4 Групповые обмены

4.1 Введение и обзор

Групповыми обменами называются обмены, в которых участвует группа процессов. MPI обеспечивает следующие функции этого типа.

- Барьерная синхронизация всех членов группы (Часть 4.3).
- Распространение данных от одного, всем членам группы (Часть 4.4). Как показано на рисунке 4.1.
- Сбор данных от всех членов группы на один (Часть 4.5). Как показано на рисунке 4.1.
- Рассылка данных от одного, всем членам группы (Часть 4.6). Как показано на рисунке 4.1.
- Вариант операции Рассылки, при котором все члены группы получают результат (Часть 4.7). Это показано как "собирают все" на рисунке 4.1.
- Сбор/Рассылка данных от всех элементов - всем членам группы (также называют полным обменом или все-всем) (Часть 4.8). Это показано как "все-всем" на рисунке 4.1.
- Групповые операции, такие как sum, max, min, и так далее, которые возвращают результат всем членам группы и вариант, в котором результат возвращается только одному (Часть 4.9). Другой вариант позволяет определённые пользователем групповые операции.
- Комбинация групповой операции и операции рассылки (Часть 4.10). Варианты позволяющие определённые пользователем операции.
- Сканирование по всем членам группы (также называемое префиксной операцией).(Часть 4.11). Варианты позволяющие определённые пользователем операции сканирования.

Групповые операции выполняются при помощи вызова соответствующей подпрограммы с подходящими аргументами всеми процессами группы. Синтаксис и семантика групповых операций определяется с сохранением преемственности синтаксиса обменов точка с точкой. Таким образом допускаются общие типы данных и должно соблюдаться соответствие типов между передающим и принимающим процессами, как описано в Главе 3. Одним из ключевых аргументов является коммуникатор, который определяет группу участвующих процессов и обеспечивает контекст для операции. Некоторые групповые операции, такие как распространение или сбор имеют один передающи й или принимающий процесс. Такой процесс называется корень. Некоторые аргументы групповых функций помечены "значимо только для корня", и игнорируются всеми процессами за исключением корня. Для более подробной информации о буферах, общих типах данных и правилах соответствия типов данных читайте главу 3, и главу 5, чтобы узнать, как определить группу и создать коммуникатор.

Условия соответствия типов для групповых операций строже, чем для посылающего и принимающего при двухточечных обменах. А именно, при групповых операциях количество передаваемых данных должно точно соответствовать количеству заявленному принимающим процессом. Различие карт типов (расположение в памяти Часть 3.12) между передающим и принимающим допустимо.

Подпрограммы групповых операций могут (но не обязаны) вернуть управление, как только участие данного процесса в операции закончено. Возврат управления показывает только, что вызвавший процесс может свободно использовать содержимое буфера. Но не говорит о том, что все остальные процессы группы завершили, или

даже начали эту операцию (если конечно это не указно в определении операции). Таким образом, групповые операции могут иметь, а могут и не иметь эффект синхронизации вызвавших процессов. Конечно за исключением барьерной функции.

Групповые операции могут использовать те же коммуникаторы, что и обмены точка с точкой; MPI гарантирует, что сообщения порождённые групповыми операциями не смешиваются с сообщениями порождаемыми обменами точка с точкой. Более подробно о корректном использовании групповых операций читайте Часть 4.12.

Целесообразность. Ограничение равенства данных (совпадение типов) было подобрано так, чтобы избежать сложности обеспечения возможностей аналогичных аргументу статуса в MPI_RECV для получения количества передаваемых данных. Некоторые из групповых операций могли бы потребовать целый массив статусов.

Замечание о синхронизации сделано так, чтобы допустить различные варианты реализации групповых операций.

Групповые операции не используют аргумент тэг. Если в дальнейшем в MPI будут определены не-блокирующие групповые операции, тогда тэги (или подобный механизм) будет необходимо добавить, чтобы обеспечить однозначность для множества ожидающих групповых операций. (Конец замечания.)

Совет пользователям. Опасно использовать синхронизующий эффект для обеспечения корректной работы программ. Например, даже если в конкретной реализации групповой операции есть эффект синхронизации, то программа использующая этот эффект не будет переносимой, так как стандарт не требует этого.

С другой стороны, корректная, совместимая программа должна допускать существование синхронизующего эффекта. Не опираясь на возможный синхронизующий эффект, программа должна допускать его существование. Эти особенности далее обсуждаются в Части 4.12. (Конец совета пользователям.)

Совет разработчикам. Хотя групповые операции можно написать оптимизированные групповые операции использующие особенности архитектуры, библиотеку групповых операций можно целиком реализовать, используя MPI обмены точка с точкой и несколько специальных функций. При реализации на основе обменов точка с точкой, необходимо создать специальные скрытые коммуникаторы для групповых обменов, чтобы избежать путаницы между работающими одновременно групповыми операциями и обменами точка с точкой. Подробнее это рассмотрено в Части 4.12. (Конец совета разработчикам.)

4.2 Аргумент коммуникатор

Ключевым понятием групповых функций является "группа" участвующих процессов. Подпрограммы не имеют конкретного аргумента являющегося идентификатором группы. Однако у них есть коммуникатор. В этой части можно считать коммуникатор идентификатором группы связанным с контекстом. Внешние коммуникаторы, то есть коммуникаторы объединяющие две группы, недопустимы в групповых функциях.

4.3 Барьерная синхронизация

```
MPI_BARRIER(comm)
IN     comm          коммуникатор (handle)

int MPI_Barrier(MPI_Comm comm)

MPI_BARRIER (COMM, IERROR)
```

```
       INTEGER COMM, IERROR
```

MPI_BARRIER блокирует вызвавший процесс до тех пор пока все члены группы не вызовут его. Управление возвращается в любой процесс, только когда все члены группы сделают вызов.

4.4 Распространение

```
MPI_BCAST (buffer, count, datatype, root, comm)
INOUT         buffer               адрес начала буфера (choice)
IN            count                количество элементов в буфере (intager)
IN            datatype             тип данных в буфере (handle)
IN            root                 ранк распространяющего корня (integer)
IN            comm                 коммуникатор (handle)

int MPI_Bcast(ovid *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
```

MPI_BCAST распространяет сообщение от процесса с ранком root на все процессы группы, включая себя. Он вызывается всеми членами группы, используя одинаковые значения comm, root. По завершению, содержимое буфера корневого узла будет скопировано в буфера всех процессов.

Общие и выведенные типы данных допускаются для datatype. Сигнатура типа count, datatype на любом процессе должна совпадать с сигнатурой типа count, datatype корня. Это подразумевает, что количество передаваемых данных должно быть равно количеству получаемых, попарно между каждым процессом и корнем. MPI_BCAST и все остальные подпрограммы групповой передачи данных следуют этому ограничению. Различие карты типов между посылающим и принимающим ещё допустимо.

4.4.1 Пример использования MPI_BCAST

Пример 4.1 Распространение 100 целых с процесса 0 на все процессы в группе.

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

Как и в большинстве наших примеров, мы допускаем, что некоторым переменным, например в данном случае comm, были присвоены соответствующие значения.

