3 Двухточечные обмены

3.1 Введение

Передача и прием сообщений процессами являются основным механизмом обменов MPI. Основные операции двухточечных обменов это - send и receive. Их использование показано на следующем примере.

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
 {
       char message[20];
       int myrank;
       MPI_Status status;
       MPI_Init( &argc, &argv);
       MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
       if (myrank == 0)        /* код для процесса ноль */
              {
              strcpy(message,"Hello, there");
              MPI_Send(message, strlen(message), MPI_CHAR, 1, 99,
              MPI_COMM_WORLD);
              }
       else                    /* код для процесса один */
              {
              MPI_Recv(message, 21, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
status);
              printf("received :%s:\n", message);
              }
       MPI_Finalize();
 }
```

В этом примере нулевой процесс (myrank = 0) посылает сообщение первому процессу, используя операцию передачи MPI_SEND. Для данной операции в памяти посылающего процесса указывается передаваемый буфер, из которого берутся данные для сообщения. В рассмотренном примере передаваемый буфер состоит из переменной message, находящейся в памяти нулевого процесса. Место нахождения, размер и тип посылаемого буфера определяются первыми тремя параметрами операции передачи. Посланное сообщение будет состоять из 21 символа этой переменной. Добавим что, операция передачи представляет сообщение как пакет. В этом пакете указывается место назначения сообщения и содержится различная информация, которая будет использоваться операцией приема для определения деталей сообщения. Три последних параметра операции передачи необходимы для самой операции.

Первый процесс (myrank = 1) получает это сообщение при помощи операции приёма MPI_RECV. Сообщение, которое будет принято выбирается согласно значению служебной части пакета, и данные сообщения запоминаются в приемном буфере. В рассмотренном выше примере, приемный буфер состоит из строки message в памяти процесса один. Первые три параметра операции приема указывают место нахождения, размер и тип приемного буфера. Следующие три параметра используются для определения поступающего сообщения. Последний параметр используется для возврата статуса только-что принятого сообщении.

Следующие части описывают основные (блокирующие) операции приема и передачи. Мы обсудим прием, передачу, основные семантики обменов, требования соответствия типов, преобразование типов в разнородных средах и более общие коммуникационные режимы. Затем рассмотрим неблокирующие обмены, канало-подобные конструкции и операции приема-передачи. Мы рассмотрим основные типы данных, которые позволяют одинаково передавать разнородные и ненепрерывные данные. Мы закончим описанием вызовов, для упаковывания и распаковывания сообщений.

3.2 Основные операции приема и передачи

3.2.1 Основная операция передачи

Синтаксис простейшей операции передачи приведен ниже.

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
IN     buf            адрес начала буфера передачи (choice)
IN     count          количество элементов в буфере передачи (integer)
IN     datatype       тип данных каждого элемента передаваемого буфера
                      (handle)
IN     dest           ранк приемника (integer)
IN     tag            тэг сообщения (integer)
IN     comm           коммуникатор (handle)

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

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

3.2.2 Данные сообщения

Буфер передачи указанный для операции MPI_SEND состоит из count последовательных записей типа указанного в datatype, и начинающихся с адреса buf. Заметим, что мы указываем длину сообщения в элементах, а не в байтах. Размер элемента машинно-зависим и скрыт от пользовательского уровня.

Часть сообщения, которая является данными, состоит из последовательности длиной count и типом обозначенным в datatype. count может быть нулем, в этом случае часть данных, этого сообщения будет пустой. Основные типы данных, которые могут быть указаны в качестве типа данных сообщения соответствуют основным типам данных базового языка. Возможные значения этого аргумента для Fortran'a и соответствующие типы Fortran'a приведены ниже.

MPI тип данных	Fortran тип данных
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Возможные значения этого аргумента для Си и соответствующие типы данных в Си приведены ниже.

MPI тип донных	Си тип данных
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

В языках Fortran и Си нет типов данных соответствующих типам MPI_BYTE и MPI_PACKED. Тип MPI_BYTE состоит из байта (8 двоичных цифр). Байт не интерпретируется и он отличается от символа. Различные машины могут иметь различное представление символов, или могут использовать более одного байта для представления символов. Другими словами байт имеет одно и тоже двоичное значение на всех машинах. Использование типа MPI_PACKED объяснено в части 3.13.

MPI требует поддержки типов данных описанных выше, которые являются основными типами данных Fortran'а 77 и ANSI Си. Дополнительные типы данных MPI могут быть обеспечены, если хост язык имеет дополнительные типы данных:
MPI_LONG_LONG_INT, для (64 бит) Си целых описанных типом longlong int; MPI_DOUBLE_COMPLEX для комплексных чисел двойной точности в Fortran'е описанных как DOUBLE PRECISION; MPI_REAL2, MPI_REAL4 и MPI_REAL8 для чисел с плавающей точкой в Fortran'е, описанных как REAL*2, REAL*4 и REAL*8, соответственно; MPI_INTEGER1, MPI_INTEGER2 и MPI_INTEGER4 для целых чисел в Fortran'е, описанных типами INTEGER*1, INTEGER*2 и INTEGER*4, соответственно; и т.д.

> Целесообразность. Одна из целей разработки - представить MPI в виде библиотеки, без необходимости дополнительной компиляции. Таким образом нельзя предполагать, что у коммуникационных вызовов информация о типах данных находится в коммуникационном буфере; эта информация должна быть предоставлена в аргументе. Необходимость в такой информации станет ясна из раздела 3.3.2. (Конец замечания.)

3.2.3 Служебная информация сообщения

Кроме данных, сообщение несет информацию, которая может использоваться для распознавания сообщений и их выборочного приема. Эта информация состоит из фиксированного количества полей, которые мы назовем служебной информацией сообщения. Это поля:

source

destination

tag

context

Источник сообщения точно определяется, посылающим сообщение. Другие поля указываются в качестве аргументов операции передачи.
Место назначения сообщения указывается в аргументе dest.
Признак сообщения, целого типа, указывается в аргументе tag.

Это целое может использоваться программой для распознавания разнообразных типов сообщений. Диапазон допустимых значений этого признака от 0 до UB, где значение UB зависит от реализации; оно может быть найдено из значения атрибута MPI_TAG_UB, как описано в главе 7. MPI требует, чтобы UB было не меньше чем 32767.

Аргумент comm определяет коммуникатор, который используется для операции передачи. Коммуникаторы описаны в главе 5; ниже мы коротко расскажем об их использовании.

Коммуникатор определяет коммуникационный контекст операции обменов. Каждый коммуникационный контекст обеспечивает особое "пространство обменов": сообщения всегда принимаются в контексте, в котором они посланы и сообщения, которые посланы в разных контекстах не пересекаются.

Коммуникатор также определяет те процессы, которые разделяют этот коммуникационный контекст. Эта группа процессов упорядочивается, и процессы отождествляются со своим ранком в этой группе. Таким образом, диапазон допустимых значений для dest от 0 до n-1, где n количество процессов в группе. (Если коммуникатор является интеркоммуникатором, тогда приемники определяются их ранком в удаленной группе. Смотри главу 5.)

MPI предоставляет предопределенный коммуникатор MPI_COMM_WORLD; он позволяет связываться со всеми процессами, которые доступны после инициализации MPI; процессы определяются с помощью своего ранка в группе MPI_COMM_WORLD. Пользователям, которым нравится понятие пространства имён для процессов и единственный коммуникационный контекст, как предлагается большинством существующих библиотек обменов, достаточно использовать только предопределенную переменную MPI_COMM_WORLD в качестве аргумента comm. Это позволит связываться со всеми инициализированными процессами.

Пользователи могут определять новые коммуникаторы, как описано в главе 5. Коммуникаторы обеспечивают важный механизм инкапсуляции для библиотек и модулей: это позволяет модулям иметь свою собственную коммуникационную среду и собственную схему нумерации процессов.

> Совет разработчикам. Служебная информация будет обычно закодирована в заголовке сообщения фиксированной длины. Однако, настоящее представление заголовка зависит от реализации: некоторая информация (источник или приемник) может подразумеваться, и отпадет необходимость передавать ее вместе с сообщением; процессы могут быть отождествлены с относительными ранками, или с абсолютными; и т.д. (Конец совета разработчикам.)

3.2.4 Основная операция приема

Синтаксис простейшей операции приема приведен ниже.

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
OUT    buf           адрес буфера приема (choice)
IN     count         количество элементов в буфере приема (integer)
IN     datatype      тип принимаемых данных в буфере (handle)
IN     source        ранк источника (integer)
IN     tag           тэг сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    status        объект статуса (Status)

int MPI_RECV(void* buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
       <type> BUF(*)
       INTEGER COUNT,DATATYPE, SOURCE, TAG, COMM,
               STATUS(MPI_STATUS_SIZE), IERROR
```

