аргументы в коллективных функциях определены как аргументы, которые используются только корневым процессом, всеми другими участниками обмена эти аргументы игнорируются.

Условия согласования типов для коллективных операций являются более строгими, чем соответствующие условия между отправителем и получателем в передаче типе «точка-точка». Для коллективных операций количество отправленных данных должно точно совпадать с количеством данных, определенным приемником. Типы данных для гетерогенных распределенных систем, соответствующих типам данных MPI, могут отличаться.

Завершение коллективной операции может происходить, как только участие процесса в данной операции окончено. Завершение в таком случае указывает на то, что вызвавшая программа может изменять буфер передачи, но при этом это не значит, что другие процессы в группе завершили выполнение данной функции или даже начали ее. Поэтому коллективная операция имеет некоторый эффект синхронизации всех вызывающих процессов. Процессы полностью синхронизирует выполнение функции барьера (**MPI_barrier**). Коллективные операций могут использовать в качестве аргумента те же коммуникаторы, что и коммуникации типа «точка-точка». MPI гарантирует, что созданные сообщения от имени коллективной передачи не будут пересекаться с сообщениями, созданными для передачи типа «точка-точка».

1.1. Коммуникатор и виды обмена

Ключевой особенностью коллективных функций является использование группы или групп участвующих процессов. Функции при этом не имеют явного идентификатора группы в качестве аргумента, для этой цели используется коммуникатор. Существуют два типа коммуникаторов:
- коммуникаторы процессов внутри одной группы(**intra-communicators**),
- коммуникаторы процессов для нескольких групп (**inter-communicators**).

В упрощённом варианте интра-коммуникатор – это обычный коммуникатор для одной группы процессов, соединенных контекстом, в то время как интеркоммуникатор определяет две отдельные группы процессов соединенных контекстом (см. рис. 1.1). Интер-коммуникатор позволяет передавать данные между процессами из разных интра-коммуникаторов (групп).

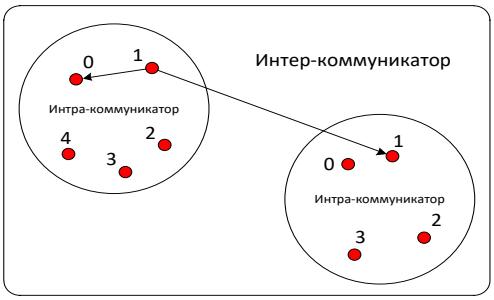


Рисунок 1.1 – Интра- и интер-коммуникаторы

Для выполнения коллективной операции (операции обмена) ее должны вызывать все процессы в группе, определенной интра-коммуникатором. В MPI используются коллективные операции интра-коммуникатора **All-To-All.** Каждый процесс в группе получает все сообщения от каждого процесса в этой же группе. Функции, используемые при обмене: **MPI_Allgather**, **MPI_Allgatherv**; **MPI_Alltoall**, **MPI_Alltoallv**, **MPI_Alltoallw**; **MPI_Allreduce**, **MPI_Reduce_scatter**; **MPI_Barrier**.

All-To-One. Все процессы группы формируют данные, но лишь один принимает их. Функции, используемые при обмене: **MPI_Gather**, **MPI_Gatherv**, **MPI_Reduce**.

One-To-All. Один процесс группы формирует данные, все процессы этой же группы принимает его. Функции, используемые при обмене: **MPI_Bcast**, **MPI_Scatter**, **MPI_Scatterv**.

Коллективные коммуникации для интер-коммуникаторов легче всего описать для двух групп. К примеру, операция «все ко всем» (**MPI_Allgather**) может быть описана как сбор данных от всех членов одной группы и получение результата всеми членами другой группы (рис.1.2).

Для интра-коммуникаторов группа обменивающихся процессов одна и та же. Для интер-коммуникаторов они различны. Для операций «все ко всем», каждой такой операции соответствуют две фазы, так что она имеет симметрию, полнодуплексное поведение. Следующие коллективные операции также применяются для интер-коммуникаций: **MPI_Barrier**, **MPI_Bcast**, **MPI_Gather**, **MPI_Gatherv**, **MPI_Scatter**, **MPI_Scatterv**, **MPI_Allgather**, **MPI_Allgatherv**, **MPI_Alltoall**, **MPI_Alltoallv**, **MPI_Alltoallw**, **MPI_AllReduce**, **MPI_Reduce**, **MPI_Reduce_scatter.**

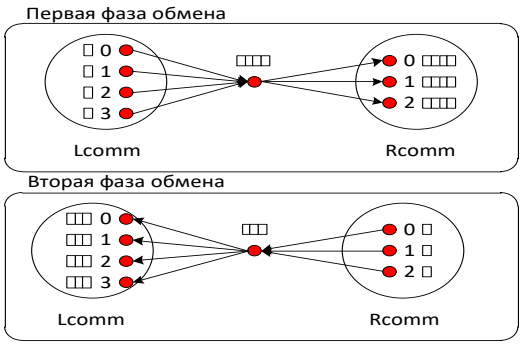


Рисунок 1.2 – Интер-коммуникатор **allgather**

Все процессы в обеих группах, определённых интер-коммуникатором, должны вызывать коллективные функции. Для коллективных операций интер-коммуникаторов если операция реализует обмен «все к одному» или «от одного ко всем», перемещение данных является однонаправленным. Направление передачи данных указывается в аргументе корня специальным значением. В этом случае для группы, содержащей корневой процесс, все процессы должны вызывать функцию, использующую для корня специальный аргумент. Для этого корневой процесс использует значение **MPI_ROOT**; все другие процессы в той же группе, что и корневой процесс, используют **MPI_PROC_NULL**. Если операция находиться в категории «все ко всем», то перенос является двунаправленным.

1.2. Барьерная синхронизация

Для синхронизации выполнения всех процессов группы используется следующая функция:

Int MPI_Barrier (MPI_Comm comm);

Атрибутом этой функции является: in comm **– коммуникатор;**

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Barrier** блокирует процесс, который его вызвал до того момента, пока все члены группы не вызовут эту же операцию. В любом процессе функция может завершиться только после того, как все члены группы достигли выполнения этой функции. Если аргумент **comm –** интер-коммуникатор, функция **MPI_Barrier** включает в себя две группы и завершается в процессах первой группы только после того, как все члены другой не начали выполнять данную функцию (и наоборот). При этом процесс может завершить выполнение данной функции до того, как все процессы в его собственной группе не начали ее выполнять.

1.3. Основные функции, реализующие широковещание

Чтобы передать значение от одного процесса всем процессам его группы (совершить **широковещательную рассылку**) используется функция, синтаксис которой имеет вид:

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

Атрибутами этой функции являются:

in out **buffer** - указатель на буфер; in **count** - количество элементов в буфере; in **datatype** - тип данных буфера; in **Root** - ранг корня широковещательной рассылки; in **Comm** – коммуникатор;

Если аргумент **comm** является интра-коммуникатор, функция **MPI_Bcast** распространяет сообщение от процесса с рангом **root** ко всем процессам группы, включая и себя. Это реализуется всеми членами группы, используя одни и те же аргументы **comm** и **root**. После завершения содержимое корневого буфера является скопированным во всех других процессах.

Рисунок 1.3 – Реализация функции широковещательной рассылки

Если аргумент **comm** является предварительно созданными интер-коммуникатором, тогда вызов включает в себя все процессы в интер-коммуникаторе, но при этом процесс-корень находиться в одной группе (группе A), а всем процессам другой группы (группы B) должен быть указан в качестве аргумента идентификатор корня (**root**) из группы, где находиться источник рассылки (группа A). Корневой процесс передает значение **MPI_ROOT** в аргумент **root**. Все другие процессы в группе A передают значение **MPI_PROC_NULL** в аргумент **root**. Данные распространяются от корня ко всем процессам в группе B. Для того чтобы собрать (**редукция**) значения со всех процессов группы в одном процессе, используется функция:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **recvcount** - количество элементов одиночного приема (неотрицательное число, используется только корневым процессом); in **recvtype** - тип данных буфера приема (используется только корневым процессом); in **root** - ранг принимающего процесса; in **comm** - коммуникатор;

Если аргумент **comm** интра-коммуникатор, тогда каждый процесс (включая корневой процесс) отправляют содержимое собственного буфера отправки корневому процессу. Корневой процесс принимает сообщения и сохраняет их в ранжированном порядке. Вызов данной функции является идентичным тому, что каждый из **n** процессов в группе (включая коневой процесс) выполнили вызов **MPI_Send(sendbuf, sendcount, sendtype, root, …)**, а корень выполнил *n* вызовов **MPI_Recv().**

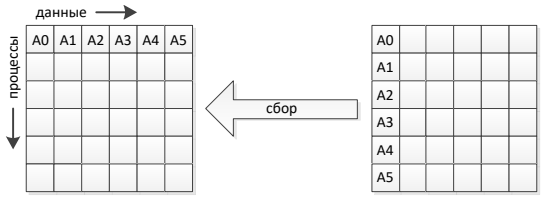


Рисунок 1.4 – Иллюстрация функции сбора

Если аргумент **comm** интер-коммуникатор, тогда вызов включает в себя все процессы интер-коммуникатора, но с одной группой (группа A), определяющей корневой процесс. Все процессы в другой группе (группа B) передают то же значение в аргумент **root**, который является корнем в группе A. В корневом процессе в качестве аргумента **root** указывается значение **MPI_ROOT**. Все другие процессы в группе A передают значение **MPI_PROC_NULL** в аргумент **root**. Данные собираются ото всех процессов в группе B к корню. Для сбора данных используется следующая функция:

Int MPI_Gatherv(void* sendbuf, intsendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных чисел (длина равна размеру группы), содержащий количество элементов принимаемых от каждого процесса (используется только корневым процессом); in **displs** - целочисленный массив (длина равна размеру группы, элемент **i** указывает смещение относительно начала массива **recvbuf**, в котором необходимо заменить входные данные от процесса **i** (используется только корневым процессом)); in **recvtype** - Тип данных буфера приема (используется только корневым процессом); in **root** - Ранг принимающего процесса;

Так как аргумент **recvcounts** является массивом, функция **MPI_Gatherv** расширяет функциональность операции **MPI_Gather**, допуская переменное количество данных в каждом процессе. Дополнительную гибкость дает аргумент **displs** для данных, размещаемых в корне.