4.5 Сбор

```
MPI_GATHER (sendbuf, sendcount, sendtype, recvtbuf, recvcount,
recvtype, root, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
IN     sendcount            количество элементов в буфере посылки
                            (integer)
IN     sendtype             тип данных в буфере посылки (handle)
OUT    recvbuf              адрес получаемого буфера (choice, значимо
                            только для корня)
IN     recvcount            количество элементов для каждого приёма
                            (integer, значимо только для корня)
IN     recvtype             тип данных в принимающем буфере (handle,
                            значимо только для корня)
IN     root                 ранк принимающего процесса (integer)
IN     comm                 коммуникатор (handle)
```

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

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, ROOT, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM,
       IERROR
```

Каждый процесс (включая корневой) посылает содержимое своего посылаемого буфера в корневой процесс. Корневой процесс получает сообщения и сохраняет их в порядке возрастания ранков. Результат получается как если бы каждый из n процессов в группе выполнил вызов

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

а корневой выполнил n вызовов

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

где extent(recvtype) это протяжённость типа, получаемая с помощью вызова MPI_Type_extent().

Или по другому, n сообщений, посылаемых процессами в группе связываются в порядке возрастания ранков, и получившееся сообщение принимается корнем, как бы вызовом MPI_RECV(recvbuf, recvcount¦n, recvtype, ...).

Приёмный буфер игнорируется всеми некорневыми процессами.

Общие и выведенные типы данных допускаются для sendtype recvtype. Сигнатура типа sendcount, sendtype на процессе i должна совпадать с сигнатурой типа recvcount, recvtype корня. Это подразумевает, что количество передаваемых данных должно быть равно количеству получаемых, попарно между каждым процессом и корнем. Различие карты типов между посылающим и принимающим ещё допустимо.

Все аргументы функции значимы для процесса root, тогда как для других процессов значимы только sendbuf, sendcount, sendtype, root, comm. Аргументы root и comm должны иметь одинаковые значения на всех процессах.

Значения recvcount и recvtype не должны допускать более одной записи в одно место на корневом процессе. Иначе вызов ошибочен.

Заметим, что аргумент recvcount в корневом процессе отражает количество информации, которое он получит от каждого процесса, а не общее количество данных.

```
MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
displs, recvtype, root, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
IN     sendcount            количество элементов в буфере посылки
                            (integer)
IN     sendtype             тип данных в буфере посылки (handle)
OUT    recvbuf              адрес получаемого буфера (choice, значимо
                            только для корня)
IN     recvcounts           массив целых (с длиной равной размеру группы),
                            содержащий количество элементов принимаемых от
                            каждого процесса (значимо только для корня)
IN     displs               массив целых (с длиной равной размеру группы),
                            на i-ом месте которого стоит смещение
                            относительно recvbuf, куда поместить данные
                            получаемые от процесса i (значимо только для
                            корня)
IN     recvtype             тип данных в принимающем буфере (handle,
                            значимо только для корня)
IN     root                 ранк принимающего процесса (integer)
IN     comm                 коммуникатор (handle)
```

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

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
              DISPLS, RECVTYPE, ROOT, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
       RECVTYPE, ROOT, COMM, IERROR
```

MPI_GATHERV функционально расширяет MPI_GATHER, допуская переменное количество данных для каждого процесса, так как recvcounts теперь массив. Он также позволяет большую гибкость при размещении данных в корневом процессе с помощью аргумента displs.

Результат получается как если бы каждый из n процессов в группе выполнил вызов

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

а корневой выполнит n вызовов

```
MPI_Recv(recvbuf + displ[i]¦extent(recvtype), recvcounts[i],
       recvtype, i, ...).
```

Сообщения помещаются в принимающий буфер корневого процесса в порядке возрастания ранков, то есть данные посланные процессом j помещаются в j-ую порцию приёмного буфера процесса .J-ая порция приёмного буфера recvbuf начинается со смещения в displs[j] элементов (с размером extent(recvtype)) в recvtype.

Приёмный буфер игнорируется всеми некорневыми процессами.

Сигнатура типа sendcount, sendtype на процессе i должна совпадать с сигнатурой типа recvcounts[i], recvtype корня. Это подразумевает, что количество передаваемых данных должно быть равно количеству получаемых, попарно между каждым процессом и корнем. Различие карты типов между посылающим и принимающим ещё допустимо, как показано в Примере 4.6.

Все аргументы функции значимы для процесса root, тогда как для других процессов значимы только sendbuf, sendcount, sendtype, root, comm. Аргументы root и comm должны иметь одинаковые значения на всех процессах.

Значения количеств и типов не должны допускать более одной записи в одно место на корневом процессе. Такой вызов ошибочен.

4.5.1 Примеры использующие MPI_GATHER, MPI_GATHERV

Пример 4.2 Сбор в корневой процесс 100 единиц с каждого процесса группы (рисунок 4.2).

```
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);
```

Пример 4.3 Предыдущий пример модифицирован - только корневой процесс выделяет память для приёмного буфера.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank(comm, myrank);
if(myrank == root){
```

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

Пример 4.4 Делается тоже, что в предыдущем примере, но используется выведенный тип данных. Заметим, что тип не может составлять всё множество gsize*100 intов, так как сравнение типов производится попарно между корнем и каждым процессом группы.

```
        MPI_Comm comm;
        int gsize, sendarray[100];
        int root, *rbuf;
        MPI_Datatype rtype;
        ...
        MPI_Comm_size(comm, &gsize);
        MPI_Type_contiguous(100, MPI_INT, &rtype);
        MPI_Type_commit(&rtype);
        rbuf = (int *)malloc(gsize*100*sizeof(int));
        MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root,
            comm);
```

Пример 4.5 Теперь пусть каждый процесс посылает 100 целых корневому, но помещает каждое множество (из 100 элементов) со смещением stride целых от предыдущего. Используем MPI_GATHERV и аргумент displs, чтобы достичь этого. Допустим stride ¦ 100 (рисунок 4.3).

```
        MPI_Comm comm;
        int gsize, sendarray[100];
        int root, *rbuf, stride;
        int *displs, i, *rcounts;
        ...
        MPI_Comm_size(comm, &gsize);
        rbuf = (int *)malloc(gsize*stride*sizeof(int));
        displs = (int *)malloc(gsize*sizeof(int));
        rcounts = (int *)malloc(gsize*sizeof(int));
        for(i = 0; i < gsize; ++i){
            displs[i] = i*stride;
            rcountsp[i] = 100;
        }
        MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs,
            MPI_INT, root, comm);
```

Заметим, что программа не верна при stride < 100.

Пример 4.6 Для принимающей стороны также как в примере 4.5, но посылается по 100 целых из нулевого столбца массива 100*150 целых, на Си (рисунок 4.4).

```
        MPI_Comm comm;
        int gsize, sendarray[100][150];
        int root, *rbuf, stride;
        MPI_Datatype stype;
        int *displs, i, *rcounts;
        ...
        MPI_Comm_size(comm, &gsize);
        rbuf = (int *)malloc(gsize*stride*sizeof(int));
        displs = (int *)malloc(gsize*sizeof(int));
        rcounts = (int *)malloc(gsize*sizeof(int));
        for(i = 0; i < gsize; ++i){
            displs[i] = i*stride;
            rcountsp[i] = 100;
        }
        /* Создание типа данных для 1 столбца массива */
        MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
```

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

Пример 4.7 Процесс i посылает (100-i) целых из i-ого столбца массива 100*150 целых Си. Данные принимаются в буфер со смещением, как и в предыдущих двух примерах (рисунок 4.5).

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i){
       displs[i] = i*stride;
       rcountsp[i] = 100 - i;            /* отметим отличие от
                                         предыдущего примера*/
}
/* Создание типа данных для столбца, который мы посылаем */
MPI_Type_vector(100 - myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr - адрес начала "myrank" столбца*/
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
       root, comm);
```

Отметим, что от каждого процесса принимается разное количество данных.