Приемный буфер состоит из count последовательных элементов,

типа указанного в datatype и начинается он с адреса buf. Длина принимаемого сообщения должна быть не больше длины приемного буфера; если все пришедшие данные, без усечения не входят в приемный буфер, возникает ошибка переполнения.

> Совет пользователям. Функция MPI_PROBE, описанная в разделе 3.8, может быть использована для приема сообщения неизвестной длины. (Конец совета пользователям.)

> Совет разработчикам. Хотя MPI не предписывается специальное поведение при выполнении ошибочных программ, рекомендуется при обработке ситуации переполнения, возвращать в status информацию об источнике приходящего сообщения и его признаке. Операция приема вернет код ошибки.

> Хорошая реализация будет также гарантировать, что память за пределами буфера приема не будет испорчена. (Конец совета разработчикам.)

Выбор сообщения операцией приема производиться согласно уникальному значению служебной информации сообщения. Сообщение может быть получено операцией приема, если значения полей служебной информации source, tag и context, данного сообщения, совпадают с соответствующими полями операции приема. Получатель может указать метасимвол MPI_ANY_SOURCE для поля source, и/или метасимвол MPI_ANY_TAG для поля tag, означающие, что принимаются сообщения с любыми значениями source и/или tag. Для поля context не может быть указан метасимвол. Таким образом, сообщение может быть принято операцией приема только если оно адресовано принимающему процессу, имеющему подходящий context, подходящий source, или source=MPI_ANY_SOURCE, и подходящий признак процесса tag, или tag=MPI_ANY_TAG.

Признак сообщения определяется аргументом tag операции приема. Поле сообщения context связано с коммуникатором, определяемом в поле comm. Источник сообщения, если он отличается от MPI_ANY_SOURCE, определяется как ранк в пределах группы процессов связанный с тем же самым коммуникатором (удаленная группа процессов, с интеркоммуникатором). Таким образом, диапазон допустимых значений для аргумента source { 0, ... , n-1 } ¦ { MPI_ANY_SOURCE }, где n количество процессов в группе.

Отметим ассиметрию между операциями приема и передачи: Операция приема может принимать сообщения от произвольного корреспондента; напротив, операция передачи должна определять уникальный приемник. Это подходит под коммуникационный механизм типа "push", где передача данных осуществляется отправителем (в отличие от механизма "pull", где передача данных осуществляется получателем).

Source = destination допускается: процесс может послать сообщение самому себе. (Однако ненадежно делать это с помощью основных операций приема и передачи, описанных выше, так как это может привести к взаимоблокировке; смотри часть 3.5)

> Совет разработчикам. К контексту сообщения может быть добавлено дополнительное поле признака. Оно отличается от обычного поля признака сообщения тем, что в этом поле не допустимо использование метасимвола, и тем, что значение установленное в данном поле контролируется функциями манипулирования коммуникаторами. (Конец совета разработчикам.)

3.2.5 Возврат статуса

Источник или признак принимаемого сообщения может быть не известен, если были использованы метасимволы в операции приема. Эта информация возвращается в аргументе status, функции MPI_RECV. Этот аргумент специально определенного в MPI типа. Необходимо, чтобы статусные переменные были выделены пользователем - это не компоненты системы.

В Си, status это структура, которая содержит два поля с именами MPI_SOURCE и MPI_TAG; структура может содержать дополнительные поля. Таким образом, status.MPI_SOURCE и status.MPI_TAG содержат,соответственно, ранк источника и признак принимаемого сообщения.

В Fortran'e status это массив типа INTEGER размером MPI_STATUS_SIZE. Две константы MPI_SOURCE и MPI_TAG определяют элементы, в которых хранятся ранк источника и признак сообщения. Таким образом, status(MPI_SOURCE) и status(MPI_TAG) содержат, соответственно, ранк источника и признак принимаемого сообщения.