Пример использования функции **MPI_Gather().**

Необходимо произвести сбор 100 целочисленных значений от каждого процесса в группе к корню (рис. 1.5):

```
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);
```

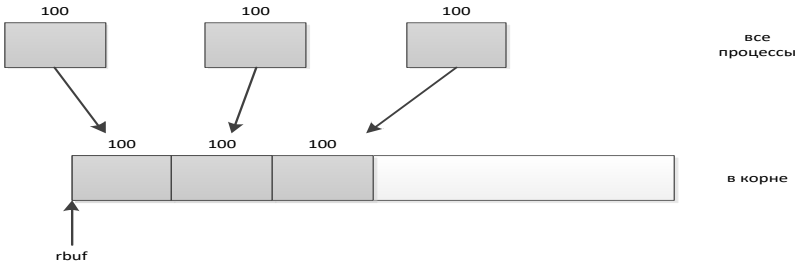


Рисунок 1.5 – Схема реализации сбора значений

Обратная функция для операции **MPI_Gather** следующая:
Int MPI_Scatter(void* sendbuf, intsendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
Атрибутами этой функции являются:
in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **recvcount** - количество элементов одиночного приема (используется только коневым процессом); in **recvtype** - тип данных буфера приема (используется только корневым процессом); in **root** - ранг принимающего процесса; in comm - Коммуникатор;

Если аргумент **comm** является интра-коммуникатор, результат можно проинтерпретировать так, как будто корень выполнил **n** операций отправки:
MPI_Send(sendbuf +i * sendcount * extent(sendtype), sendcount, sendtype, i, …);
и каждый процесс выполнил прием:
MPI_Recv(recvbuf, recvcount, recvtype, i, …);

Альтернативное описание является следующим: корень отсылает сообщение с помощью
MPI_Send(sendbuf, sendcount * n, sendtype, …);

Это сообщение расщепляется на **n** равных сегментов, **i**-ый сегмент отправляется **i**-ому процессу в группе, и каждый процесс принимает сообщение так, как описано выше. Буфер отправки игнорируется для всех некорневых процессов.

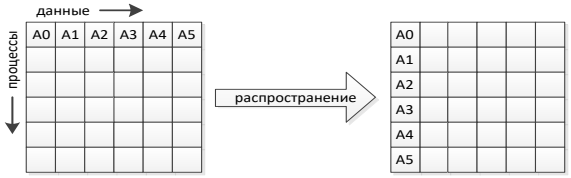


Рисунок 1.6 – Иллюстрация функции распространения MPI_Scatter ()

Если аргумент **comm** является интер-коммуникатором, тогда вызов включает в себя все процессы интер-коммуникатора, но только в одной группе (группа A) определяется корневой процесс. Все процессы в другой группе (группа B) передают одно и то же значение в аргумент **root**, которое является рангом корневого процесса в группе A. Корень передает значение **MPI_ROOT** в аргумент **root**. Данные распространяются от корня ко всем процессам в группе B. Чтобы совершить действия, обратные операции **MPI_Gatherv**, используется функция, синтаксис которой следующий:
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
Атрибутами этой функции являются:
in **sendbuf** - указатель на буфера отправки; in **sendcount** - количество элементов буфера отправки; in **displs** - целочисленный массив (длина–размер группы, элемент **i** указывает смещение относительно **sendbuf**, у которого необходимо взять данные, передаваемые процессу **I);** in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcount** - количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **root** - ранг принимающего процесса; in **comm** – коммуникатор;

Так как **sendcounts** является массивом, функция **MPI_Scatterv** расширяет функциональность **MPI_Scatter**, допуская переменное количество данных для отправки каждому процессу. Также дает дополнительную гибкость аргумент **displs**, указываемый для данных, которые берутся в корне.

Для того чтобы передать данные всем процессам, а не одному корню (как в функции **MPI_Gather**), используется следующая функция:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcount** - количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор.

Блок данных, отправленных от **j**-ого процесса, принимается каждым процессом и размещается в **j**-ом блоке буфера **recvbuf**. Если аргумент **comm** является интра-коммуникатором, результат вызова функции **MPI_Allgather** можно считать таким, что все процессы выполнили **n** вызовов: **MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);** где аргумент root принимает значения от 0 до n-1. Правила правильного использования функции **MPI_Allgather** аналогичны соответствующим правилам использования функции **MPI_Gather**.

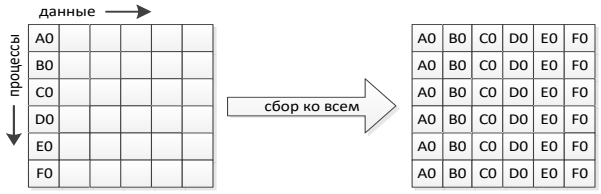


Рисунок 1.7 – Иллюстрация функции сбора ко всем

Если необходимо передать различное количество данных всем процессам, то используется функция, синтаксис которой следующий:

Int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфера отправки; out recvbuf - указатель на буфер приема; in **recvcounts** - количество элементов буфера приема; in **displs** -цЦелочисленный массив (длина равна размеру группы). Элемент **I** указывает смещение относительно **recvbuf**, где необходимо поместить входные данные от процесса **I;** in recvtype - Тип данных буфера приема; in comm – Коммуникатор;

Блок данных, отправленных от **j**-ого процесса, принимается всеми процессами и размещается в **j**-ом блоке буфера **recvbuf**. Для того чтобы каждый процессом отправил различные данные каждому процессу-приемнику, используется следующая функция:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcount** - количество элементов буфера отправки; in **sendtype** - тип данных буфераотправки; out **recvbuf** - указатель на буфер приема; in **recvcount**- количество элементов буфера приема; in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор;

Функция **MPI_Alltoall** является расширением функции **MPI_Allgather**, где **j**-ый блок, отправленный от **i**-ого процесса, принимается **j**-ым процессом и размещается в **i**-ом блоке **recvbuf**. Все аргументы на всех процесса являются значимыми. Аргумент **comm** должен иметь идентичное значение на всех процессах.

Рисунок 1.8 – Иллюстрация функции полного обмена

Если аргумент **comm** является интер-коммуникатором, тогда результат таков, что каждый процесс группы A отправляет сообщение каждому процессу группы B и наоборот.

Для того чтобы каждый процессом отправил данные различного размера каждому процессу-приемнику, используется следующая функция:

Int MPI_Alltoallv(void*sendbuf, int*sendcounts,int *sdispls,MPI_Datatypesendtype, void*recvbuf, int *recvcounts,int*rdispls,MPI_Datatype recvtype,MPI_Comm comm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; in **sendcounts** - массив неотрицательных целочисленных значений (длина равна размеру группы) указывающий количество элементов, которые необходимо отправить каждому процессу; in **sdispls** - целочисленный массив (длина – размер группы, элемент **j** указывает смещение относительно аргумента **sendbuf**, у которого необходимо взять данные передаваемые процессу **j);** in **sendtype** - тип данных буфера отправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных чисел (длина равна размеру группы), указывающий количество элементов, которое может быть принято от каждого процесса; in **rdispls** - целочисленный массив (длина равна размеру группы,

элемент **i** указывает смещение относительно аргумента **recvbuf**, где необходимо поместить входные данные от процесса **I);** in **recvtype** - тип данных буфера приема; in **comm** – коммуникатор;

1.4. Глобальные операции предварительной обработки

Функции, описываемые в этом разделе, выполняют глобальные операции предварительной обработки (к примеру, сумма, максимум и т.д.) для всех членов группы. Операция предварительной обработки может быть как одной из списка, так и определенной пользователем. Глобальные функции предварительной обработки имеют несколько разновидностей:

- обработка, которая возвращает результат одному процессу группы,
- обработка, которая возвращает результат всем членам группы,
- две операции сканирования,
- операция одновременной обработки и распространения.

Синтаксис первой из них следующий:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; out **recvbuf** - указатель на буфер приема (используется только корневым процессом); in **count** - количество элементов буфера отправки; in **datatype** - тип данных буфера отправки; in **op** - операция предварительной обработки; in **root** - ранг корневого процесса; in **comm** – коммуникатор;

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Reduce** собирает все элементы, переданные во входной буфер каждого процесса в группе, выполняет операцию **op** и возвращает значение в выходной буфер процесса с рангом **root**. Входной буфер определяется аргументами **sendbuf**, **count** и **datatype**; выходной буфер – аргументами **recvbuf**, **count** и **datatype**; оба буфера имеют одинаковое количество элементов с одним и тем же типом. Функция вызывается всеми членами группы, используя одни и те же аргументы **count**, **datatype**, **op**, **root** и **comm**. Поэтому все процессы обеспечивают функцию входными (выходным для корня **root**) буферами одной и той же длины, с элементами одинакового типа. Каждый процесс может предоставлять один элемент или последовательность элементов, в таком случае комбинированная операция выполняется над каждым элементом последовательности:

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, \qquad 0 \le j < n;$$

где $\otimes$ – операция, задаваемая при вызове функции **MPI_Reduce**. Пример выполнения операции редукции при суммировании пересылаемых данных трех процессов. В каждом сообщении 4 элемента, сообщения собираются на процессе с рангом 2 (рис. 1.9):

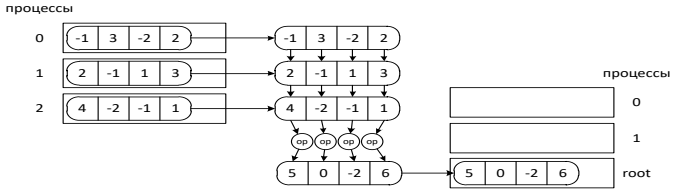


Рисунок 1.9 – Пример выполнения операции редукции

Список предопределенных функций рассмотрен в табл. 1.1.Каждая функция может выполняться только с определёнными типами данных. В дополнении к этому пользователь может определить собственную операцию, которая может быть перегружена с несколькими типами.