Пример 4.8 Такой же как Пример 4.7, но посылает другим способом. Мы создаём тип данных, который обеспечивает корректное смещение на посылающей стороне, так чтобы мы читали столбец Си массива. Так же сделано во Втором Примере раздела 3.12.7.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype, type[2];
int *displs, i, *rcounts;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i){
       displs[i] = i*stride;
       rcountsp[i] = 100 - i;
}
/* Создание типа данных для одного целого, с размером в строку
*/
disp[0] = 0;                  disp[1] = 150*sizeof(int);
type[0] = MPI_INT;            type[1] = MPI_UB;
blocklen[0] = 1;              blocklen[1] = 1;
MPI_Type_struct(2, blocklen, disp, type, &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 100 - myrank, stype, rbuf, rcounts, displs,
       MPI_INT, root, comm);
```

Пример 4.9 Для посылающей стороны аналогично Примеру 4.7, но принимающая сторона изменяет смещение между принимаемыми блоками (рисунок 4.6).

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs, i, *rcounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] устанавливается для i от 0 до gsize - 1 */
/* сначала выделяем место для векторов displs и rcounts */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for(i = 0; i < gsize; ++i){
       displs[i] = offset;
       offset += stride[i];
       rcountsp[i] = 100 - i;
}
/* Теперь легко получить размер буфера необходимого для rbuf
*/
bufsize = displs[gsize-1] + rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Создание типа данных для столбца, который мы посылаем */
MPI_Type_vector(100 - myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
       root, comm);
```

Пример 4.10 Процесс i посылает num целых из i-ого столбца массива целых в Си. Усложняющий фактор состоит в том, что различные значения num не известны root, так чтобы получить их необходим отдельный сбор. Данные размещаются непрерывно на принимающем узле.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype, type[2];
int *displs, i, *rcounts, num;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
/* сначала соберём значения num в корень*/
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gatherv(&num, 1, MPI_INT, rcounts, 1, MPI_INT, root,
       comm);
/* теперь корень имеет верные значения в rcounts, используя их
  мы установим displs[], так чтобы данные на принимающем узле
  располагались непрерывно (или связанно)*/
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for(i = 1; i < gsize; ++i){
       displs[i] = displs[i-1] + rcountsp[i-1];
}
/* Теперь создадим приёмный буфер */
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
       * sizeof(int));
/* Создание типа данных для одного целого, с размером в строку
*/
disp[0] = 0;                disp[1] = 150*sizeof(int);
```

```
        type[0] = MPI_INT;          type[1] = MPI_UB;
        blocklen[0] = 1;            blocklen[1] = 1;
        MPI_Type_struct(2, blocklen, disp, type, &stype);
        MPI_Type_commit(&stype);
        sptr = &sendarray[0][myrank];
        MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
               root, comm);
```

4.6 Рассылка

```
MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount,
        recvtype, root, comm)
IN      sendbuf             адрес начала посылаемого буфера (choice,
                            значимо только для корня)
IN      sendcount           количество элементов в буфере посылки
                            (integer, значимо только для корня)
IN      sendtype            тип данных в буфере посылки (handle,
                            значимо только для корня)
OUT     recvbuf             адрес получаемого буфера (choice)
IN      recvcount           количество элементов для каждого приёма
                            (integer)
IN      recvtype            тип данных в принимающем буфере (handle)
IN      root                ранк принимающего процесса (integer)
IN      comm                коммуникатор (handle)

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

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
               RECVTYPE, ROOT, COMM, IERROR)
        <type> SENDBUF(*), RECVBUF(*)
        INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM,
        IERROR
```

MPI_SCATTER операция обратная MPI_GATHER. Результат получается как если бы корень выполнил n операций посылки

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

а каждый процесс выполнил приём

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

По другому можно сказать, что корень посылает сообщение с помощью MPI_Send(sendbuf, sendcount¦n, sendtype, ...). Это сообщение разбивается на n равных сегментов, i-ый сегмент посылается i-ому процессу в группе, и каждый процесс получает сообщение так же как выше.

Посылаемый буфер игнорируется всеми некорневыми процессами.

Сигнатура типа ассоциированная с sendcount, sendtype на корневом процессе должна совпадать с сигнатурой типа ассоциированной с recvcount, recvtype на всех процессах (однако, карты типов могут быть разными). Это подразумевает, что количество передаваемых данных должно быть равно количеству получаемых, попарно между каждым процессом и корнем. Различие карты типов между посылающим и принимающим ещё допустимо.

Все аргументы функции значимы для процесса root, тогда как для других процессов значимы только recvbuf, recvcount, recvtype, root, comm. Аргументы root и comm должны иметь одинаковые значения на всех процессах.

Значения количеств и типов не должны допускать более одного чтения из одного места на корневом процессе.

Целесообразность. Хотя в последнем ограничении нет необходимости, оно налагается, чтобы достичь симметрии с

MPI_GATHER, где соответствующее ограничение (ограничение на многократную запись) необходимо.(Конец замечания.)

```
MPI_SCATTERV(sendbuf, sendcount, displs, sendtype, recvbuf,
        recvcounts, recvtype, root, comm)
IN      sendbuf             адрес начала посылаемого буфера (choice,
                            значимо только для корня)
IN      sendcounts          массив целых (с длиной равной размеру группы),
                            содержащий количество элементов посылаемых
                            каждым процессом (значимо только для корня)
IN      displs              массив целых (с длиной равной размеру группы),
                            на i-ом месте которого стоит смещение
                            относительно sendvbuf, откуда берутся данные
                            передаваемые процессу i (значимо только для
                            корня)
IN      sendtype            тип данных в буфере посылки (handle, значимо
                            только для корня)
OUT     recvbuf             адрес получаемого буфера (choice)
IN      recvcount           количество элементов в буфере посылки
                            (integer)
IN      recvtype            тип данных в принимающем буфере (handle)
IN      root                ранк принимающего процесса (integer)
IN      comm                коммуникатор (handle)

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

MPI_GATHER(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
              RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
        <type> SENDBUF(*), RECVBUF(*)
        INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
        RECVTYPE, ROOT, COMM, IERROR
```

MPI_SCATTERV операция обратная MPI_GATHERV.

MPI_SCATTERV функционально расширяет MPI_SCATTER, допуская передачу переменного количества данных для каждого процесса, так как sendcounts теперь массив. Она также позволяет большую гибкость при выборке данных в корневом процессе с помощью аргумента displs.

Результат получается как если бы корень выполнил n операций передачи

```
MPI_Send(sendbuf + displ[i]¦extent(sendtype), sendcounts[i],
        sendtype, i, ...),
```

а каждый процесс выполнил приём

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

Посылаемый буфер игнорируется всеми некорневыми процессами.

Сигнатура типа подразумеваемая sendcounts[i], sendtype на корне должна совпадать с сигнатурой типа recvcount, recvtype процесса i (однако, карты типов могут быть разными). Это подразумевает, что количество передаваемых данных должно быть равно количеству получаемых, попарно между каждым процессом и корнем. Различие карты типов между посылающим и принимающим ещё допустимо.

Все аргументы функции значимы для процесса root, тогда как для других процессов значимы только sendbuf, sendcount, sendtype, root, comm. Аргументы root и comm должны иметь одинаковые значения на всех процессах.

Значения количеств, типов и смещений не должны допускать более одного чтения из одного места на корневом процессе.

4.6.1 Примеры использования MPI_SCATTER, MPI_SCATTERV

Пример 4.11 Обратный Примеру 4.2. Рассылается множество из 100

целых из корневого процесса всем процессам группы (рисунок 4.7).