Аргумент статус также возвращает информацию о длине принятого сообщения. Однако, эта информация не доступна как поле переменной status; чтобы "расшифровать" эту информацию необходимо вызвать MPI_GET_COUNT.

```
MPI_GET_COUNT(status, datatype, count)
IN     status              статус операции приема (Status)
IN     datatype            тип принимаемых данных (handle)
OUT    count               количество принятых элементов (integer)

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

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
       INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Возвращает количество принятых элементов. (Здесь, снова, мы считаем элементы, а не байты.) Аргумент datatype должен иметь то же значение, что и в операции приема, которая установила переменную status. (Мы увидим позже, в разделе 3.12.5, что MPI_GET_COUNT может вернуть в определенных ситуациях значение MPI_UNDEFINED.)

> Целесообразность. В некоторых библиотеках передачи сообщений используются INOUT аргументы count, tag и source, таким образом используя как для указания критерия отбора приходящих сообщений, иак и для возврата действительной служебной информации о полученном сообщении. Использование отдельного статусного аргумента предотвращает ошибки, которые часто возникают с INOUT аргументом (например, при передаче константы MPI_ANY_TAG в качестве действительного аргумента для tag). Некоторые библиотеки используют вызовы, которые полностью ссылаются на "последнее принятое сообщение".
>
> Аргумент datatype передается MPI_GET_COUNT, чтобы увеличить производительность: сообщение может быть принято без подсчета количества элементов, из которых оно состоит, и посчитанное значение часто не нужно. Это требование также позволяет использовать эту функцию после вызова MPI_PROBE. (Конец замечания.)

Все операции приема и передачи используют аргументы buf, count, datatype, source, dest, tag, comm и status точно также, как и основные операции MPI_SEND и MPI_RECV описанные в данном разделе.

3.3 Соответствие типов и преобразование данных

3.3.1 Правила соответствия типов

Предположим, что передача сообщения состоит из трех фаз:

1. Данные выходят из буфера передачи, и собираются в сообщение.

2. Сообщение передается от передатчика к приемнику.

3. Данные от пришедшего сообщения помещаются в приемный буфер.

Необходимо соблюдать соответствие типов на каждой из трех стадий: Тип каждой переменной в буфере передачи должен соответствовать типу каждой переменной в операции передачи; тип указанный в операции передачи должен соответствовать типу указанному в операции приема; и тип каждой переменной в приемном буфере должен соответствовать типу для этой переменной в операции приема. Программы, которые нарушают три рассмотренных правила являются ошибочными.

Определим соответствие типов более точно, мы имеем дело с двумя случаями: соответствие типов хост языка и типов коммуникационной операции; и соответствие типов у передатчика и приемника.

Тип указанный для операции передачи соответствует типу указанному для операции приема, если обе операции используют одинаковые имена типов: MPI_INTEGER соответствует MPI_INTEGER, MPI_REAL соответствует MPI_REAL, и так далее. Есть одно исключение из этого правила, которое будет обсуждаться в части 3.13: тип MPI_PACKED может соответствовать любому другому типу.

Тип переменной в хост программе соответствует типу указанному в операции обмена, если имя типа, используемого этой операцией, соответствует основному типу переменной хост программы: переменная с типом MPI_INTEGER соответствует переменной Fortran'а типа INTEGER, переменная с типом MPI_REAL соответствует Fortran'овской переменной типа REAL, и так далее. Из последнего правила есть два исключения: переменная типа MPI_BYTE или MPI_PACKED может использоваться для соответствия любой байтовой переменной (на байт-адресуемой машине), независимо от типа переменной содержащей этот байт. Тип MPI_PACKED используется для передачи данных, которые были запакованы, или для получения данных, которые будут распакованы; смотри часть 3.13. Тип MPI_BYTE позволяет передать двоичное значение байта без изменения.

Таким образом у нас есть три случая:

- Передача данных определенного типа (то есть, с типом данных отличным от MPI_BYTE), где типы данных в соответствующих местах посылающей программы, в вызове send, в вызове recive и в принимающей программе должны быть одинаковы.

- Передача данных неопределенного типа (то есть, с типом данных MPI_BYTE), где оба передатчик и приемник используют тип данных MPI_BYTE. В этом случае нет ни требований к типам в соответствующих местах посылающей и принимающей программ, ни требований, что они должны быть одинаковыми.

- Обмен упакованными данными, при использовании типа MPI_PACKED.

Следующие примеры иллюстрируют первые два случая.

Первая программа:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
  ELSE
    CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Эта программа правильна если оба массива a и b типа real и их размер ¦ 10. (В Fortran'е можно использовать эту программу, даже если a или b имеют размер < 10: то есть, когда a(1) может быть эквивалентно массиву из десяти чисел типа real.)

Вторая программа:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
```

```
    CALL MPI_SEND(a(1),10, MPI_REAL, 1, tag, comm, ierr)
  ELSE
    CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

Эта программа ошибочна, так как приемник и передатчик не обеспечивают соответствия типов данных аргументов.

Третья программа:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
  ELSE
    CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

Эта программа правильная, независимо от типа и размера a и b (пока это не приводит к выходу за пределы доступной памяти).

Совет пользователям. Если буфер типа MPI_BYTE передан как аргумент для MPI_SEND, MPI будет посылать последовательные данные, начиная с адреса указанного в аргументе buf. Может получиться непредсказуемый результат, когда расположение данных не случайное, пользователь должен следить за этим: Например, некоторые компиляторы Fortran'a создают переменные типа CHARACTER как структуры, которые содержат длину символа и указатель на символьную строку. В данном случае, если для посылки и приема переменных CHARACTER в Fortran'е используется тип MPI_BYTE, нет возможности предвидеть результаты передачи символьных строк. На этом основании пользователям рекомендуется использовать типовые средства связи где это возможно. (Конец совета пользователям.)

Тип MPI_CHARACTER

Тип MPI_CHARACTER скорее соответствует одному символу Fortran'овской переменной CHARACTER, чем полной символьной строке хранящейся в переменной. Fortran'овские переменные типа CHARACTER или подстроки передаются так, как если бы они были массивами символов. Это иллюстрируется примером ниже.