Операция **op** всегда предполагается как ассоциативная. Все предопределенные операции также предполагаются, что являются и коммутативными. «Каноническое» определение порядка предварительной обработки обуславливается рангом процессов в группе. Как бы то ни было, реализация может получать преимущество ассоциативности или ассоциативности и коммутативности при изменении порядка обработки, нов таком случае может измениться результат предварительной обработки для операций, которые не строго ассоциативные или коммутативные, такие например как сложение чисел с плавающей точкой.

Аргумент **datatype** функции **MPI_Reduce** должен быть совместимым с операцией **op**. Более того аргумент **datatype** и операция **op** для предопределённых операторов должны быть одинаковы во всех процессах.

Необходимо заметить, что для пользователя возможно определение различных пользовательских операций, но тогда MPI не ведет контроль над тем, какая операция используется над каким операндом.

Таблица 1.1

Предопределенные операций в Reduce

MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма

MPI_PROD	произведение
MPI_LAND	логическое И
MPI_BAND	побитовое И
MPI_LOR	логическое ИЛИ
MPI_BOR	побитовое ИЛИ
MPI_LXOR	логическое исключающее ИЛИ (xor)
MPI_BXOR	побитовое исключающее ИЛИ (xor)
MPI_MAXLOC	максимальное значение и местоположение
MPI_MINLOC	минимальное значение и положение

Оператор **MPI_MINLOC** используется для вычисления глобального минимума, а также индекса прикрепленного к минимальному значению. Операция **MPI_MAXLOC** подобным образом вычисляет глобальный максимум и индекс. Операция, которая определяет **MPI_MAXLOC** следующая:

$$\binom{u}{i} \circ \binom{v}{j} = \binom{w}{k}$$

где

$$w = max(u, v)$$

и

$$k = \begin{cases} i & if\ u > v \\ min(i,j) & if\ u = v \\ j & if\ u < v \end{cases}$$

Операция **MPI_MINLOC** определена следующим образом:

$$\binom{u}{i} \circ \binom{v}{j} = \binom{w}{k}$$

где

$$w = min(u, v)$$

и

$$k = \begin{cases} i & if\ u < v \\ min(i,j) & if\ u = v \\ j & if\ u > v \end{cases}$$

Пример использования функции Reduce:

```
// каждый процесс имеет массив 30 double: ain[30]
double ain[30], aout[30];
int ind[30];
struct
{
   double val;
   int rank;
} in[30], out[30];
int i, myrank, root;
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i)
{
   in[i].val = ain[i];
   in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
// В этой точке ответ находится в корневом процессе
if (myrank == root)
{
   // read ranks out
   for (i = 0; i < 30; ++i)
   {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;
   }
}
```

Каждый процесс содержит массив 30 вещественных чисел. Для каждого из 30 местоположений вычисляется значение и ранг процесса содержащего наибольшее значение.

Для того, чтобы соединить определенные пользователем функции предварительной обработки с дескриптором операции op, используется следующая функция: int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op);

Атрибутами этой функции являются:

in **Function** - определенная пользователем функция; in **Commute** - True если функция коммутативная, false в любом другом случае; out **Op** – операция;

Пользовательские определенные операции предполагаются ассоциативными. Если аргумент **commute** = **true**, тогда операция должна быть как коммутативной, так и ассоциативная. Если аргумент **commute** = **false**, тогда порядок операндов фиксирован и определяется восходящим порядком рангов процессов, начиная с нулевого процесса. Если аргумент **commute** = **true**, тогда порядок вычисления может быть изменен. Аргумент **function** является пользовательской функцией, которая должна иметь следующие четыре аргумента: **invec**, **inoutvec**, **len** и **datatype**.

Прототип выглядит следующим образом:

void MPI_User_function(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);

Аргументы **invec** и **inoutve –** массивы с **len** элементами, которые функция комбинирует. Результат предварительной обработки сохраняет значения в аргументе **inoutvec**. Каждый вызов функции ведет к поточечному вычислению оператора предварительной обработки **len** элементов.

Пример использования определенных пользователем операций с использованием интра-коммуникатора:

```
typedef struct {
   double real, imag;
} Complex;
// определенная пользователем функция
void myProd(Complex *in, Complex *inout, int *len, MPI_Datatype *dptr)
{
   int i;
   Complex c;
   for (i=0; i< *len; ++i)
   {
      c.real = inout->real*in->real - inout->imag*in->imag;
      c.imag = inout->real*in->imag + inout->imag*in->real;
      *inout = c;
      in++; inout++;
   }
}
// каждый процесс имеет массив 100 Complex-ов
   Complex a[100], answer[100];
   MPI_Op myOp;
   MPI_Datatype ctype;
   // объяснение MPI как тип Complex определен
   MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
   MPI_Type_commit(&ctype);
   // создание комплексного произведения пользовательской операцией
   MPI_Op_create(myProd, 1, &myOp);
   MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
   /* В этой точке ответ, который состоит из 100 Complex-ов, находящийся в корневом процессе */
```

Функция, которая относится ко второй разновидности редукции, имеет следующий синтаксис:

Int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Opop, MPI_Commcomm);

Атрибутами этой функции являются:

in **sendbuf** - указатель на буфер отправки; out **recvbuf** - указатель на буфер приема; in **count** - количество элементов буфера отправки (неотрицательное целочисленное значение); in **datatype** - тип данных буфера отправки; in **op** - операция предварительной обработки; in **comm** – коммуникатор;

Если аргумент **comm–**интра-коммуникатор, функциия **MPI_Allreduce** ведет себя также как и функция **MPI_Reduce** за исключением того, что после выполнения результат содержится в буфере приема всех членов группы.

Если аргумент **comm** является интер-коммуникатором, тогда результат предварительной обработки данных, обеспеченных процессами в группе A,сохраняются в каждом процессе группы B, и наоборот. Обе группы должны обеспечить аргументы **count** и **datatype**.

MPI включает в себя варианты операций предварительной обработки, где после выполнения результат распространяется ко всем процессами в группе. Один вариант распространяет блоки одинокого размера всем процессам, в то время как другой вариант распространяет блоки, которые могут варьироваться по размеру для каждого процесса.

Вариант второй из них следующий:

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

Атрибутами этой функции являются:

in **sendbuf** - указатель набуферотправки; out **recvbuf** - указатель на буфер приема; in **recvcounts** - массив неотрицательных целочисленных значений (длина равна размеру группы), указывающий количество элементов распределенного результата для каждого процесса; in **datatype** - тип данных буфера отправки и приема; in **op** – операция; in **comm** – коммуникатор;

Функция **MPI_Reduce_scatter** является расширением **MPI_Reduce_scatter_block**, так как в отличие от нее распространяет блоки, которые могут отличаться в размере. Размеры блоков определяются массивом **recvcounts**, **i**-ый блок содержит **recvcounts[i]** элементов.

Если аргумент **comm** является интра-коммуникатором, функция **MPI_Reduce_scatter** сначала выполняет глобальную поэлементную предварительную обработку над вектором:

$$count = \sum_{i=0}^{n-1} recvcount[i]$$

где **n** является количеством процессов в группе аргумента **comm**. Функция вызывается всеми членами группы, используя одинаковые аргументы **recvcount**, **datatype**, **op** и **comm**. Результирующий вектор принимается, как **n** последовательных блоков, где количество элементов **i**-ого блока является значением **recvcount[i]**. Таким образом, **i**-ый блок отправляется процессу **I** и сохраняется в буфере приема, определенном с помощью аргументов **recvbuf**, **recvcount[i]** и **datatype**.

Если аргумент **comm–** интер-коммуникатор, тогда результат предварительной обработки данных предоставленных процессами группы А распространяется по процессам в группе В и наоборот. Внутри каждой группы все процессы имеют одинаковый аргумент **recvcounts**, входное количество (значение **count**) элементов сохраненных в буфере отправки. Результирующий вектор от другой группы распространяется в блоки **recvcount[i]** элементов по процессам в группе. Количество элементов **count** должно быть одинаково для двух групп.

1.5. Группы, контексты, коммуникаторы

Группы определяют упорядоченную совокупность процессов, каждый из которых имеет ранг, т.е. группа определяет низкоуровневые имена для коммуникаций между процессами. Поэтому группы определяют границы для имен процессов в коммуникации типа точка-точка. В дополнение, группы определяют границы для коллективных операций. Группы управляются отдельно от коммуникаторов в MPI, но только коммуникаторы могут быть использованы в коммуникационных операциях внутри групп.

Наиболее часто используемые средства для передачи сообщений в группах являются интра-коммуникаторами. Они содержат экземпляр группы, контексты коммуникаций (информацию для коммуникации) как для связи типа точка-точка, так и для коллективного типа связи, возможность включать в себя топологию и другие атрибуты. Интер-коммуникаторы реализуют взаимодействие между двумя не накрадывающимися друг на друга группами. Когда приложение построено образованием нескольких параллельных модулей, удобно разрешать одному модулю взаимодействовать с другим, используя локальные ранги для адресации внутри второго модуля. Это особенно удобно в клиент-серверной обрабатывающей парадигме, где как клиент, так и сервер являются параллельными. Интер-коммуникаторы связывают две группы вместе с коммуникационным контекстом, разделяемым обеими группами. Для интер-коммуникаторов использование контекстов обеспечивают возможность иметь обособленные защищенные «среды» передачи сообщений между двумя группами. Отправка в локальной группе всегда принимается в удалённой группе, и наоборот. Процесс установления различия организует система. Локальная и удалённая группы определяют пункты отправки и назначения для интер-коммуникатора.

1.6. Основная концепция работы с несколькими процессами

Группа является упорядоченным набором идентификаторов процессных (далее процессы). Каждый процесс в группе ассоциируется с целочисленным рангом. Ранги являются последовательными и начинаются с нуля. Группа используется внутри коммуникатора для описания участников коммуникационной «среды» и для ранжирования этих участников (т.е. происходит раздача им уникального имени внутри «среды» коммуникации).

Существует специальная предопределенная группа: **MPI_GROUP_EMPTY**, которая является группой без участников. Предопределённая константа **MPI_GROUP_NULL** является значением, используемым для недопустимого идентификатора группы.