```
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);
```

Пример 4.12 Обратный Примеру 4.5. Корневой процесс рассылает множество из 100 целых всем процессам, но множества по 100 целых размещены с шагом stride целых по посылаемому буферу. Требуется использование MPI_SCATTERV. Допустим stride ¦ 100 (рисунок 4.8).

```
MPI_Comm comm;
int gsize, *sendbuf, stride;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for(i = 0; i < gsize; ++i){
       displs[i] = i*stride;
       rcountsp[i] = 100;
}
MPI_Scatterv(sendbuf, displs, scounts, MPI_INT, rbuf, 100,
       MPI_INT, root, comm);
```

Пример 4.13 Обратный Примеру 4.9. Мы изменяем смещение между блоками посылающей стороны, на принимающем мы получаем в i-ый столбец Си массива 100*150 (рисунок 4.9).

```
MPI_Comm comm;
int gsize, secvarray[100][150], *rptr;
int root, *sendbuf, *stride, myrank, bufsize;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] устанавливается для i от 0 до gsize - 1
* sendbuf где-то заполняется
*/
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for(i = 0; i < gsize; ++i){
       displs[i] = offset;
       offset += stride[i];
       rcountsp[i] = 100 - i;
}
/* Создание типа данных для столбца, который мы принимаем */
MPI_Type_vector(100 - myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1,
       rtype, root, comm);
```

4.7 Сбор для всех

```
MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvtbuf, recvcount,
              recvtype, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
IN     sendcount            количество элементов в буфере посылки
                            (integer)
IN     sendtype             тип данных в буфере посылки (handle)
OUT    recvbuf              адрес получаемого буфера (choice)
IN     recvcount            количество элементов получаемых от каждого
                            процесса (integer)
IN     recvtype             тип данных в принимающем буфере (handle)
IN     comm                 коммуникатор (handle)

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

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
              RECVTYPE, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLGATHER можно рассматривать как MPI_GATHER, где все процессы получают результат, а не только корень. J-ый блок данных посылается каждым процессом и принимается каждым процессом в j-ый блок буфера recvbuf.

Сигнатура типа ассоциированная с sendcount, sendtype на процессе должна совпадать с сигнатурой типа ассоциированной с recvcount, recvtype любого другого процесса.

Результат получается как если бы каждый процесс выполнил n вызов

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

для root = 0, ..., n-1. Правила корректного использования MPI_ALLGATHER легко получить из соответствующих правил MPI_GATHER.

```
MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
              displs, recvtype, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
IN     sendcount            количество элементов в буфере посылки
                            (integer)
IN     sendtype             тип данных в буфере посылки (handle)
OUT    recvbuf              адрес получаемого буфера (choice)
IN     recvcounts           массив целых (с длиной равной размеру группы),
                            содержащий количество элементов принимаемых от
                            каждого процесса
IN     displs               массив целых (с длиной равной размеру группы),
                            на i-ом месте которого стоит смещение
                            относительно recvbuf, куда поместить данные
                            получаемые от процесса i
IN     recvtype             тип данных в принимающем буфере (handle)
IN     comm                 коммуникатор (handle)

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

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
              DISPLS, RECVTYPE, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
       RECVTYPE, COMM, IERROR
```

MPI_ALLGATHERV можно рассматривать как MPI_GATHERV, где все процессы получают результат, а не только корень. J-ый блок данных посылается каждым процессом и принимается каждым процессом в j-ый блок буфера recvbuf. При разных j величина блоков может быть различной.

Сигнатура типа ассоциированная с sendcount, sendtype на процессе j должна совпадать с сигнатурой типа ассоциированной с recvcount[j], recvtype любого другого процесса.

Результат получается как если бы каждый процесс выполнил n вызов

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

для root = 0, ..., n-1. Правила корректного использования MPI_ALLGATHERV легко получить из соответствующий правил MPI_GATHERV.

4.7.1 Пример использования MPI_ALLGATHER, MPI_ALLGATHERV

Пример 4.14 Версия Примера 4.2. Используя Allgather, мы соберём 100 целых с каждого процесса группы на каждом процессе.

```
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);
```

После вызова каждый процесс будет иметь набор множеств данных от всех процессов группы.