```
CHARACTER*10 a
CHARACTER*10 b

CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
  ELSE
    CALL MPI_RECV(B(6:10), 5, MPI_CHARACTER, 0, tag, comm, status,
ierr)
END IF
```

Последние пять символов строки b в процессе 1 заменяются на первые пять символов строки a в процессе 0.

Целесообразность. Альтернативный выбор должен быть для MPI_CHARACTER, чтобы соответствовать символам переменной длины. Это ведёт к ряду проблем.

Fortran'овская символьная переменная это строка постоянной длины, без специального заключительного символа. Нет жесткого соглашения, как представлять символы и как хранить их длину. Некоторые компиляторы передают символьный аргумент программе, как пару аргументов, один содержит адрес строки, а другой длину строки. Рассматривая случай коммуникационного вызова

MPI, который является передачей коммуникационного буфера с типом определенным при получении типа данных (Часть 3.12). Если этот переданный буфер содержит переменные типа CHARACTER, тогда информация об их длине не будет передана программе MPI.

Эта проблема заставляет нас передать определенную информацию о длине символа с вызовом MPI. Можно добавить параметр длины к типу MPI_CHARACTER, но это не добавит удобства и та же самая функциональность может быть достигнута определением подходящего передаваемого типа данных. (Конец замечания.)

Совет разработчикам. Некоторые компиляторы передают Fortran'овский символьный аргумент, как структуру с полем "длина" и указателем на строку. В подобном случае, вызову MPI необходимо отличать указатель для того, чтобы достичь строку. (Конец совета разработчикам.)

3.3.2 Преобразование данных

Одна из целей MPI это поддержка параллельных вычислений в разнородном окружении. Связь в разнородных средах может потребовать преобразования данных.

Мы используем следующую технологию:

преобразование типа изменяет значения типа, например округление REAL до INTEGER.

преобразование представления изменяет двоичное представление значения,например из шестнадцатиричного с плавающей точкой в IEEE с плавающей точкой.

Правила соответствия типов подразумевают, что в MPI обменах никогда не произойдет преобразования типов. Другими словами, MPI требует, чтобы преобразование представления было произведено, когда типовое значение передается через среду, которая использует отличное представление для типа этого значения. MPI не указывает правил для преобразования представления. Такое преобразование предполагает сохранение целых, логических и символьных значений, и преобразование значений с плавающей точкой к ближайшему значению, которое может быть представлено в данной системе.

Переполнение и потеря значимых разрядов может происходить во время преобразования чисел с плавающей точкой; преобразование целых и символьных переменных может также привести к потере значимости, когда переменная представленная в одной системе не может быть представлена другой системе. Потеря значимости происходит во время преобразования представления результата из-за ошибки связи; ошибка происходит или при операции передачи, или при операции приема, или при обоих операциях.

Если значение посланное в сообщении не имеет типа (т.е., имеет тип MPI_BYTE), тогда двоичное представление принятое приемником идентично двоичному представлению байта отправленного передатчиком. Это остаётся верным, работают ли приемник и передатчик в одинаковых или различных средах; преобразования представления не требуется. (Заметим, что преобразование представления могут происходить, когда переводятся значения типа MPI_CHAR, например из кодировки EBCDIC в кодировку ASCII.)

Заметим, что не происходит преобразования, когда MPI программа в однородной системе, где все процессы выполняются в том же самом окружении.

Рассмотрим предыдущие три примера.

Первая программа корректна, примем, что a и b массивы типа REAL размером ¦ 10. Если приемник и передатчик находятся в различных средах, тогда десять вещественных чисел, которые выбираются из буфера передатчика, будут преобразованы в вещественное представление со стороны приемника, перед тем, как они

будут записаны в буфер приемника. Несмотря на то, что количество вещественных чисел, выбранных из буфера передатчика, равно количеству вещественных чисел запомненных в буфере приемника, количество запомненных байт не обязано равняться количеству загруженных байт: например передатчик может использовать четырех-байтовое представление для вещественных чисел, а приемник восьми-байтовое представление.

Вторая программа ошибочна, и ее поведение не предсказуемо.

Третья программа корректна. Точно такая же последовательность из сорока байтов, которая была загружена из буфера передатчика, будет запомнена в буфере приемника, даже если передатчик и приемник работают в различных средах. Посланное сообщение имеет точно ту же длину (в байтах) и то же двоичное представление, как и принятое сообщение. Если a и b различных типов, или они одинакового типа, но используется различное представление, тогда биты запомненные в буфере приемника могут иметь значения отличные от переданных из буфера передатчика.

Преобразование представления данных, также касается и служебной информации сообщения: источника, место назначения и признака, все эти целые может быть потребуется преобразовать.

> Совет разработчикам. Текущее определение не требует, чтобы сообщение имело информацию о типе данных. Как передатчик, так и приемник обеспечивают полную информацию о типе данных. В разнородных средах, один из них может использовать машинно независимое кодирование, такое как XDR, или иметь приемник, конвертирующий из представления передатчика в свое собственное, или иметь передатчик выполняющий преобразование.
>
> Дополнительная информация о типе может быть добавлена к сообщению, для того чтобы дать возможность определить несоответствие между типом данных передатчика и приемника. Это может быть частично использовано в более медленном, но зато в более надежном отладочном режиме. (Конец совета разработчикам.)

MPI не требует поддержки для меж-языковой связи: Поведение программы непредсказуемо, если сообщение послано Си процессом, а принято Fortran процессом, или наоборот.

> Целесообразность. MPI не управляет меж-языковым взаимодействием, потому что, нет согласованных стандартов между типами Си и Fortran'а. Поэтому, MPI программы, которые смешивают языки непереносимы. (Конец замечания.)
>
> Совет разработчикам. Разработчики MPI могут захотеть поддерживать меж-языковое взаимодействие, позволяя программам на Fortran'е использовать "типы Си MPI", такие как MPI_INT, MPI_CHAR, и т.д., и позволять Си программам использовать типы Fortran'а. (Конец совета разработчикам.)

3.4 Режимы связи

Основной вызов передачи описанный в разделе 3.2.1 блокирующий: он не возвращает управление до тех пор, пока сообщение и его служебная информация не будут надежно запомнены так, чтобы передатчик стал свободен и мог использовать свой буфер. Сообщение может быть скопировано прямо в соответствующий буфер приемника; или, оно может быть скопировано во временный системный буфер.

Буферизация сообщений "развязывает" операции приема и передачи: Блокирующая передача может быть завершена только, когда сообщение будет буферизовано, даже если еще не выставлен соответствующий запрос приема приемником. С другой стороны, буферизация сообщения может дорого обойтись, так как она требует дополнительного копирования память-память, и выделения возможно скудной памяти под буферизацию. MPI предоставляет выбор нескольких режимов связи, которые позволяют управлять выбором

протокола связи.

Операция передачи описанная в разделе 3.2.1 использует стандартный режим связи. В этом режиме MPI решает будут ли исходящие сообщения буферизироваться. MPI может буферизировать исходящие сообщения. В этом случае, операция передачи может завершиться до выставления соответствующего запроса на прием. С другой стороны, буфер может быть не доступен, или MPI может не буферизировать выходящее сообщение из соображений производительности. В этом случае, операция передачи не будет завершена до тех пор, пока соответствующая операция приема не будет выставлена, и данные не поступят в приемник.

Таким образом, началась ли операция передачи в стандартном режиме или нет соответствующая операция приема должна быть выставлена. Передача в стандартном режиме не локальна: успешное завершение операции передачи зависит от появления соответствующей операции приема.

Есть три дополнительных режима связи:

В буферизующем режиме, началась ли операция передачи или нет, должна быть выставлена соответствующая операция приема. Операция передачи может завершиться до выставления соответствующей операции приема. Однако, в отличие от стандартной передачи, эта операция локальна, и ее завершение не зависит от появления соответствующей операции приема. Таким образом, если произошла передача и не выставлено соответствующей операции приема, тогда MPI должна буферизировать выходящее сообщение, так как это позволяет закончить передачу. Если не будет достаточно места для буферизации произойдет ошибка. Количеством места предназначенного для буферизации управляет пользователь - смотри часть 3.6. Может потребоваться выделение буфера пользователем, чтобы буферизованный режим был эффективен.

Передача в синхронном режиме может стартовать независимо от того, выставлена соответствующая операция приема или нет. Однако передача будет завершена успешно, только если соответствующая операция приема выставлена, и начавшаяся операция приема получит сообщение посланное синхронной передачей. (Т.е., операция приема выставлена, и входящее сообщение соответствует этой выставленной операции приема.) Таким образом, завершение синхронной передачи не только показывает, что передающий буфер может быть снова использован, но также показывает, что приемник достиг определенного места в своей работе, а именно, что он начал выполнять соответствующую операцию приема. Если оба, прием и передача, блокирующие операции, тогда используется синхронный режим, обеспечивающий синхронную семантику связи: связь не закончиться, пока оба процесса "присутствуют" в связи. Передача выполняющаяся в этом режиме не локальная.

Операция приема, которая использует связь по готовности, может быть запущена, только если уже выставлена соответствующая операция приема; иначе операция ошибочна и результат непредсказуем. В некоторых системах, это позволит не производить операцию сброса, которая требуется в противном случае, и приведёт к увеличению производительности. Завершение операции передачи не зависит от состояния выставленной операции приема, и означает только, что буфер передатчика может быть использован. Операция передачи, которая использует режим по готовности имеет ту же самую семантику, что и стандартная операция передачи, или операция синхронной передачи; просто передатчик обеспечивает дополнительную информацию системе (а именно, что соответствующая операция приема уже выставлена), это поможет избежать некоторых накладных расходов. В корректной программе, следовательно, передача по готовности может быть заменена на стандартную передачу без изменения поведения программы, разве только производительности.

Три дополнительных функции передачи обеспечивают три дополнительных режима связи. Режим связи обозначается одним буквенным префиксом: B для буферизующего, S для синхронного, и R по готовности.

Передача в буферизирующем режиме