MPI-коммуникационные операции ссылаются на коммуникаторы для определения границ и «коммуникационной среды», в которой операции типа точка-точка или коллективного типа являются возможными. Каждый коммуникатор содержит группу допустимых участников; эта группа всегда включает в себя локальный процесс. Источник и место назначения сообщения определяются рангом процесса внутри группы. Для коллективных коммуникаций интра-коммуникатор определяет ряд процессов, которые участвуют в коллективной операции (и их порядок, когда это необходимо). Поэтому коммуникатор ограничивает «пространственные» границы коммуникации, и обеспечивает адресацию процессов независящую от машины посредством ранга.

После инициализации локальный процесс может общаться со всеми процессами интра-коммуникатора **MPI_COMM_WORLD** (включая и себя), определённого один раз **MPI_INIT** или

MPI_INIT_THREAD вызовами. Предопределенная константа **MPI_COMM_NULL** является значением, использующимся для недопустимого дескриптора коммуникатора.

В реализации статической модели процессов MPI, все процессы, которые участвуют в общении, являются доступными после инициализации. Для этого случая **MPI_COMM_WORLD** является коммуникатором всех процессов доступных для коммуникации. В реализации MPI, процессы начинают вычисления без доступа ко всем другим процессам. В таком случае **MPI_COMM_WORLD** является коммуникатором, объединяющим все процессы, с которыми присоединяющийся процесс может незамедлительно общаться. По этой причине **MPI_COMM_WORLD** может одновременно представлять отделенные группы в разных процессах.

Коммуникатор **MPI_COMM_WORLD** не может быть освобождён в течение функционирования процесса. Группа соответствующая этому коммуникатору нигде не проявляется, так как является предопределенной константой, но она может быть доступна через использование функции **MPI_Comm_Group**.

1.7. Управление группами

Чтобы получить информацию о группе, используются следующие функции:
int MPI_Group_size(MPI_Group group, int *size).
Атрибутами этой функции являются:
in **group** – группа; out **size** - количество процессов в группе.
int MPI_Group_rank(MPI_Group group, int *rank).
Атрибутами этой функции являются:
in **group** – группа; out **rank** - ранг вызывающего процесса в группе, или значение **MPI_UNDEFINED**,если процесс не является членом группы.

Следующая функция используется для определения относительной нумерации одних и тех же процессов в двух разных группах:
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)
Атрибутами этой функции являются:
in **group1** - первая группа; in **n** - количество рангов в массивах **ranks1**и **ranks2;** in **ranks1** - массив нулевого или более допустимых рангов в **group1;** in **group2** - вторая группа; out **ranks2** - массив соответствующих рангов в **group2**, значение **MPI_UNDEFINED,** если нет соответствий;

К примеру, если известны ранги текущих процессов в группе **MPI_COMM_WORLD**, может понадобиться узнать их ранги в подмножестве этой группы. Значение**MPI_PROC_NULL**является допустимым рангом для ввода в **MPI_Group_translate_ranks**, который возвращает значение **MPI_PROC_NULL** как переданный ранг.

Конструкторы группы используются для создания подмножества и расширения существующих групп. Эти конструкторы создают новые группы из существующих групп. Существуют локальные операции и отдельные группы, которые могут быть определены на разных процессах; процесс может также определять группу, которая не включает себя. Устойчивое определение необходимо, когда группы используются как аргументы в функциях создания коммуникаторов. MPI не обеспечивает механизм для создания групп с нуля, их можно создать только из других до этого определенных групп. Основная группа, с помощью которой строятся все остальные группы, является группа, ассоциированная с начальным коммуникатором **MPI_COMM_WORLD**, доступная через функцию **MPI_Comm_group**:
Int MPI_Comm_group(MPI_Commicomm, MPI_Group *group);
Атрибутами этой функции являются:
in **comm** – коммуникатор; out **group** –группа, соответствующая **comm;**

Следующие функции позволяют создавать новые группы:
Int MPI_Group_union(MPI_Groupgroup1, MPI_Groupgroup2,
MPI_Group *newgroup);
Атрибутами этой функции являются:
in group1 - первая группа; in group2 - вторая группа; out newgroup - группа объединения.
Int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
Атрибутами этой функции являются:
in **group1** - первая группа; in **group2** - вторая группа; out **newgroup** - группа пересечения.
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup) **Атрибутами этой функции являются:**
in **group1** - первая группа; in **group2** - вторая группа; out **newgroup** - группа разницы;

Операции определены следующим образом:
1) в **MPI_Group_Union()** все элементы первой группы (**group1**), затем все элементы второй группы (**group2**), которых нет в первой группе.
2) в **MPI_Group_Intersection()** все элементы первой группы, которые также есть и во второй группе, упорядоченные как в первой группе.
3) в **MPI_Group_Difference()** все элементы первой группы, которых нет во второй группе, упорядоченные как в первой группе.

Для этих операций порядок процессов выходной группы определяется главным образом порядком в первой группе (если возможно) и , если необходимо, порядком во второй группе. Ни объединение, ни пересечение не являются коммутативными, но обе являются ассоциативными. Чтобы включить процессы в новую группу с определенными рангами используется следующая функция:

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

Атрибутами этой функции являются:

in **group** – группа; in **n** - количество элементов в массиве рангов (и размер **newgroup**); in **ranks** - ранги процессов в **group**, которые должны появиться в **newgroup;**out **newgroup** - новая группа производная от вышеупомянутой, в порядке определения **ranks;**

Функция **MPI_Group_incl()** создает группу **newgroup**, которая состоит из **n** процессов в **group** с рангами **ranks[0]**, …, **ranks[n-1]**; процесс с рангом **i** в **newgroup** является процессом с рангом **ranks[i]** в **group**. Каждый из **n** элементов ранга **ranks** должен быть допустимым рангом в **group**, и все элементы должны быть индивидуальными, иначе программа является ошибочной. Если **n = 0**, тогда группа **newgroup** является **MPI_GROUP_EMPTY**. Эти функции могут, к примеру, использоваться для переупорядочивания элементов группы, или для сравнения. Для того, чтобы исключить процессы из группы с определенными рангами используется следующая функция:

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

Атрибутами этой функции являются:

in **group** – группа; in **n** - количество элементов в массиве рангов (и размер **newgroup**); in **ranks** - массив целочисленных рангов в **group**, которые не должны появиться в **newgroup;** out **newgroup** - новая группа производная от вышеупомянутой, сохраняющая порядок определенный **group;**

Функция **MPI_Group_excl** создает группу процессов **newgroup** путем удаления из **group** этих процессов с рангами **ranks[0]**, …, **ranks[n-1]**. Упорядочивание процессов в **newgroup** является идентичным упорядочиванию в **group**. Каждый из **n** элементов рангов **ranks** должен быть допустимым в **group.** Если **n = 0**, тогда группа **newgroup** является идентичной **group**.

1.8. Управление коммуникаторами

Операции, которые получают доступ к коммуникаторам, являются локальными и их выполнение не требует взаимодействия процессов. Операции, которые создают коммуникаторы, являются коллективными и могут требовать взаимодействия между процессами. Для того чтобы сравнить два коммуникатора используется следующая функция:

int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result);

Атрибутами этой функции являются:

in **comm1** - первый коммуникатор; in **comm2** - второй коммуникатор; out **result** – результат.

Результат будет иметь значение **MPI_IDENT**, если аргументы **comm1** и **comm2** являются дескрипторами одного и того же объекта (идентичные группы и одинаковый контекст). Значение **MPI_CONGRUENT** будет результатом, если группы являются идентичными в компонентах и порядке рангов; эти коммуникаторы отличаются только контекстом. Значение **MPI_SIMILAR** является результатом, если члены группы обоих коммуникаторов одинаковы, но порядок рангов отличается. Результат будет равен значению **MPI_UNEQUAL** в любом другом случае. Следующие операции являются коллективными функциями, которые вызываются всеми процессами в группе или группах ассоциированных с аргументом **comm**.

MPI обеспечивает четыре функции конструирования коммуникатора, которые применяются к интра-коммуникаторам и интер-коммуникаторам. Функция конструирования **MPI_Intercomm_create** применяется только к интер-коммуникаторам. В интер-коммуникаторе группы называются левой и правой. Процесс в интер-коммуникаторе является членом либо левой, либо правой группы. С точки зрения этого группа процесса, членом которой он является, называется локальной; другая группа (относительно этого процесса) является удаленной (дистанционной). Метки левой и правой групп дают возможность указывать две группы в интер-коммуникаторе, которые не относятся к какому-либо конкретному процессу.

Для создания коммуникатора используется следующая функция:

Int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm) **;**

Атрибутами этой функции являются:

in **comm** – коммуникатор; in **group** - группа, которая является подмножеством группы; коммуникатора **comm;** out **newcomm** - новый коммуникатор**.**

Если аргумент **comm** является интра-коммуникатором, эта функция возвращает новый коммуникатор **newcomm** с коммуникационной группой, определённой аргументом **group**. Никакая кэшированная информации не распространяется от коммуникатора **comm** к новому коммуникатору **newcomm**. Каждый процесс должен выполнять вызов функции с аргументом **group**, который является подгруппой группы ассоциированной с аргументом **comm**. Процессы могут указывать различные значения для аргумента **group**. Если аргумент **comm** является интер-коммуникатором, тогда выходной коммуникатор также является интер-коммуникатором, где локальная группа состоит только из тех процессов, которые содержатся в

группе **group**. Аргумент **group** должен иметь те процессы в локальной группы входного интер-коммуникатора, который является частью коммуникатора **newcomm**. Все процессы в одной и той же локальной группе коммуникатора **comm** должны указывать одинаковое значение для аргумента **group**. Если группа **group** не указывает хотя бы один процесс в локальной группе интер-коммуникатора, или если вызывающий процесс не является включенным в аргумент **group**, возвращается **MPI_COM_NULL**.