4.8 Все-всем рассылка/сбор

```
MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvtbuf, recvcount,
             recvtype, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
IN     sendcount            количество элементов в буфере посылки
                            (integer)
IN     sendtype             тип данных в буфере посылки (handle)
OUT    recvbuf              адрес получаемого буфера (choice)
IN     recvcount            количество элементов получаемых от каждого
                            процесса (integer)
IN     recvtype             тип данных в принимающем буфере (handle)
IN     comm                 коммуникатор (handle)

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

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

MPI_ALLTOALL является расширением MPI_ALLGATHER на случай, когда каждый процесс посылает конкретные данные каждому получателю. J-ый блок посылаемый процессом i, будет получен процессом j и помещён в i-ый блок буфера recvbuf.

Сигнатура типа ассоциированная с sendcount, sendtype на процессе должна совпадать с сигнатурой типа ассоциированной с recvcount, recvtype любого другого процесса. Это подразумевает, что количество посылаемых и принимаемых данных должно совпадать попарно для каждой пары процессов. Как обычно, карты типов могут быть

различны.
Результат получается как если бы каждый процесс передал сообщение каждому процессу (включая себя), вызвав

MPI_SEND(sendbuf + i¦sendcount¦extent(sendtype), sendcount, sendtype, i, ...),

и принял от каждого узла, вызвав

MPI_RECV(recvbuf + i¦recvcount¦extent(recvtype), recvcount, recvtype, i, ...).

Все аргументы значимы на всех процессах. Аргумент comm должен иметь одно и то же значение на всех процессах.

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

IN	sendbuf	адрес начала посылаемого буфера (choice)
IN	sendcounts	массив целых (с длиной равной размеру группы), содержащий количество элементов посылаемых каждому процессу
IN	sdispls	массив целых (с длиной равной размеру группы), на j-ом месте которого стоит смещение относительно sendbuf, откуда берутся данные для передачи процессу i
IN	sendtype	тип данных в буфере посылки (handle)
OUT	recvbuf	адрес получаемого буфера (choice)
IN	recvcounts	массив целых (с длиной равной размеру группы), содержащий максимальное количество элементов, которое может быть получено от каждого процесса
IN	rdispls	массив целых (с длиной равной размеру группы), на i-ом месте которого стоит смещение относительно recvbuf, куда поместить данные получаемые от процесса i
IN	recvtype	тип данных в принимающем буфере (handle)
IN	comm	коммуникатор (handle)

int MPI_Alltoallv(void *sendbuf, int sendcounts, int *displs, MPI_Datatype sendtype, void *recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, IERROR

MPI_ALLTOALLV добавляет гибкости MPI_ALLTOALL, так что размещение передаваемых данных специфицируется sdispls, а место для размещения принимаемых данных специфицируется с помощью rdispls.
J-ый блок данных посылаемых процессом i принимается процессом j и помещается в i-ый блок буфера recvbuf. Для разных i размер блока может быть разным.
Сигнатура типа ассоциированная с sendcounts[j], sendtype на процессе i должна совпадать с сигнатурой типа ассоциированной с recvcount[i], recvtype процесса j. Это подразумевает, что количество посылаемых и принимаемых данных должно совпадать попарно для каждой пары процессов. Различие карты типов принимающего и передающего допустимо.
Результат получается как если бы каждый процесс передал сообщение каждому процессу (включая себя), вызвав

MPI_SEND(sendbuf + sdispls[i], sendcounts[i], sendtype, i, ...),

и принял сообщение от каждого процесса, вызвав

MPI_RECV(recvbuf + rdispls[i], recvcounts[i], recvtype,
i, ...).

Все аргументы значимы на всех процессах. Аргумент comm должен иметь одно и то же значение на всех процессах.

Целесообразность. Определения MPI_ALLTOALL и MPI_ALLTOALLV дают почти такую же гибкость, какую можно достичь используя n независимых обменов точка с точкой. Только два исключения: все сообщения используют один тип данных, и сообщения рассылаются с (или собираются в) последовательно. (Конец замечания.)

Совет разработчикам. Хотя обсуждение групповых обменов в понятиях передач точка-с-точкой подразумевает, что каждое сообщение передаётся непосредственно от передающего принимающему, при реализации возможно использование обменов по дереву. Сообщения могут направляться непосредственно узлам, где они разбиваются на блоки (для рассылающего) или связываются (для собирающего), если это более эффективно. (Конец совета разработчикам.)

4.9 Групповые операции

Функции в этой части производят групповые операции (такие как sum, max, логический AND, и так далее) над всеми членами группы. Возможно использование либо одной из списка функций, либо определённой пользователем. Имеется несколько видов групповых операций: которые возвращают результат на один узел, которые возвращают этот результат всем узлам, и операция сканирования (префиксная). В дополнение определяется групповая операция с рассылкой она сочетает функциональные возможности групповой операции и операции рассылки.

4.9.1 Групповая операция

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

IN	sendbuf	адрес передаваемого буфера (choice)
OUT	recvbuf	адрес принимаемого буфера (choice, значимо только для корня)
IN	count	количество элементов в передаваемом буфере (integer)
IN	datatype	тип данных в передаваемом буфере (handle)
IN	op	операция (handle)
IN	root	ранк корневого процесса (integer)
IN	comm	коммуникатор (handle)

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

MPI_REDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_REDUCE выполняет операцию op над элементами входного буфера каждого процесса группы и возвращает получившееся значение в выходной буфер процессора с ранком root. Входной буфер определяется аргументами sendbuf, count и datatype; выходной буфер определяется аргументами recvbuf, count и datatype, то есть оба содержат одинаковое количество элементов одного и того же типа. Подпрограмма вызывается всеми членами группы используя одинаковые значения аргументов count, datatype, root и comm. Таким образом все процессы определяют входной и выходной буфера одинаковой длины с элементами одинакового типа. Каждый процесс может определить, как один элемент так и целую последовательность, в последнем случае операция выполняется поэлементно над каждым членом

последовательности. Например, если операция MPI_MAX и буфер содержит два элемента, которые являются числами с плавающей точкой (count = 2 и datatype = MPI_FLOAT), тогда recvbuf(1) = globalmax(sendbuf(1)) (то есть максимум выбирается среди значений sendbuf(1) всех процессов группы) recvbuf(2) = globalmax(sendbuf(2)).

Ниже приводится список операций реализуемых MPI; для каждой операции приводится список применяемых типов данных. Пользователи могут дополнительно определить свои собственные операции, которые можно написать, чтобы они работали с различными типами данных, как базовыми, так и с выведенными. Это подробнее обсуждается в разделе 4.9.4.

Операция op всегда полагается ассоциативной. Все предопределённые в MPI операции ещё и коммутативны. Пользователи могут определять операции, которые не являются коммутативными. "Канонический" порядок выполнения операций определяется ранками процессов в группе. Однако при реализации возможно использование ассоциативности, либо ассоциативности и коммутативности для изменения порядка выполнения операций. Это может приводить к изменению результатов групповых операций, которые не являются строго ассоциативными и коммутативными, например, сложение чисел с плавающей точкой.

> Совет разработчикам. Настоятельно рекомендуется, чтобы реализация MPI_REDUCE отвечала следующему условию: при выполнении функции с одними и теми же аргументами, которые появляются в одинаковом порядке должен получаться одинаковый результат. Заметим, что это может ограничить возможности оптимизаций, которые используют физическое размещение процессоров. (Конец совета разработчикам.)

Аргумент datatype может быть выведенным типом данных. В этом случае каждый аргумент используемый групповой операцией является элементом данного типа, который может содержать несколько базисных значений. Подробнее об этом в разделе 4.9.4.

4.9.2 Определённые в MPI групповые операции

Следующие операции определены для MPI_REDUCE и соответственно для функций MPI_ALLREDUCE, MPI_REDUCE_SCATTER и MPI_SCAN. Эти операции вызываются, используя следующие значения в качестве аргумента op.

Значение	Операция
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логический and
MPI_BAND	побитовый and
MPI_LOR	логический or
MPI_BOR	побитовый or
MPI_LXOR	логический xor
MPI_BXOR	побитовый xor
MPI_MAXLOC	максимум с индексом
MPI_MINLOC	минимум с индексом

Две операции MPI_MINLOC и MPI_MAXLOC обсуждаются позднее. Ниже мы перечисляем допустимые комбинации аргументов op и datatype для остальных операций. Сначала определим группы основных типов данных MPI, следующим способом.

C integer: MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_USIGNED, MPI_UNSIGNEG_LONG

Fortran integer: MPI_INTEGER

Floating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL,
MPI_DOUBLE_PRECISION

Logical: MPI_LOGICAL

Complex: MPI_COMPLEX

Byte: MPI_BYTE

А теперь, правильные типы данных для каждой операции.
Операция

Допустимые типы
MPI_MAX, MPI_MIN, MAXLOC, MINLOC
C integer, Fortran integer,
Floating point

MPI_SUM, MPI_PROD
C integer, Fortran integer,
Floating point, Complex

MPI_LAND, MPI_LOR, MPI_LXOR
C integer, Logical

MPI_BAND, MPI_BOR, MPI_BXOR
C integer, Byte
Пример 4.15 Подпрограмма вычисляет скалярное произведение двух
векторов, которые распределены по группе процессоров, и возвращает
ответ нулевому узлу.

```
SUBROUTINE PAR_BLAS1(m, a(m), b(m), c, comm)
REAL a(m), b(m)             ! локальная часть массива
REAL c                      ! результат (на нулевом узле)
REAL sum
INTEGER m, comm, i, ierr

! локальная сумма
sum = 0.0
DO i = 0, m
       sum = sum + a(i)*b(i)
END DO

! глобальная сумма
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)

! возвращает результат на нулевом узле (и мусор на остальных узлах)
RETURN c
```

Пример 4.16 Подпрограмма, которая вычисляет произведение вектора на
матрицу, которая распределена по группе процессов и возвращает
результат нулевому узлу.