```
MPI_BSEND(buf, count, datatype, dest, tag, comm)
IN     buf           начальный адрес буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип данных в буфере передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)

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

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Передача в синхронном режиме

```
MPI_SSEND(buf, count, datatype, dest, tag, comm)
IN     buf           начальный адрес буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип данных в буфере передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)

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

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Передача режиме по готовности

```
MPI_RSEND(buf, count, datatype, dest, tag, comm)
IN     buf           начальный адрес буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип данных в буфере передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)

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

MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Есть только одна операция приема, которая может соответствовать любому режиму передачи. Это блокирующая операция, описанная в последнем разделе: она возвращает управление тогда, когда приемный буфер будет содержать новое полученное сообщение. Прием может завершиться до завершения соответствующей передачи (конечно, он может завершиться только после запуска соответствующей передачи).

В многопотоковых реализациях MPI, система может отложить выполнение ветки, которая заблокирована операцией приема или передачи, и начать выполнение другой ветви в том же адресном пространстве. В этом случае пользователь не должен обращаться и модифицировать коммуникационный буфер до тех пор, пока не завершиться обмен; иначе исход вычислений не предсказуем.

Целесообразность. Мы запрещаем доступ на чтение к буферу передачи, пока он используется, хотя операция передачи не предполагает изменения содержимого этого буфера. Это может показаться более строгим, чем необходимо, но добавочное

ограничение вызывает небольшие потери функциональности и позволяет увеличить производительность на некоторых системах - принимая во внимание случай, где данные передаются через канал DMA, который не кэш-совместим с центральным процессором. (Конец замечания.)

Совет разработчикам. Так как синхронная передача не может завершиться прежде, чем будет выставлена соответствующая операция передачи, такая операция не будет нормально отправлять буферизованные сообщения.

Рекомендуется, где возможно, выбирать буферизирующую передачу, а не блокирующую, для стандартной передачи: программист может отдавать свое предпочтение блокирующей передаче до тех пор, пока соответствующая операция приема использует синхронный режим.

Возможный коммуникационный протокол, для возможных режимов связи приведен ниже:

передача по готовности: Сообщение передается как только возможно.

синхронная передача: Передатчик посылает запрос на посылку сообщения. Приемник принимает этот запрос. Когда выставлена соответствующая операция приема, приемник посылает обратно разрешение на прием сообщения, и тогда передатчик посылает сообщение.

стандартная передача: Первый протокол может быть использован для коротких сообщений, и второй протокол для длинных сообщений.

буферизирующая передача: Передатчик копирует сообщение в буфер и посылает его с помощью неблокирующей передачи (используя тот же самый протокол, как в стандартной передаче).

Дополнительные управляющие сообщения могут понадобиться для управления потоками сообщений и исправлением ошибок. Конечно, есть много других возможных протоколов.

Передача по готовности может выполняться, как стандартная передача; в этом случае не будет преимуществ (или потерь) в производительности при применении передачи по готовности.

Стандартная передача может быть выполнена как синхронная передача. В этом случае нет необходимости буферизации данных. Однако, многие (большинство ?) пользователи предполагают некоторую буферизацию.

В много-потоковой среде, выполнение блокирующей связи должно блокировать только выполняющуюся ветвь, позволяя программе управляющей потоками, снять этот поток и начать выполнять другой поток. (Конец совета разработчикам.)

3.5 Семантики двухточечных обменов

Существующие разновидности MPI гарантируют определенные общие свойства двухточечных обменов, которые мы опишем в этой части.

Порядок Сообщения не перекрываются: Если передатчик посылает два сообщения последовательно на тот же самый адрес, и оба соответствуют той же самой операции приема, тогда эта операция не может получить второе сообщение если все еще ожидается первое. Если приемник выставляет последовательно две операции приема, и обе соответствуют одному и тому же сообщению, тогда вторая операция

приема не будет удовлетворена этим сообщением, если первая все еще ожидает. Это требование содействует соответствию передач приемам. Это гарантирует, что код передаваемого сообщения определен, если процессы однопотоковые и в приемной операции не использован шаблон MPI_ANY_SOURCE. (Некоторые из вызовов описанных позднее, такие как MPI_CANCEL или MPI_WAITANY, являются дополнительными источниками неопределенностей.),

Если процесс имеет единственный поток выполнения, тогда любые два обмена выполняются этим процессом по порядку. С другой стороны, если процесс многопотоковый, тогда семантика выполнения соединения может не определить соответствующий порядок между двумя операциями передачи выполняемыми двумя различными ветвями: операции логически состязаются, даже если одна из них физически опередила другую. В этом случае, два посланных сообщения могут быть приняты в любом порядке. Если две операции приема, которые логически конкурируют, получат два последовательно посланных сообщения, тогда эти два сообщения могут соответствовать ожидаемым сообщениям в любом порядке.

Например:

```
CALL MPI_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
       CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
       CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
  ELSE  ! rank.EQ.1
       CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm,
       status, ierr)
       CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status,
       ierr)
END IF
```

Сообщение посланное первой передачей должно быть принято первой операцией приема, и сообщение посланное вторым должно быть принято вторым.