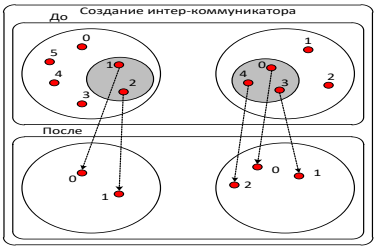


Рисунок 1.10 – Создание интер-коммуникатора

Для разделения коммуникатора используется следующая функция:
Int MPI_Comm_split(MPI_Commcomm, int color, int key, MPI_Comm *newcomm);
Атрибутами этой функции являются:
in **comm** – коммуникатор; in **color** - контроль распределения подмножества; in **key** - контроль распределения рангов; out **newcomm** - новый коммуникатор.

Функция разделяет группу, ассоциированную с аргументом **comm** на отдельные подгруппы, одна для каждого значения аргумента **color**. Каждая подгруппа содержит все процессы одинакового цвета. Внутри каждой группы процессы ранжируются в порядке определенном значением аргумента **key**, со связями, соответствующими их рангу в старой группе. Новый коммуникатор создается для каждой подгруппы и возвращается в аргументе **newcomm**. Процесс может обеспечивать значение аргумента color как **MPI_UNDEFINED**, в таком случае коммуникатор **newcomm** будет иметь значение **MPI_COMM_NULL**. Это коллективный вызов, но каждому процессу разрешается задавать различное значения **color** и **key**.

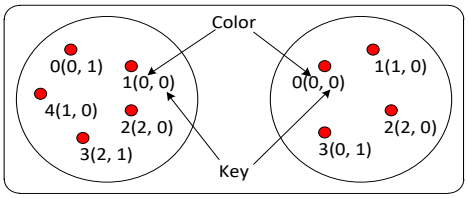


Рисунок 1.11 – Входной коммуникатор (**comm**)

С интра-коммуникатором **comm** вызов **MPI_Comm_create(comm, group, newcomm)** является эквивалентом вызова **MPI_Comm_split(comm, color, key, newcomm)**, где процессы, которое являются членами их аргумента **group** обеспечивают **color**= числу групп (основывается на уникальной нумерации всех отдельных групп) и **key** = рангу в группе, все процессы которые не являются членами их аргумента **group** обеспечивают **color**= **MPI_UNDEFINED**.

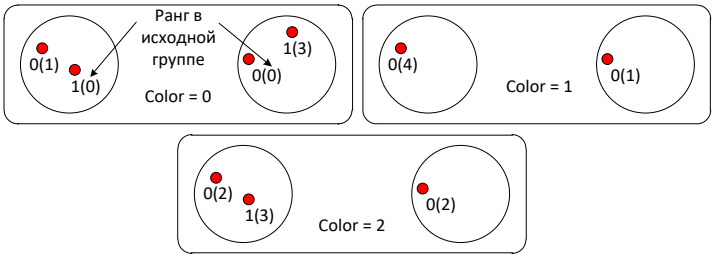


Рисунок 1.12 – Отделенные выходные коммуникаторы (**newcomm**)

Результат функции **MPI_Comm_split** на интер-коммуникаторе является таким, что процессы слева с одинаковым значением аргумента **color**, как у процессов справа, объединяются, чтобы создать новый интер-коммуникатор. Аргумент **key** описывает относительный ранг процессов на каждой стороне интер-коммуникатора. Для тех цветов, которые указываются только на одной стороне интер-коммуникатора, возвращается значение **MPI_COMM_NULL**. Для освобождения коммуникатора используется функция, синтаксис которой следующий:
Int MPI_Comm_free(MPI_Comm *comm)**;**
Атрибутом этой функции является: in comm **- коммуникатор, который необходимо освободить.**

Дескриптор устанавливается в значение **MPI_COMM_NULL**. Любые неоконченные операции, которые используют этот коммуникатор, будут завершены в нормальном режиме; объект освобожден фактически, только если не существует активных ссылок на него. Этот вызов применяется для интра- и интер-

коммуникаторов. Функция **MPI_Intercomm_create** используется для того, чтобы связать два интра-коммуникатора в интер-коммуникатор. Функция **MPI_Intercomm_merge** создает интра-коммуникатор, соединяя локальную и удаленную группы интер-коммуникатора. Частичное совпадение локальной и удаленной групп, которые связаны в интер-коммуникаторе, запрещено. Если существует частичное совпадение, то тогда программа является ошибочной и вероятней всего ведет к взаимной блокировке.
Функция **MPI_Intercomm_create** может быть использована для создания интер-коммуникатора из двух существующих интра-коммуникаторов в следующей ситуации: хотя бы один выбранный член из каждой группы («лидер группы») имеет возможность взаимодействовать с выбранным членом из другой группы; другими словами существует так называемый равноправный коммуникатор (“peer” communicator), которому принадлежат оба лидера, и каждый лидер знает ранг другого лидера в этом равноправном коммуникаторе. Кроме того члены каждой группы знают ранг их лидера.
Построение интер-коммуникатора из двух интра-коммуникаторов требует вызовов отдельных коллективных операций в локальной группе и в удаленной группе, также как передача типа «точка-точка» между процессом в локальной группе и процессом в удалённой группе.
Алгоритм создания интер-коммуникатора (для локальных групп с последующим обменом между ними): 1) формирование групп процессов для коммуникатора по умолчанию (**MPI_COMM_WORLD**); 2) исключение процессов из групп (формирование ограниченных групп процессов); 3) создание коммуникатора для обмена внутри группы (связывание интра-коммуникатора с каждой из групп); 4) создание интер-коммуникатора.
Синтаксис функции создания интер-коммуникатора следующий:
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag,
MPI_Comm *newintercomm);
Атрибутами этой функции являются:
in **local_comm** - локальный интра-коммуникатор; in **local_leader** - ранг лидера локальной группы в коммуникаторе **local_comm;**in **peer_comm** - Равноправный коммуникатор; используется только процессом **local_leader; i**n **remote_leader** - ранг лидера удаленной группы в коммуникаторе **peer_comm**; используется только процессом **local_leader;** In **tag** - метка «безопасности»; out **newintercomm** - новый интер-коммуникатор.

Вызов данной функции является коллективным через объединение локальной и удаленной групп. Процессы должны обеспечивать одинаковые аргументы **local_comm** и **local_leader** внутри каждой группы. Групповое значение не разрешено для аргументов **remote_leader**, **local_leader** и **tag**.
Этот вызов использует коммуникацию типа «точка-точка» с коммуникатором **peer_comm** и с меткой **tag**.
Для создания интра-коммуникатора используется следующая функция:
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
MPI_Comm *newintracomm);
Атрибутами этой функции являются:
in **intercom** - Интер-коммуникатор; in **high** – флаг; out **newintracomm** - новый интра-коммуникатор.

Эта функция создает интра-коммуникатор из объединения двух групп, которые ассоциированы с коммуникатором **intercomm**. Все процессы должны иметь одинаковое значение **high** внутри каждой группы. Если процессы в одной группе указывают значение **false** для аргумента **high**, и процессы в другой указывают значение **true** для аргумента **high**, тогда объединение упорядочивается так, что «нижняя» группа располагает до «верхней» группы. Если все процессы указывают одинаковое значение для аргумента **high**, тогда порядок в объединении является произвольным. Вызов данной функции является блокирующимся и коллективным внутри объединения двух групп.
Рассмотрим пример конвейера трех групп. Группы 0 и 1 являются связанными. Группы 1 и 2 также являются связанными. По этой причине группа 0 требует один интер-коммуникатор, группа 1 требует два интер-коммуникатора, и группа 2 требует один интер-коммуникатор. Код программы следующий:

```
int main(int argc, char **argv)
{
   MPI_Comm myComm;       // интра-коммуникатор локальной подгруппы
   MPI_Comm myFirstComm;   // интер-коммуникатор
   MPI_Comm mySecondComm;  // второй интер-коммуникатор (группа 1 только)
   int membershipKey;
   int rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // Пользовательский код должен генерировать membershipKey
   // в диапозоне [0, 1, 2]
   membershipKey = rank % 3;
   // Построение интракоммуникатора для локальной подгруппы
MPI_Comm_split(MPI_COMM_WORLD,membershipKey, rank, &myComm);
   // Построение интер-коммуникатора. Метки жестко закодированные
```

```
if(membershipKey == 0)
{
   // Группа 0 связывается с группой 1
   MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 1, &myFirstComm);
}
else
   if (membershipKey == 1)
   {
      // Группа 1 связывается с группами 0 и 2
  MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0, 1,
         &myFirstComm);
 MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2, 12,
         &mySecondComm);
   }
   else
      if (membershipKey == 2)
      {
         // Группа 2 связывается с группа 1
         MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1, 12,
            &myFirstComm);
      }
switch(membershipKey)
{
case 1:
   MPI_Comm_free(&mySecondComm);
case 0:
case 2:
   MPI_Comm_free(&myFirstComm);
   break;
}
MPI_Finalize(); }
```

ИСЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ ФОРМИРОВАНИЯ ВИРТУАЛЬНЫХ ТОПОЛОГИЙ ВЫЧИСЛИТЕЛЬНЫХ КЛАСТЕРОВ

Топология является дополнительным необязательным атрибутом, который может соответствовать интра-коммуникатору; топологии не могут быть добавлены к интер-коммуникатору. Виртуальная топология вычислительного кластера – это удобный механизм именования для процессов группы внутри коммуникатора. Как было сказано ранее, группа в MPI является коллекцией *n* процессов. Каждый процесс в группе имеет ранг от **0** до *n-1*. Во многих параллельных приложениях линейное ранжирование процессов недостаточно отображает логическую коммуникационную модель взаимодействия процессов (которая обычно определяется проблемой геометрии топологии и используемым алгоритмом нумерации). Часто процессы организуются в топологические модели, такие как двух- или трех-мерные сетки. В общем виде логическая организация процессов описывается графом. Такая логическая организация процессов называется «виртуальной топологией». Существует четкое различие между виртуально топологией процессов и топологией физической аппаратуры. Виртуальная топология может быть использована системой для распределения процессов на физических процессорах, если это помогает улучшить коммуникационную производительность на данной машине. Описание виртуальной топологии, с другой стороны, зависит только от приложения и является машинно-независимой.