```
SUROUTINE PAR_BLAS2(m, n, a(m), b(m, n), c(n), comm)
REAL a(m), b(m, n)                ! локальная часть массива
REAL c(m)                         ! результат
REAL sym(m)
INTEGER n, comm, i, j, ierr

! локальная сумма
DO j = 1, n
       sum(j) = 0.0
       DO i = 1, m
              sum(j) = sum(j) + a(i)*b(i, j)
       END DO
END DO
```

```
! глобальная сумма
CALL MPI_REDUCE(sum, c, m, MPI_REAL, MPI_SUM, 0, comm, ierr)

! возвращает результат на нулевом узле (и мусор на остальных узлах)
RETURN (c)
```

4.9.3 MINLOC и MAXLOC

Функция MPI_MINLOC (MPI_MAXLOC) используется, чтобы вычислить глобальный минимум (максимум) и возвращает минимальное (максимальное) значение вместе с соответствующим индексом. Её можно использовать, чтобы вычислить глобальный минимум (максимум) и ранк процесса содержащего это значение. Эти групповые операции определяются для действий над аргументами, состоящими из пары: значение и индекс. Потенциально смешанная структура таких аргументов представляет проблему для Fortran'a. Эту проблему можно обойти, используя индексы того же типа, что и значение, на Fortran'e.

Операции MPI_MINLOC и MPI_MAX_LOC применяются к парам значений. Операция, которая определяется MPI_MAXLOC это

где

w = max(u, v)

и

MPI_MINLOC определяется аналогично:

где

w = min(u, v)

и

Обе операции ассоциативны и коммутативны. Заметим, что операция MPI_MAXLOC применяется к последовательности пар (u0, 0), (u1, 1), ..., (un-1, n-1), возвращаемым значением будет (u, r), где u = max ui и r это индекс первого глобального максимума в последовательности. Таким образом, если каждый процесс снабдит значение ранком внутри группы, тогда групповая операция с op = MPI_MAXLOC вернёт максимальное значение и ранк первого процесса, содержащего это значение. Аналогично MPI_MINLOC можно использовать, чтобы получить минимум и его индекс. Более обобщённо, MPI_MINLOC вычисляет лексикографический минимум, где элементы упорядочиваются согласно первой компоненте пары, а спорные ситуации разрешаются согласно второй компоненте.

Для того, чтобы использовать MPI_MINLOC и MPI_MAXLOC в групповых операциях необходимо выбрать тип данных, который представляет пары (значение и индекс). MPI определяет семь таких типов данных. Операции MPI_MAXLOC и MPI_MINLOC можно использовать с каждым из этих типов.

MPI_2REAL	пара REAL
MPI_2DOUBLE_PRECISION	пара DOUBLE PRECISION переменных
MPI_2INTEGER	пара INTEGER
MPI_FLOAT_INT	float и int
MPI_DOUBLE_INT	double и int
MPI_LONG_INT	long и int
MPI_2INT	пара int

Тип данных MPI_2REAL можно было бы получить вызвав MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL) (смотрите Часть 3.12). Аналогично MPI_2INTEGER, MPI_2DOUBLE_PRECISION и MPI_2INT.

Тип данных MPI_FLOAT_INT можно было бы получить выполнив следующую последовательность команд:

```
type[0] = MPI_FLOAT;
type[1] = MPI_INT;
disp[0] = 0;
disp[1] = sizeof(float);
```

```
block[0] = 1;
block[1] = 1;
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT);
```

Аналогично MPI_LONG_INT и MPI_DOUBLE_INT.

Пример 4.17 Каждый процесс имеет массив из 30 double в Си. Для каждого из 30 значений вычисляется значение и ранк процесса, содержащего наибольшее значение.

```
...
/* каждый процесс имеет массив из 30 double: a[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(MPI_COMM_WORLD, 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);
/* С этой точки ответ находится на процессоре root */

if (myrank == root) {
       /* переписываем в наш буфер */
       for (i = 0; i < 30; ++i) {
              aout[i] = out[i].val;
              ind[i] = out[i].rank;
       }
}
```

Пример 4.18 Тоже самое на Fortran'e.

```
...
! каждый процесс имеет массив из 30 double: a(30)
DOUBLE PRECISION ain(30), aout(30)
INTEGER ind(30)
DOUBLE PRECISION in(2, 30), out(2, 30)
INTEGER i, myrank, root

MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
DO i = 1, 30
       in(1, i) = ain(i)
       in(2, i) = myrank           ! myrank преобразуется в double
END DO
MPI_REDUCE(in, out, 30, MPI_2DOUBLE, MPI_MAXLOC, root, comm)
! С этой точки ответ находится на процессоре root

IF(myrank .EQ. root)
       ! переписываем в наш буфер

       DO i = 1, 30
              aout(i) = out(1, i)
              ind(i) = out(2, i) ! rank преобразуется обратно
       END DO
END IF
```

Пример 4.19 Каждый процесс имеет непустой массив значений. Найти глобально минимальное значение, ранк процесса, который его содержит и его индекс на этом процессе.

```
#define       LEN    1000

float val[LEN];                     /* локальный массив значений */
int count;                          /* локальное количество значений */
int myrank, minrank, minindex;
float minval;

struct
       {
       float value;
       ind index;
       } in, out;

       /* локальная часть */
in.value = val[0];
in.index = 0;
for(i = 1; i < count; i++)
       if(in.value < val[i])
              {
              in.value = val[i];
              in.index = i;
              }
       /* глобальная часть */
MPI_Comm_rank(MPI_COMM_WORLD, myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce(in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
       /* С этой точки ответ находится на процессоре root */
if(myrank == root)
       {
       /* читаем результат из out */
       minval = out.value;
       minrank = out.index / LEN;
       minindex = out.index % LEN;
       }
}
```

Целесообразность. Определение MPI_MINLOC и MPI_MAXLOC данное здесь имеет преимущество, в том что не требует специального описания этих двух функций: и они используются так же как другие групповые операции. Программист может построить своё собственное определение, для функций аналогичных MPI_MAXLOC и MPI_MINLOC если пожелает. Недостатки этого определения в том, что значения и индексы должны сначала разложены, и индексы и значения должны быть приведены к одному типу в Fortran'e.(Конец замечания.)

4.9.4 Операции определённые пользователем

```
MPI_OP_CREATE(function, commute, op)
IN     function      функция определённая пользователем (function)
IN     commute       true если коммутативна; false иначе
OUT    op            операция (handle)

int MPI_Op_create(MPI_Uop function, int commute, MPI_Op *op)

MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
       EXTERNAL FUNCTION
       LOGICAL COMMUTE
       INTEGER OP, IERROR
```

MPI_OP_CREAT связывает определённую пользователем операцию с хэндлером op, который потом можно использовать в MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER и MPI_SCAN. Определённая пользователем операция должна быть ассоциативной. Если commute = true, тогда операция должна быть ещё и коммутативной. Если, commute = false, тогда порядок операций фиксирован и определяется порядком

ранков процессов, начиная с нулевого процесса.
function это определённая пользователем функция, которая должна иметь следующие четыре аргумента: invec, inoutvec, len и datatype.
ANSI-C прототип для этой функции имеет следующий вид

```
typedef void MPI_Uop(void *invect, void *inoutvec, int *len,
       MPI_Datatype *datatype);
```

Определение для Fortran'а имеет следующий вид.