Развитие Если пара соответствующая приему и передаче была инициирована двумя процессами, тогда хотя бы одна из этих двух операций будет завершена, независимо от других действий в системе: операция передачи будет завершена если операция приема не примет другое сообщение и не завершится; операция приема будет завершена, если посланное сообщение не будет принято другой подходящей операцией приема, которая была выставлена принимающим процессом.

Пример:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
       CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
       CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
  ELSE  ! rank.EQ.1
       CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status,
       ierr)
       CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status,
       ierr)
END IF
```

Оба процесса делают свой первый коммуникационный вызов. Так как первая передача процесса ноль использует буферизирующий режим, она должна завершиться, независимо от состояния процесса один. Поскольку не выставлена операция приема, сообщение будет скопировано в буферное пространство. (Если не окажется буферного пространства, то программа выйдет из строя.) Затем выполняется вторая передача. В этом месте, возможна соответствующая пара операций приема и передачи, и обе операции должны завершиться. Затем, процесс один выполняет вторую операцию приема, которая будет удовлетворена буферизованным сообщением. Заметим, что процесс один

получает сообщения в порядке обратном передаче.

Недостатки MPI не дает ни каких гарантий относительно недостатков в управлении связью. Предположим, что передача выставлена. Тогда возможно, что принимающий процесс продолжает выставлять запросы на прием соответствующий данной передаче, сообщение может не быть получено, из-за того что, каждый раз перекрывалось другим сообщением, посланным от другого источника. Теперь, предположим, что запрос был выставлен мульти-потоковым процессом. Тогда возможно, что сообщения, которые соответствуют этому приему принимаются, но прием не будет удовлетворен, потому что он перекрывался другими запросами на прием с этого узла (от других выполняющихся потоков). Это задача программиста предотвращать зависание процесса в этой ситуации.

Ограничение ресурсов Любое ожидание операций связи расходует системные ресурсы, которые ограничены. Ошибки могут возникать, когда недостаток ресурсов препятствует выполнению MPI вызова. Реализация хорошего качества будет использовать (малое) фиксированное количество ресурсов для каждой отложенной передачи, как в режиме по готовности, так и в синхронном, и в ожидающем приеме. Однако, буферное пространство может расходоваться для хранения переданных сообщений в стандартном режиме, и должно расходоваться для хранения сообщений в буферизующем режиме, когда нет соответствующего приема. Количество места доступного для буферизации будет намного меньше, чем место занимаемое данными программы, на многих системах. Таким образом, совсем легко написать программу, которая будет выходить за границы возможного буферного пространства.

MPI позволяет пользователям предусмотреть наличие буферной памяти, для передачи сообщений в буферизующем режиме. Более того, MPI указывает подробную, рабочую модель для использования этого буфера. В реализации MPI требуется делать не хуже, чем подразумевает эта модель. Это позволит избежать пользователям переполнения буфера, когда они используют буферизующие передачи. Выделение буфера и его использование описывается в части 3.6

Буферизующая операция передачи, которая не может завершиться из-за недостатка буферного пространства ошибочна. Когда обнаруживается подобная ситуация, выдается сообщение об ошибке, которое может вызвать ненормальное завершение программы. С другой стороны, стандартная операция передачи, которая не может завершиться из-за недостатка буферного места будет просто заблокирована, ожидая освобождения буферного места или выставления соответствующего приема. Это поведение предпочтительно во многих ситуациях. Рассмотрим ситуацию, когда производитель неоднократно выдает новые значения и посылает их потребителю. Предположим, что производитель выдает значения быстрее, чем потребитель может принимать их. Если используется буферная передача, тогда результатом будет переполнение буфера; необходимо добавлять дополнительную синхронизацию в программу, чтобы этого не происходило. Если используется стандартная передача, тогда производитель будет автоматически приостановлен, так как операция передачи будет заблокирована, если буфер не доступен.

В подобных ситуациях, недостаток буферного пространства может привести к тупиковым ситуациям. Это проиллюстрировано на приведённых ниже примерах:

Пример 1:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status,
       ierr)
  ELSE    ! rank.EQ.1
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status,
```

```
       ierr)
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

Эта программа будет успешно выполняться, даже если отсутствует буферное пространство. Стандартная операция передачи может быть заменена на синхронную передачу.

Пример 2:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status,
       ierr)
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  ELSE    ! rank.EQ.1
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status,
       ierr)
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

Операция приема первого процесса, должна завершиться до его передачи, и может завершиться только если выполнится соответствующая передача у второго процесса; операция приема второго процесса должна завершиться до его передачи и может завершиться только если соответствующая передача выполняется у первого процесса. Эта программа будет всегда в безысходном положении. Тоже самое будет происходить с любым другим режимом передачи.

Пример 3:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status,
       ierr)
  ELSE    ! rank.EQ.1
       CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
       CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status,
       ierr)
END IF
```

Сообщение посланное каждым процессом будет скопировано до завершения операции передачи и начнется операция приема. Для завершения программы, необходимо, чтобы по крайней мере хотя бы одно или оба сообщения были буферизованы. Таким образом, эта программа может успешно выполниться, только если система связи может буферизовать не меньше count слов данных.

Когда используется стандартная передача, тогда тупиковые ситуации могут возникать, когда оба процесса заблокированы из-за недостатка буферного места. Тоже самое конечно случиться, если используется синхронный режим. Если используется буферизующий режим, и не достаточно буферного пространства, тогда программа также не будет завершена. Однако, мы скорее получим ошибку переполнения буфера, чем тупиковую ситуацию.

Программа надежна, если не требуется буферизации сообщений для завершения программы. Можно заменить все передачи в такой программе синхронными передачами и программа будет все еще работать корректно. Этот консервативный стиль программирования обеспечивает прекрасную совместимость, так как завершение программы не зависит от количества доступного буферного пространства и используемого протокола связи.

Многие программисты предпочитают использовать более надежные методы и используют "ненадежный" стиль программирования показанный

на третьем примере. В подобном случае использование стандартных передач вероятно обеспечивает наилучшее соотношение между производительностью и устойчивостью: качественные реализации будут обеспечивать достаточную буферизацию, чтобы "общая практика" программирования не была тупиковой. Буферизующий режим передачи может быть использован для программ, которые требуют много буферизации, или в ситуациях, когда программисту требуется больше управления. Этот режим также может быть использован для отладочных целей, так как переполнение буфера легче определить, чем тупиковые ситуации.

Неблокирующие операции передачи сообщений, как говориться в части 3.7, могут быть использованы для избежания необходимости буферизовать передаваемые сообщения. Это предотвращает тупики из-за недостатка буферного пространства, и увеличивает производительность, позволяя совмещать вычисления и связь, и избежать чрезмерного выделения буферов и копирования сообщений в буфер.