1.1. Виртуальные топологии

Коммуникационная модель взаимодействия процессов может быть представлена в виде графа. Узлы представляют собой процессы, ребра соединяют процессы, которые взаимодействуют друг с другом. MPI обеспечивает передачу сообщений между любой парой процессов в группе. После создания связей между процессами (в виртуальной топологии) отсутствуют специальные условия для открытия канала, поэтому недостающее ребро в определённом пользователем графе не запрещает соответствующую передачу между процессами. Данное состояние говорит только о том, что связь является игнорируемой в виртуальной топологии. Такая стратегия предполагает, что топология определяет способ именования путей коммуникации. Указание виртуальной топологии в виде графа является достаточным для всех приложений. Большая часть всех параллельных приложений использует топологию процессов, таких как кольцо, сетку с двумя или более измерениями, или торы. Эти структуры являются совершенно различными по количеству измерений и количеству процессов в каждом координатном направлении. Координаты процесса в координатной структуре начинают свое исчисление с **0**. Измерение по строкам является всегда более

важным в координатной структуре. Это значит что отношение между рангами групп и координатами для четырех процессов в сетке (2*2) является следующим:

координата (0, 0): ранг 0
координата (0, 1): ранг 1
координата (1, 0): ранг 2
координата (1, 1): ранг 3

1.2. Конструкторы топологий

Для создания декартовой топологии используется функция:

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

Атрибутами этой функции являются:

in **comm_old** - входной коммуникатор; in **ndims** - количество измерений декартовой сетки; in **dims** - целочисленный массив размера **ndims**, указывающий количество процессов в каждом измерении; in **periods** - логический массив размера **ndims**, указывающий является ли сетка периодической (**true**) или нет (**false**) в каждом измерении; in **reorder** - ранжирование может быть переупорядочено (**true**) или нет (**false**); out **comm_cart** - коммуникатор с новой декартовой топологией.

Функция возвращает дескриптор нового коммуникатора, к которому прикреплена информация о декартовой топологии. Если аргумент **reorder** равен значению **false**, тогда ранг каждого процесса в новой группе является идентичным его рангу в старой группе. В противном случае функция может переупорядочить процессы. Если общий размер декартовой сетки является меньше, чем размер группы коммуникатора **comm**, тогда некоторые процессы возвращаются как **MPI_COMM_NULL**, по аналогии с функцией **MPI_Comm_split**. Если аргумент **ndims** равен нулю, тогда создается декартовая нуль-мерная топология. Вызов является ошибочным, если он определяет сетку больше чем размер группы, или если аргумент **ndims** является отрицательным. Для декартовых топологий функция **MPI_Dims_create** помогает пользователю выбрать сбалансированное распределение процессов по каждому координатному направлению в зависимости от количества процессов в группе и от дополнительных ограничений, которые могут быть указаны пользователем. Одним из использований является разбитие всех процессов на **n**-мерную топологию. Вид вызова функции следующий:

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

Атрибутами этой функции являются:

in **nnodes** - количество узлов в сетке; in **ndims** - количество декартовых измерений; inout **dims** - целочисленный массив размера **ndims** указывающий количество узлов в каждом измерении;

Элементы в массиве **dims** определяются для описания декартовой сетки с **ndims** измерениями и суммой **nnodes** узлов. Измерения устанавливаются так, чтобы быть как можно ближе друг к другу, используя подходящий алгоритм делимости. Если элемент **dims[i]**является положительным числом, функция не будет модифицировать количество узлов в измерении **i**; только те элементы, где значение **dims[i] = 0**, будут модифицированы вызывающей стороной.

Отрицательные входные значения **dims[i]** являются ошибочными. Ошибка будет происходить, если аргумент **nnodes** не является произведением:

$$\prod_{i,dims[i]\neq 0} dims[i].$$

Функция **MPI_Dims_create** является локальной.

Таблица 1.1

Пример использования функции **MPI_Dims_create**

Dims до вызова	вызов функции	**Dims** на выходе
(0,0)	**MPI_Dims_create(6,2,dims)**	(3, 2)
(0, 0)	**MPI_Dims_create(7,2,dims)**	(7, 1)
(0, 3, 0)	**MPI_Dims_Create(6,3,dims)**	(2, 3, 1)
(0, 3, 0)	**MPI_Dims_create(7,3,dims)**	Ошиб. вызов

Для создания топологии графа используется следующая функция:

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

Атрибутами этой функции являются:

in **comm_old** - входной коммуникатор; in **nnodes** - количество узлов в графе; in **index** - целочисленный массив, описывающий степени узлов; in **edges** - целочисленный массив, описывающий ребра графа; in **reorder** - ранжирование может быть переупорядочено (**true**) или нет (**false**); out **comm_graph** - коммуникатор с добавленной топологией графа.

Функция возвращает дескриптор нового коммуникатора, к которому прикреплена информация о топологии графа. Если аргумент **reorder** равен значению **false**, тогда ранг каждого процесса в новой группе является идентичным его рангу в старой группе. Если размер (аргумент **nnodes)** графа меньше чем размер группы коммуникатора **comm**, тогда некоторые процессы будут возвращены как **MPI_COMM_NULL**, по аналогии с **MPI_Cart_create**. Если граф пустой, т.е. аргумент **nnodes = 0**, тогда значение

MPI_COMM_NULL возвращается во всех процессах. Вызов является ошибочным, если он указывает граф, который является больше, чем размер группы входного коммуникатора. Три параметра **nnodes**, **index** и **edges** определяют структуру графа. Аргумент **nnodes** является количеством узлов графа. Узлы нумеруются от **0** до **nnodes-1**. При этом ***i***-ый элемент массива **index** сохраняет общее число соседей первых ***i*** узлов графа. Список соседей узлов **0**, **1**, …, **nnodes-1** сохраняется в последовательности массива **edges**. Массив **edges** является развернутым представление списка ребер. Общее количество элементов в аргументе **index** является равным **nnodes,**общее количество элементов в аргументе **edges** является равным количеству ребер графа.

Пример: Предположим, что существует четыре процесса 0, 1, 2, 3 со следующей матрицей смежности:

процессы	соседи
0	1, 3
1	0
2	3
3	0, 2

Тогда, входные аргументы следующие:

nnodes = 4

index = 2, 3, 4, 6

edges = 1, 3, 0, 3, 0, 2

Поэтому в С элемент **index[0]** является степенью нулевого узла, а **index[i] – index[i-1]** является степенью узла **i**, где **i=1**, …, **nnodes-1**; список соседей нулевого узла сохраняется в элементе **edges[j]**, для **0**≤ **j** ≤ **index[0]-1** исписок соседей узлов **i (**где**i>0)**, сохраняются в элементе **edges[j]**, для**index[i-1]** ≤ **j**≤ **index[i]-1**.

1.3. Реализация топологических запросов

Если топология была определена одной из предыдущих функции, тогда информация о топологии может быть просмотрена, используя функции запросов, которые являются локальными вызовами:

intMPI_Topo_test(MPI_Commcomm, int *status) **Атрибутами этой функции являются:**

in **comm** – коммуникатор; out **status** - тип топологии коммуникатора **comm;**

Функция возвращает тип топологии, которая назначена коммуникатору.

Выходное значение аргумента **status** является одним из следующих:

MPI_GRAPH	топология графа
MPI_CART	декартова топология
MPI_DIST_GRAPH	распределенная топология графа
MPI_UNDEFINED	топологии нет

Функции **MPI_Graphdims_get** и **MPI_Graph_get** получают информацию о топологии графа, которая была ассоциирована с коммуникатором через **MPI_Cart_create**:

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

Атрибутами этой функции являются:

in **comm** - коммуникатор с топологией графа; out **nnodes** - количество узлов в графе; out **nedges** -количество ребер в графе;

Функции **MPI_Cartdim_get** и **MPI_Cart_get** возвращают информацию о декартовой топологии, которая была ассоциирована с коммуникатором, используя функцию **MPI_Cart_create**:

int MPI_Cartdim_get(MPI_Comm comm, int *ndims);

Атрибутами этой функции являются:

in **comm** - коммуникатор с декартовой структурой; out **ndims** - количество измерений декартовой структуры.

Если аргумент **comm** ассоциирован с нуль-мерной декартовой топологией, функция **MPI_Cartdim_get** возвратит аргумент **ndims=0**, функция **MPI_Cart_get** при этом оставит все выходные аргументы неизмененными:

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,int *coords). **Атрибутами этой функции являются:**

in **comm** - коммуникатор с декартовой структурой; in **maxdims** - длина вектора **dims**, **period** и **cords** в вызывающей программе; out **dims** - количество процессов для каждого декартового измерения; out **periods** - периодичность для каждого декартового измерения; out **cords** - координаты вызывающего процесса в декартовой структуре;

Для того, чтобы на основе координат получить ранг процесса, используется следующая функция:

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

Атрибутами этой функции являются:

in **comm** - коммуникатор с декартовой структурой; in **cords** - целочисленный массив (размера **ndims**) определяющий декартовы координаты процесса; out **rank** - ранг указанного процесса;

Для группы процессов с декартовой структурой функция **MPI_Cart_rank** переведет логические координаты процесса в ранг процесса, как они использовались бы процедурами точка-точка.

Для обратного отображения, перевод ранга в координаты, используется функция, синтаксис которой следующий:

Int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords);

Атрибутами этой функции являются:

in **comm** - коммуникатор с декартовой структурой; in **rank** - ранг процесса внутри группы **comm; i**n **maxdims** - длина вектора **cords** в вызывающей программе; out **cords** - целочисленный массив (размера **ndims**) содержащий декартовы координаты указанного процесса.

Если аргумент **comm** ассоциирован снуль мерной декартовой топологией, аргумент **cords** будет неизменен.

Функции **MPI_Graph_neighbors_count** и **MPI_Graph_neighbors** обеспечивают информацию о смежности для общей топологии графа:

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

Атрибутами этой функции являются:

in **comm** - коммуникатор с топологией графа; in **rank** - ранг процесса в группе **comm;** out **nneighbors** - количество соседей указанного процесса.