```
FUNCTION UOP(INVEC(*), INOUTVEC(*), LEN, TYPE)
       <type> INVEC(LEN), INOUTVEC(LEN)
       INTEGER LEN, TYPE
```

Аргумент datatype это тип данных, который передаётся MPI_REDUCE. Определённая пользователем функция должна удовлетворять следующим условиям. Пусть u[0], ..., u[len-1] это len элементов в буфере описанном аргументами invec, len и datatype, и v[0], ..., v[len-1] это len элементов в буфере описанном аргументами inoutvec, len и datatype, до вызова функции; w[0], ..., w[len-1] это len элементов в буфере описанном аргументами inoutvec, len и datatype, по окончании работы функции. Тогда w[i] = u[i]¦v[i], для i=0, ..., len-1, где ¦ это операция, значение которой вычисляет функция.
Неформально, мы можем рассматривать invec и inoutvec как массивы из len элементов, на которые воздействует функция. Получившийся результат записывается в inoutvec, отсюда и название. Каждый вызов функции производит поточечную операцию над len элементами. То есть функция возвращает в inoutvec[i] значение invec[i] ¦ inoutvec[i], для i = 0, ..., count-1, где ¦ это операция вычисляемая функцией.

Целесообразность. Аргумент len позволяет избежать вызова функции для каждого элемента входного буфера. Вместо этого система сможет выбрать часть вводного буфера. В Си посылается по ссылке для совместимости с Fortran'ом.

С помощью внутреннего сравнения значения аргумента типа данных с известными, возможно использование одной определённой пользователем функции для нескольких различных типов данных.(Конец замечания.)

Общие типы данных могут быть переданы функции пользователя. Однако использование ненепрерывных типов данных может привести к неэффективности.
Внутри определённой пользователем функции запрещён вызов функций обмена. В случае ошибки можно вызвать MPI_Abort() внутри функции.

Совет пользователям. Допустим, что есть библиотека определённых пользователем функций, которые перекрываются: аргумент datatype используется для выбора правильного пути при каждом вызове, согласно типу данных операндов. Определённая пользователем функция не может "декодировать" присылаемый аргумент datatype, и не может сама идентифицировать соответствие между заголовком типа и типом данных, который он представляет. Это соответствие определяется при создании типов данных. Перед использованием библиотеки необходимо выполнить её предварительную инициализацию. Этот предварительный код определит типы данных, которые используются библиотекой, и сохраняет заголовки в глобальных, статических переменных, которые используются и библиотечными, и пользовательскими программами.

Fortran версия MPI_REDUCE будет вызывать определённую пользователем функцию используя Fortran соглашения о связях и передаёт Fortran типы данных, Си версия использует Си

соглашения о связях и Си представления типов данных. Пользователи, которые планируют использовать оба языка должны учитывать это и определять свои функции соответственно. (Конец совета пользователям.)

Совет разработчикам. Мы приводим ниже наивную (неэффективную) реализацию MPI_REDUCE

- если (rank < groupsize), то MPI_SEND(sendbuf, count, datatype, rank+1,...) (* посылаем вводной буфер следующему процессу *)
- если (rank > 0) то MPI_RECV(tempbuf, count, datatype, rank-1, ...) (* принимаем содержимое вводного буфера от предыдущего процесса во временный буфер; полученное сообщение займёт память начиная с lb(datatype) до lb(datatype) + count*extent(datatype) *)
- применяем user_reduce_function(tempbuf, recvbuf, count, datatype)

Более эффективную реализацию можно получить, заменив последовательный порядок выполнения, применённый выше, на логарифмическое дерево. Также можно сократить размер временного буфера при этом совместив обмен данными и вычисления, с помощью разбиения буфера на части размером len < count.

Предопределённые групповые операции можно реализовать как библиотеку определённых пользователем функций. Однако если MPI_REDUCE будет обрабатывать их специальным образом, можно добиться лучшей производительности. (Конец совета разработчикам.)

MPI_OP_FREE(op)
IN op операция (handle)

int MPI_op_free(MPI_Op *op)

MPI_OP-FREE (OP, IERROR)
INTEGER OP, IERROR

Отмечает операцию определённую пользователем для освобождения и устанавливает op в MPI_OP_NULL.

Пример определённой пользователем групповой операции

А теперь пример определённой пользователем групповой операции.
Пример 4.20 Вычисление произведения массива комплексных чисел на Си.

```
typedef struct {
       double real, imag;
} Complex;
/* определённая 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++;
       }
}
```

```
/* и чтобы вызвать её */
...
/* каждый процесс имеет массив из 100 комплексных чисел */
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;
/* Объясняем как в MPI определяется тип Complex*/
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
/* создать пользовательскую операцию произведение комплексных */
MPI_Op_create(myProd, True, &myOp);

MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);

/* С этой точки answer, который состоит из 100 Complex,
       размещается на процессе с ранком root */
```

4.9.5 Групповая операция всем

MPI включает варианты всех групповых операций, при которых результат возвращается всем процессам группы. MPI требует, чтобы все процессы, участвующие в этих операциях, получали идентичные результаты.

```
MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)
IN     sendbuf              адрес передаваемого буфера (choice)
OUT    recvbuf              адрес принимаемого буфера (choice)
IN     count                количество элементов в передаваемом буфере
                            (integer)
IN     datatype             тип данный в передаваемом буфере (handle)
IN     op                   операция (handle)
IN     comm                 коммуникатор (handle)
```

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

MPI_ALLREDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

Такая же как MPI_REDUCE за исключением того, что результат появляется в приёмном буфере всех членов группы.

Совет разработчикам. Групповую операцию всем можно реализовать как обычную групповую операцию, за которой следует операция распространения. Однако прямая реализация может привести к более высокой производительности. (Конец совета разработчикам.)

Пример 4.21 Подпрограмма вычисляет произведение матрицы, которая распределена по процессам группы, на вектор и возвращает ответ всем узлам.

```
SUBROUTINE PAR_BLAS2(m, n, a(m), b(m, n), c(n), comm)
REAL a(m), b(m, n)                ! локальная часть массива
REAL c(m)                         ! результат
REAL sum(m)
INTEGER n, comm, i, j, ierr

! локальная сумма
DO j = 1, n
       sum(j) = 0.0
       DO i = 1, m
              sum(j) = sum(j) + a(j)*b(i, j)
       END DO
END DO
```

```
! глобальная сумма
CALL MPI_ALLREDUCE(sum, c, m, MPI_REAL, MPI_SUM, 0, comm, ierr)

! возвращает результат на всех узлах
RETURN (c)
```

4.10 Групповая операция с рассылкой

MPI включает варианты всех групповых операций, при которых по окончании результат рассылается всем процессам группы.

```
MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op,
              comm)
IN     sendbuf              адрес начала передаваемого буфера (choice)
OUT    recvbuf              адрес начала приёмного буфера (choice)
IN     recvcounts           массив целых содержащий количества элементов
                            распределённых на каждый процесс. Массив
                            должен быть одинаковым на всех вызвавших
                            процессах
IN     datatype             тип данных вводного буфера (handle)
IN     op                   операция (handle)
IN     comm                 коммуникатор (handle)

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

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP,
COMM, IERROR)
       <type>SENDBUF(*), RECVBUF(*)
       INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

MPI_REDUCE_SCATTER сначала выполняет поэлементную групповую операцию над векторами хранящимися в буферах определяемых с помощью sendbuf, count и datatype, где count = ¦recvcounts[i]. Затем получившийся вектор разбивается на n непересекающихся сегментов, где n - количество элементов группы; сегмент i содержит recvcounts[i] элементов. I-ый сегмент посылается процессу i и сохраняется в приёмном буфере определяемом recvbuf, recvcounts[i] и datatype.