3.6 Выделение и использование буфера

Пользователь может выделить буфер для буферизации сообщений в буферизующем режиме. Буферизация производиться передатчиком.

```
MPI_BUFFER_ATTACH(buffer, size)
IN     buffer              начальный адрес буфера (choice)
IN     size                размер буфера в байтах (integer)

int MPI_Buffer_attach( void* buffer, int size)

MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)
       <type> BUFFER(*)
       INTEGER SIZE, IERROR
```

Обеспечивает для MPI буфер в пользовательской памяти, который будет использоваться для буферизации выходящих сообщений. Буфер используется только для сообщений посылаемых в буферизующем режиме. Только один буфер может быть присоединен к процессу одновременно.

```
MPI_BUFFER_DETACH( buffer, size)
OUT    buffer              начальный адрес буфера (choice)
OUT    size                размер буфера в байтах (integer)

int MPI_Buffer_detach( void* buffer, int* size)

MPI_BUFFER_DETACH( BUFFER, SIZE, IERROR)
       <type> BUFFER(*)
       INTEGER SIZE, IERROR
```

Отсоединяет буфер связанный с MPI. Эта операция будет заблокирована до тех пор, пока все сообщения буферизованные в буфере не будут переданы. По возвращении из этой функции пользователь может выделить или освободить место занимаемое буфером.

Поведение MPI, когда нет выделенного буфера такое же, как и при выделении буфера нулевой длины.

MPI должен обеспечить такую буферизацию сообщений, как если бы выходящие сообщения буферизировались передающим процессом, в указанном буферном пространстве, используя политику выделения циклическое непрерывное пространство. Мы набросаем ниже модель реализации, которая определяет эту политику. MPI может обеспечивать лучшую буферизацию, и может использовать лучшие алгоритмы выделения буфера, чем описанные ниже. Другими словами, MPI может обнаружить ошибку, всякий раз когда простое выделение буфера, описанное ниже, не будет иметь достаточно памяти. В особенности, если нет буфера определенно связанного с процессом, тогда любая буферная передача может вызвать ошибку.

MPI не обеспечивает механизмы для запросов и управления

буферизацией в стандартном режиме передачи. Он предполагает, что производители будут обеспечивать подобной информацией своих разработчиков.

Целесообразность. Существует широкий спектр возможных разработок в буферизации связи: буферизация может выполняться передатчиком, приемником, или обоими; буфера могут быть предназначены одной приемо-передающей паре, или разделяться всеми потоками; буферизация может происходить в реальной или виртуальной памяти; она может использовать предназначенную память, или память используемую другими процессами; буферное пространство может быть выделено статично или динамично изменяться; и.т.д. Не представляется возможным обеспечить переносимый механизм, для запроса или управления буферизацией, который будет совместим со всеми этими случаями, еще обеспечивая правдоподобную информацию. (Конец замечания.)

3.6.1 Модель реализации буферизующего режима

Модель реализации использующей упаковывающие и распаковывающие функции описана в части 3.13 и неблокирующие функции связи описаны в разделе 3.7.

Мы предполагаем, что хранится циклическая очередь ожидающих сообщений (pending message entries PME). Каждый элемент содержит коммуникационный запрос обработки, который определяет ожидающую неблокирующую передачу, указатель на следующий элемент и запакованное сообщение. Элементы хранятся последовательно в буфере. Свободное пространство возможно между концом очереди и ее началом.

Буферизующая передача реализуется следующим образом.

- Последовательное сканирование PME очереди назад от головы, удаление всех элементов для передач, которые уже закончились, до первого элемента с незавершенным запросом; перемещение головы очереди до этого элемента.

- Вычисление количества байт n необходимых, для хранения нового сообщения (длина упакованного сообщения вычисляется как MPI_PACK_SIZE + место для запроса обработки и указателя).

- Найти следующее свободное место размером n байт в буфере (место следующее за концом очереди, или место в начале буфера, если конец очереди слишком близок к концу буфера). Если место не найдено, тогда выставить ошибку переполнения буфера.

- Добавить в конец PME очереди новый элемент, который содержит запрос обработки, затем указатель и упакованное сообщение; MPI_PACK используется для упаковки данных.

- Выставить не блокирующую передачу (стандартный режим) для упакованных данных.

- Вернуться

3.7 Неблокирующие обмены

Можно увеличить производительность на многих системах, частично совмещая вычисления и передачу данных. Это особенно справедливо для систем, в которых обмены могут выполняться автоматически под управлением контроллера связи. Потоки с малыми весами - один из механизмов достижения подобного совмещения. Альтернативный механизм, который часто приводит к лучшей производительности это использование неблокирующих обменов. Неблокирующий вызов начала передачи начинает операцию передачи, но не завершает ее. Вызов начала передачи завершится до того, как сообщение будет скопировано в передающий буфер. Отдельный вызов завершения передачи необходим для завершения обмена, т.е. для проверки того, что данные были переданы из буфера. При

соответствующем оборудовании, передача данных из памяти передатчика может производиться одновременно с вычислениями. Также, неблокирующий вызов начала приема начинает операцию приема, но не завершает ее. Вызов возвратит управление до того, как сообщение будет занесено в приемный буфер. Отдельный вызов завершения приема необходим для завершения приема и проверки того, что данные получены приемным буфером. При соответствующем оборудовании, передача данных в память приемника может производиться одновременно с вычислениями. Использование неблокирующих приемов поможет также избежать системной буферизации и копирования память-память, информации накопленной ранее в приемный буфер.

Вызов начала неблокирующей передачи может использовать те же самые четыре режима, как и блокирующая передача: стандартный, буферизующий, синхронный и по готовности. Они имеют тоже самое значение. Передачи во всех режимах, кроме по готовности, могут быть начаты независимо от того, выставлен ли соответствующий прием или нет; блокирующая передача по готовности может начаться, только если соответствующий прием выставлен. Во всех случаях, начало передачи локально: оно заканчивается немедленно, независимо от состояния других процессов. Если вызов приводит к исчерпанию некоторых системных ресурсов, тогда он будет ошибочен и вернет код ошибки. Качественные реализации MPI должны гарантировать, что это произойдет только в "патологических" случаях. Т.е., реализация MPI должна поддерживать большое количество ожидающих неблокирующих операций.

Операция завершения передачи возвратит управление, когда данные будут скопированы из передающего буфера. Это может иметь дополнительное значение, зависящее от режима передачи.

Если режим передачи синхронный, тогда передача может завершиться, если начнется соответствующий прием. Т.е., прием будет выставлен, и будет соответствовать передаче - вызов завершения передачи не локальный. Заметим, что синхронная, неблокирующая передача может завершиться, если соответствующий неблокирующий прием, произошел перед вызовом завершения приема.(Он может завершиться как только передатчик узнает о завершении передачи, но до того, как приемник узнает о завершении передачи.)