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

Атрибутами этой функции являются:

in comm - коммуникатор с топологией графа; in rank - ранг процесса в группе **comm;** in maxneighbors - размер массива соседей; out neighbors - ранги процессов, которые являются соседями указанного процесса;

Возвращаемое количество и массив соседей для запрашиваемого ранга будет как включать всех соседей, так и отражать такой же порядок ребер, как было определено оригинальным вызовом функции **MPI_Graph_create**. Точнее данные функции будут возвращать значения, соответствующие аргументам **index** и **edges**, переданным в функцию **MPI_Graph_create.** Количество соседей возвращенное функцией **MPI_Graph_neighbors_count** будет равно **index[rank]–index[rank–1]**. Массив **neighbors**, возвращенный функцией **MPI_Graph_neighbors** будет от **edges[index[rank – 1]]** до **edges[index[rank – 1]]**.

ПАРАЛЛЕЛЬНЫЕ МЕТОДЫ МАТРИЧНОГО УМНОЖЕНИЯ

Постановка задачи

Умножение матрицы *A* размера *m*n* и матрицы *B* размера *n*l* приводит к получению матрицы *C* размера *m*l*, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj},\ 0 \le i < m,\ 0 \le j < l$$

Каждый элемент результирующей матрицы *C* есть скалярное произведение соответствующих строки матрицы *A* и столбца матрицы *B*:

$$c_{ij} = (a_i, b_j^T),\ a_i = (a_{i0}, a_{i1} \ldots, a_{in-1}),\ b_j^T = (b_{0j}, b_{1j}, \ldots, b_{n-1j})^T.$$

Этот алгоритм предполагает выполнение *m*n*l* операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера *n*n* количество выполненных операций имеет порядок $O(n^3)$. Предполагается, что все матрицы являются квадратными и имеют размер *n*n*.

.

1.2. Последовательный алгоритм

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы *C*. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной *i*) вычисляется одна строка результирующей матрицы (см. рис. В.1).

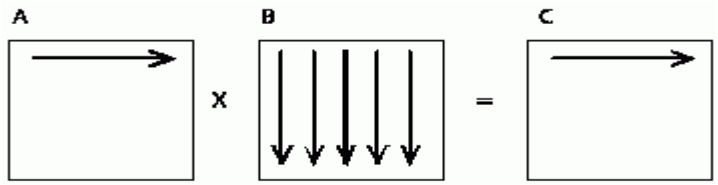


Рисунок В.1 – Схема получения результата на первой итерации цикла по переменной *i* (используется первая строка матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*)

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы *C* размером *n*n* необходимо выполнить $n^2*(2n-1)$ скалярных операций и затратить время: $T = n^2 * (2n-1) * \tau$, где τ – время выполнения одной элементарной скалярной операции.

1.3. Умножение матриц при ленточной схеме разделения данных

Рассмотрим два параллельных алгоритма умножения матриц, в которых матрицы *A* и *B* разбиваются на непрерывные последовательности строк или столбцов (полосы).

1.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы *C* может быть выполнено независимо друг от друга. Организация параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы *C*. Для проведения всех необходимых вычислений каждая подзадача должна содержать по одной строке матрицы *A* и одному столбцу матрицы *B*. Общее количество получаемых при таком подходе подзадач равно n^2 (по числу элементов матрицы *C*). Достигаемый при этом уровень параллелизма является избыточным т.к. при проведении практических расчетов такое количество сформированных подзадач превышает число имеющихся процессоров и делает неизбежным этап укрупнения базовых задач. В этом этом случае выполняется агрегация вычислений на шаге выделения базовых подзадач. Возможное решение связано с объединением в рамках одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы *C*. Определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы C. Такой подход приводит к снижению общего количества подзадач до величины *n*.

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы *A* и все столбцы матрицы *B*. Простое решение – дублирование матрицы *B* во всех подзадачах – является неприемлемым в силу больших затрат памяти для хранения данных. Поэтому организация вычислений должна быть построена таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть данных, необходимых для проведения расчетов, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Два возможных способа выполнения параллельных вычислений подобного типа рассмотрены в пункте 1.3.2.

1.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы *C* необходимо, чтобы в каждой подзадаче содержалась строка матрицы *A* и был обеспечен доступ ко всем столбцам матрицы *B*. Возможные способы организации параллельных вычислений представлены следующими алгоритмами.

1. **Алгоритм 1.** Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач. На каждой итерации алгоритма каждая подзадача содержит по одной строке матрицы *A* и одному столбцу матрицы *B*. При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк и столбцов, что приводит к получению соответствующих элементов результирующей матрицы *C*. По завершении вычислений в конце каждой итерации столбцы матрицы *B* должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы *B*, могли быть вычислены новые элементы матрицы *C*. При этом данная передача столбцов между подзадачами должна быть организована таким образом, чтобы после завершения итераций алгоритма в каждой подзадаче последовательно оказались все столбцы матрицы *B*. Простая схема организации последовательности передач столбцов матрицы *B* между подзадачами состоит в представлении топологии информационных связей подзадач в виде кольцевой структуры. В этом случае на каждой итерации подзадача *i*, *0<=i<n*, будет передавать свой столбец матрицы *B* подзадаче с номером *i+1* (в соответствии с кольцевой структурой подзадача *n-1* передает свои данные подзадаче с номером 0 – Рис. В.2). После выполнения всех итераций алгоритма необходимое условие будет обеспечено – в каждой подзадаче поочередно окажутся все столбцы матрицы *B*.

На Рис. В.2 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов (*n=4*). В начале вычислений в каждой подзадаче *i* (0<=i<n), располагаются *i* -я строка матрицы *A* и *i* -й столбец матрицы *B*. В результате их перемножения подзадача получает элемент c_{ii} результирующей матрицы *C*. Далее подзадачи осуществляют обмен столбцами, в ходе которого каждая подзадача передает свой столбец матрицы *B* следующей подзадаче в соответствии с кольцевой структурой информационных взаимодействий. Выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

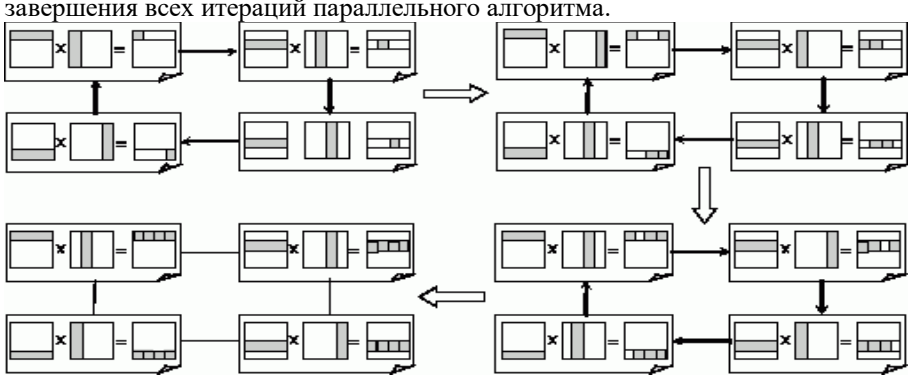


Рисунок В.2– Схема передачи данных для первого параллельного алгоритма матричного умножения (ленточная схема разделения данных)

2. **Алгоритм 2.** Алгоритм предполагает, что в подзадачах располагаются не столбцы, а строки матрицы *B*. Как результат, перемножение данных каждой подзадачи сводится не к скалярному умножению имеющихся векторов, а к их поэлементному умножению. В результате подобного умножения в каждой подзадаче получается строка частичных результатов для матрицы *C*. При рассмотренном способе разделения данных для выполнения операции матричного умножения нужно обеспечить последовательное получение в подзадачах всех строк матрицы *B*, поэлементное умножение данных (элементов строк матриц *A* и *B)* и суммирование вновь получаемых значений с ранее вычисленными результатами. Организация необходимой последовательности передач строк матрицы *B* между подзадачами также может быть выполнена с использованием кольцевой структуры информационных связей (рис. В.3).

На рис. В.3 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов (*n=4*). В начале вычислений в каждой подзадаче *i* (*0<=i<n*), располагаются *i*-е строки матриц *A* и *B*. В результате их перемножения подзадача определяет *i* -ю строку частичных результатов искомой матрицы *C*. Далее подзадачи осуществляют обмен строками, в ходе которого каждая подзадача передает свою строку матрицы *B* следующей подзадаче в соответствии с кольцевой структурой информационных связей (вновь определяется *i* -я строка частичных результатов искомой матрицы *C*, элементы которой складываются с частичными результатами с предыдущей итерации) . Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

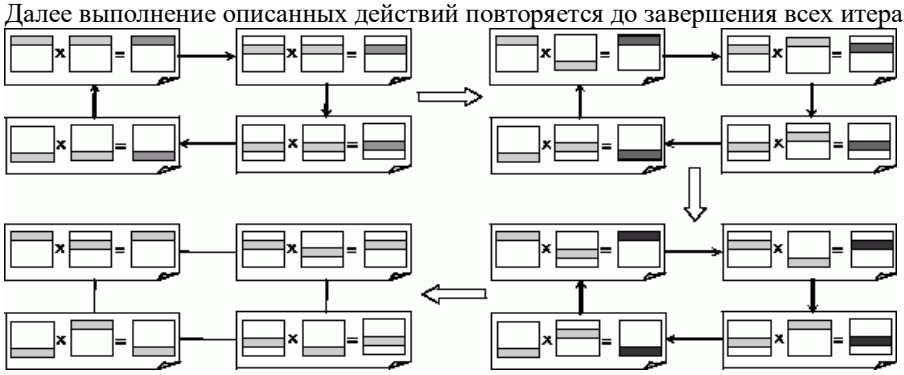


Рисунок В.3– Схема передачи данных для второго параллельного алгоритма матричного умножения (ленточная схема разделения данных)

1.4. Алгоритмы умножения матриц при блочном разделении данных

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Рассмотрим более подробно данный способ организации вычислений. При таком способе разделения данных исходные матрицы *A*, *B* и результирующая матрица *C* представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера *n*n*, количество блоков по горизонтали и вертикали одинаково и равно *q* (размер каждого блока