Совет разработчику. Процедура MPI_REDUCE_SCATTER функционально эквивалентна: функции MPI_REDUCE с count равным сумме recvcounts[i], за которой следует MPI_SCATTERV с sedcounts равным recvcounts. Однако прямая реализация может работать быстрее. (Конец совета разработчику.)

4.11 Сканирование

```
MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)
IN     sendbuf              адрес начала посылаемого буфера (choice)
OUT    recvbuf              адрес начала принимаемого буфера (choice)
IN     count                количество элементов в буфере (integer)
IN     datatype             тип данных в буфере (handle)
IN     op                   операция (handle)
IN     comm                 коммуникатор (handle)

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

MPI_SCAN (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
       <type>SENDBUF(*), RECVBUF(*)
       INTEGER COUNT, DAATYPE, OP, COMM, IERROR
```

MPI_SCAN используется, чтобы произвести над данными префиксную операцию над распределёнными по группе данными. Операция возвращает в приёмный буфер процесса с ранком i, результат групповой операции над значениями посылаемых буферов процессов с ранками 0, ..., i

включительно. На тип данных буфера приёма и передачи налагает ограничения выполняемая операция, аналогично MPI_REDUCE.

Целесообразность. Мы определили включающий scan, при котором групповая операция для iого процесса использует данные от процесса i. Можно напротив определить исключающий scan, где iый результат включает данные до i-1 и не использует данные iого процесса. Можно использовать оба определения. Второй вариант имеет несколько преимуществ: включающий scan может быть вычислен используя значение исключающего scan без дополнительных обменов; для необратимых операций таких как min или max потребуются дополнительные обмены, чтобы получить исключающий scan из включающего. С исключающим сканированием, однако, есть одна проблема, необходимо определить "единичный" элемент, который будет результатом для процесса 0. Это сложно для операций определяемых пользователем, поэтому мы остановились на включающем сканировании. (Конец замечания.)

4.12 Корректность

Корректная, совместимая программа должна использовать групповые операции так, чтобы не возникали тупиковые ситуации, независимо от того синхронизующие или нет эти операции. Следующие примеры иллюстрируют опасности использования групповых процедур.

Пример 4.22 Неверная программа

```
switch(rank)
        {
        case 0:       MPI_Bcast(&var1, count, type, 0, comm);
                      MPI_Bcast(&var2, count, type, 1, comm);
                      break;
        case 1:       MPI_Bcast(&var2, count, type, 1, comm);
                      MPI_Bcast(&var1, count, type, 0, comm);
                      break;
}
```

Мы допускаем, что группа ассоциированная с comm это {0,1}. Два процесса выполняют две операции распространения в обратном порядке. Если операции синхронизующие то произойдёт тупиковая ситуация.

Групповые операции должны выполняться в одинаковом порядке всеми членами группы.

Пример 4.23 Неверная программа

```
switch(rank)
        {
        case 0:       MPI_Bcast(&var1, count, type, 0, comm0);
                      MPI_Bcast(&var2, count, type, 2, comm2);
                      break;
        case 1:       MPI_Bcast(&var1, count, type, 1, comm1);
                      MPI_Bcast(&var2, count, type, 0, comm0);
                      break;
        case 2:       MPI_Bcast(&var1, count, type, 2, comm2);
                      MPI_Bcast(&var2, count, type, 1, comm1);
                      break;
        }
```

Мы допускаем, что группа ассоциированная с comm0 это {0,1}, с comm1 это {1,2}, а с comm2 это {2,0}. Если операции синхронизующие то возникнет циклическая зависимость: операция с comm2 закончится только после окончания операции с comm0; операция с comm0 закончится только после окончания операции с comm1; и операция с comm1 закончится только после окончания операции с comm2. Таким образом произойдёт тупиковая ситуация.

Групповые операции должны выполняться в такой последовательности, чтобы не появлялась циклическая зависимость.

Пример 4.24 Неверная программа

```
switch(rank)
        {
        case 0:       MPI_Bcast(&var1, count, type, 0, comm);
                      MPI_Send(&var2, count, type, 1, tag, comm);
                      break;
        case 1:       MPI_Recv(&var2, count, type, 0, tag, comm);
                      MPI_Bcast(&var1, count, type, 0, comm);
                      break;
        }
```

Процесс 0 выполняет распространение и затем следует блокирующая операция передачи; процесс 1 сначала выполняет блокирующий приём, за которым следует вызов операции распространения на процесс 0. Эта программа может зайти в тупик. Вызов операции распространения на процессе 0 может не возвращать управление до тех пор пока не выполнится соответствующий вызов на процессе 1, таким образом не выполнится передача. Процесс 1 определённо блокируется до окончания приёма, и таким образом никогда не выполнит операцию распространения.

Относительный порядок выполнения групповых операций и обменов точка с точкой должен быть таким, чтобы не возникали тупиковые ситуации даже если и групповые операции, и обмены будут синхронизующими.

Пример 4.25 Корректная, но недетерминированая программа

```
switch(rank)
        {
        case 0:       MPI_Bcast(&var1, count, type, 0, comm);
                      MPI_Send(&var2, count, type, 1, tag, comm);
                      break;
        case 1:       MPI_Recv(&var2, count, type, MPI_ANY_SOURCE,
                             tag, comm);
                      MPI_Bcast(&var1, count, type, 0, comm);
                      MPI_Recv(&var2, count, type, MPI_ANY_SOURCE,
                             tag, comm);
                      break;
        case 2:       MPI_Send(&var2, count, type, 1, tag, comm);
                      MPI_Bcast(&var1, count, type, 0, comm);
                      break;
        }
```

Все три процесса принимают участие в операции распространения. Процесс 0 посылает сообщение процессу 1 после операции распространения, а процесс 2 посылает сообщение процессу 1 до этой операции. Процесс 1 принимает сообщения до и после операции распространения, с произвольным номером источника в качестве аргумента.

Два варианта выполнения этой программы, для разных видов передач и приёмов, показаны на рисунке 4.10.

Заметим, что второй вариант выполнения имеет специфический эффект, который состоит в том, что сообщение передаваемое одним узлом после операции распространения принимается другим до этой операции. Этот пример иллюстрирует тот факт, что нельзя полагаться на синхронизующий эффект групповых функций. Программы, которые работают правильно только в случае выполнения команд в порядке появления (то есть когда распространения синхронизующие) ошибочны.

В сложных реализациях возможно одновременное выполнение одним процессом нескольких групповых операций. В таких случаях обязанность пользователя: обеспечить, чтобы один и тот же коммуникатор не использовался одновременно двумя разными групповыми операциями на одном процессе.

Совет системным программистам. Допустим, что операция распространения реализована используя MPI обмены точка с

точкой. Предположим использование следующих двух правил.

1. Все приёмы производятся с конкретный номером источника.

2. Каждый процесс посылает все сообщения относящиеся к одной групповой операции до передачи любого сообщения относящегося к следующей групповой операции.

Тогда сообщения относящиеся к закончившейся операции не могут быть ошибочными, так как порядок передач точка с точкой фиксирован.

Обязанность системного программиста обеспечить, чтобы обмены точка с точкой не перепутались с групповыми обменами. Один способ добиться этого, при создании коммуникатора создать также "скрытый коммуникатор" для групповых операций. Можно получить подобный эффект более простым способом, например, используя специальный тэг или бит контекста, который показывает используется ли коммуникатор для обменов точка с точкой или для групповых операций. (Конец совета системным программистам.)