равен *k*k*, *k=n/q*). При таком представлении данных операция матричного умножения матриц *A* и *B* в блочном виде может быть представлена так:

$$\begin{pmatrix} A_{00} & \square_{01} & \cdots & \square_{0\,\square-1} \\ \cdots & \cdots & \cdots & \cdots \\ A_{q-1\,0} & A_{q-1\,1} & \cdots & \square_{\square-1\,\square-1} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} & \cdots & \square_{0\,\square-1} \\ \cdots & \cdots & \cdots & \cdots \\ \square_{\square-1\,0} & B_{q-1\,1} & \cdots & \square_{\square-1\,\square-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \cdots & \square_{0\,\square-1} \\ \cdots & \cdots & \cdots & \cdots \\ \square_{\square-1\,0} & \square_{\square-1\,1} & \cdots & \square_{\square-1\,\square-1} \end{pmatrix},$$

каждый блок Cij матрицы C определяется в соответствии с выражением вида $C_{ij} = \sum_{s=0}^{q-1} A_{is} \cdot B_{sj}$, где A_{is} и B_{sj} – соответствующие блоки матриц *A* и *B*, которые перемножаются между собой, а затем получены результаты складываются для получения блока C_{ij} матрицы *C*. При блочном разбиении данных базовым подзадачам требуется реализовывать вычисления, выполняемые над матричными блоками. С учетом сказанного базовая подзадача– это процедура вычисления всех элементов одного из блоков матрицы *C*.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы *A* и столбцов матрицы *B*. Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи данных между процессорами. Возможным подходом к решению задачи распределения данных (обмена данными) является алгоритм Фокса.

1.4.1. Алгоритм Фокса. Выделение информационных зависимостей

За основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы *C* и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц *A* и *B*. Для нумерации подзадач использованы индексы размещаемых в подзадачах блоков матрицы *C*, т.е. подзадача *(i,j)* реализует вычисление блока C_{ij} . Тогда набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче *(i,j)* располагается четыре матричных блока:

- блок C_{ij} матрицы *C*, вычисляемый подзадачей;
- блок A_{ij} матрицы *A*, размещаемый в подзадаче перед началом вычислений;
- блоки A'_{ij} , B'_{ij} матриц *A* и *B*, получаемые подзадачей в ходе выполнения вычислений.

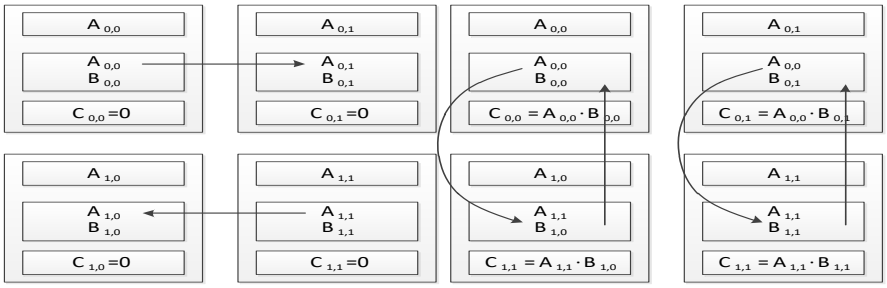
Выполнение параллельного метода включает:

1) этап инициализации, на котором каждой подзадаче *(i,j)* передаются блоки A_{ij} , B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;

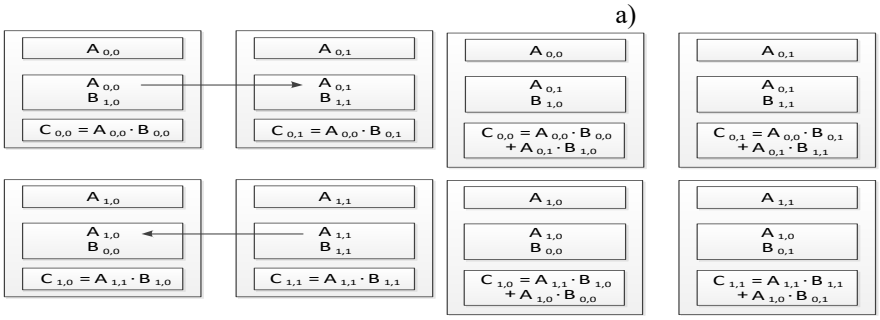
2) этап вычислений, в рамках которого на каждой итерации *l* (*0<=l<q*, т.е. всего *q* итераций), осуществляются следующие операции:

- для каждой строки *i* (*0<=i<q*) блок A_{ij} подзадачи *(i,j)* пересылается подзадаче этой же строки *i* решетки; индекс *j*, определяющий положение подзадачи в строке (которой пересылается блок A_{ij}), вычисляется в соответствии с выражением $j = (i + l) \bmod q$, где *mod* есть операция получения остатка от целочисленного деления;
- полученные в результаты пересылок блоки A'_{ij} , B'_{ij} каждой подзадачи *(i,j)* перемножаются и прибавляются к блоку C_{ij}.
- блоки B'_{ij} каждой подзадачи *(i,j)* пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Напомним, что число итераций алгоритма поблочного матричного перемножения равно количеству блоков *q*, на которое разбиваются исходные матрицы *A* и *B*. На Рис.В.4 представлена схема обмена блоками матриц *A* и *B* между подзадачами при *q=2*.



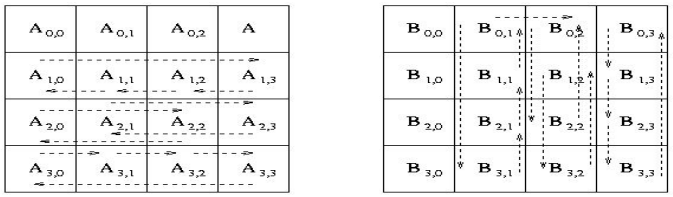
а)



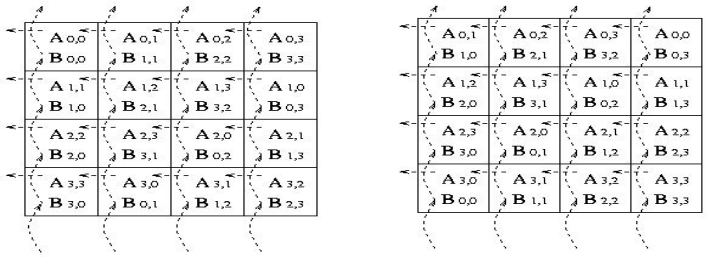
б)

Рисунок В.4– Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса. а) первая итерация алгоритма; б) вторая итерация алгоритма

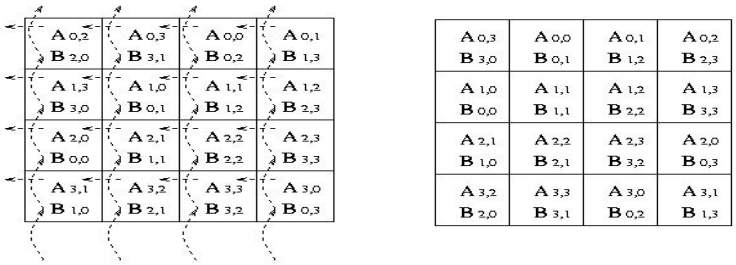
На Рис.В.5 представлена схема обмена блоками матриц *A* и *B* между подзадачами при *q=4.*



а)



б) в)



г) д)

Рисунок В.5– Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса при *q=4.* а) первоначальный обмен блоками матриц; б) умножение блоков и обмен (1 шаг); в) умножение блоков и обмен (2 шаг); г) умножение блоков и обмен (3 шаг);д) умножение блоков (4 шаг);

1.4.2. Программная реализация алгоритма Фокса

Представим возможный вариант *программной реализации* алгоритма Фокса для *умножения матриц* при блочном представлении данных. Приводимый программный код содержит основные модули параллельной программы, отсутствие отдельных вспомогательных функций не сказывается на общем понимании реализуемой схемы параллельных вычислений.

Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Условия выполнения программы: все матрицы квадратные,
// размер блоков и их количество по горизонтали и вертикали
// одинаково, процессы образуют квадратную решетку
int ProcNum = 0;     // Количество доступных процессов
int ProcRank = 0;    // Ранг текущего процесса
int GridSize;        // Размер виртуальной решетки процессов
int GridCoords[2];   // Координаты текущего процесса в процессной
                     // решетке
MPI_Comm GridComm;   // Коммуникатор в виде квадратной решетки
MPI_Comm ColComm;    // коммуникатор – столбец решетки
MPI_Comm RowComm;    // коммуникатор – строка решетки
void main ( int argc, char * argv[] )
{
    double* pAMatrix;   // Первый аргумент матричного умножения
    double* pBMatrix;   // Второй аргумент матричного умножения
    double* pCMatrix;   // Результирующая матрица
    int Size;           // Размер матриц
    int BlockSize;      // Размер матричных блоков, расположенных
                        // на процессах
    double *pAblock;    // Блок матрицы A на процессе
    double *pBblock;    // Блок матрицы B на процессе
    double *pCblock;    // Блок результирующей матрицы C на процессе
    double *pMatrixAblock;
    double Start, Finish, Duration;
    setvbuf(stdout, 0, _IONBF, 0);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    GridSize = sqrt((double)ProcNum);
    if (ProcNum != GridSize*GridSize)
    {
        if (ProcRank == 0)
        {
            printf ("Number of processes must be a perfect square \n");
        }
    }
    else
    {
        if (ProcRank == 0)
            printf("Parallel matrix multiplication program\n");

        // Создание виртуальной решетки процессов и коммуникаторов строк и столбцов
        CreateGridCommunicators();
        // Выделение памяти и инициализация элементов матриц
       ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock,pBblock, pCblock
, pMatrixAblock, Size, BlockSize );
        // Блочное распределение матриц между процессами
        DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
            BlockSize);
        // Выполнение параллельного метода Фокса
        ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
            pCblock, BlockSize);
        // Сбор результирующей матрицы на ведущем процессе
       ResultCollection(pCMatrix, pCblock, Size, BlockSize);
```

```
        // Завершение процесса вычислений
        ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock, pCblock,
pMatrixAblock);
    }
    MPI_Finalize();
}
```