Если режим передачи буферизующий, тогда сообщение должно быть буферизовано, если нет ожидающего приема - вызов завершения передачи локальный, и должен завершиться независимо от состояния соответствующего приема.

Если режим передачи стандартный, вызов завершения передачи может вернуть управление до появления соответствующего приема, если сообщение буферизовано. С другой стороны, передача может не завершиться пока ожидается соответствующий прием, и сообщение копируется в приемный буфер.

Неблокирующие передачи могут соответствовать блокирующим приемам, и наоборот.

> Совет пользователям. Завершение операции передачи может быть задержано, для стандартного режима, и должно быть задержано, для синхронного режима, пока не будет выставлен соответствующий прием. Использование неблокирующих передач в этих двух случаях позволяет передатчику быть впереди приемника, для того, чтобы вычисления были более терпимы к отклонениям скоростей двух процессов.

> Неблокирующая передача в буферизующем и по готовности режиме имеет более ограниченное влияние: неблокирующая передача будет заканчиваться так скоро, как только возможно, тогда как блокирующая передача будет завершаться после передачи данных из памяти передатчика. Использование неблокирующих передач выгодней в тех случаях, только если копирование будет происходить одновременно с вычислениями.

> Модель передачи сообщений влияет на обмены начатые передатчиком. Связь будет иметь более низкий уровень накладных расходов, если прием уже выставлен, когда передатчик

инициирует связь (данные могу поступать прямо в приемный буфер, и нет необходимости выстраивать в очередь ожидающие запросы передач). Однако, операция приема может завершиться только после прихода соответствующей передачи. Использование неблокирующих приемов позволяет уменьшить накладные расходы обменов не блокируя принимающий процесс, для ожидания передачи. (Конец совета пользователям.)

3.7.1 Объекты связи

Неблокирующие обмены используют непрозрачный запрос объектов для идентификации операции связи и соответствия операции, которая инициировала связь и операции, которая завершила ее. Это системные объекты, которые доступны через хэндлеры. Объект запроса идентифицирует различные свойства операции связи, такие как режим передачи, буфер связи связанный с ней, ее контекст, аргументы признака и назначения используемые для передачи, или аргументы признака и источника используемые для приема. Добавим, этот объект хранит информацию о состоянии ожидающих операций связи, которые производятся с этим объектом.

3.7.2 Инициация связи

Мы используем тоже самое соглашение имен как и в блокирующей связи: префикс B, S, или R используется для буферизующего, синхронного или режима по готовности.

Добавим префикс I (для немедленного) означающий, что вызов неблокирующий.

Начало неблокирующей передачи в стандартном режиме.

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
IN     buf           адрес начала буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип каждого элемента буфера передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    request       запрос связи (handle)

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

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Начало неблокирующей передачи в буферизующем режиме.

```
MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)
IN     buf           адрес начала буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип каждого элемента буфера передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    request       запрос связи (handle)

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

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Начало неблокирующей передачи в синхронном режиме.

```
MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)
```

```
IN     buf           адрес начала буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип каждого элемента буфера передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    request       запрос связи (handle)

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

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Начало неблокирующей передачи в режиме по готовности.

```
MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)
IN     buf           адрес начала буфера передачи (choice)
IN     count         количество элементов в буфере передачи (integer)
IN     datatype      тип каждого элемента буфера передачи (handle)
IN     dest          ранк приемника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    request       запрос связи (handle)

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

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Начало неблокирующего приема.

```
MPI_IRECV(buf, count, datatype, source, tag, comm, request)
OUT    buf           адрес начала буфера приема(choice)
IN     count         количество элементов в буфере приема (integer)
IN     datatype      тип каждого элемента буфера приема (handle)
IN     source        ранк источника (integer)
IN     tag           признак сообщения (integer)
IN     comm          коммуникатор (handle)
OUT    request       запрос связи (handle)

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

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

Эти вызовы выделяют запрос объекта связи и связывают его с хэндлером. Запрос может использоваться позже, для опроса состояния связи или для ожидания ее завершения.

Вызов неблокирующей передачи показывает, что система может начать выдачу данных из буфера передатчика. Передатчик не должен обращаться к буферу после вызова операции неблокирующей передачи, до тех пор пока передача не завершиться.

Неблокирующий прием означает, что система может начать записывать данные в приемный буфер. Приемник не должен обращаться к буферу после начала неблокирующего приема, до его завершения.

3.7.3 Завершение обмена

Функции MPI_WAIT и MPI_TEST используются для завершения неблокирующей связи. Завершение операции передачи показывает, что передатчик теперь может обновить буфер передач (сама операция

передачи оставляет содержимое буфера без изменений). Это не означает, что сообщение было принято; вернее, оно может быть буферизовано подсистемой связи. Однако, если был использован синхронный режим, завершение операции передачи означает, что был запущен соответствующий прием, и что сообщение будет возможно получено этим соответствующим приемом.

Завершение операции приема показывает, что буфер приемника содержит принятое сообщение, и что объект состояния установлен; приемник теперь может свободно обращаться к своему буферу. Это не означает, что соответствующая операция передачи завершилась (но означает, конечно, что передача была начата).

Мы будем использовать следующую терминологию. Нулевой хэндлер - это хэндлер со значением MPI_REQUEST_NULL. Постоянный запрос и его хэндлер - неактивный, если запрос не связан с текущей связью - смотри часть 3.9. Хэндлер активный если он ни нулевой, ни неактивный.

MPI_WAIT(request, status)

INOUT	request	запрос (handle)
OUT	status	объект статуса (Status)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Вызов MPI_WAIT возвращает управление, когда операция передачи определяемая request завершается. Если объект связи ассоциированный с этим запросом был создан не блокирующим приемом или передачей, тогда объект высвобождается вызовом MPI_WAIT и хэндлер запроса устанавливается в MPI_REQUEST_NULL. MPI_WAIT не локальная операция.

Вызов возвращает информацию о завершенной операции в status. Содержимое объекта состояния может быть получено, как описано в разделе 3.2.5. Объект состояния для операции передачи может быть запрошен вызовом MPI_TEST_CANCELLED (часть 3.8, ниже).

Допустимо вызвать MPI_WAIT с нулевым или неактивным аргументом request. В таком случае операция завершается немедленно; аргумент status возвращает tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, и таким образом, что вызовы MPI_GET_ELEMENTS и MPI_GET_COUNT возвращают count = 0.

Целесообразность. Это делает MPI_WAIT функционально эквивалентной MPI_WAITALL со списком длиной один и некоторой элегантностью. Состояние устанавливается так, чтобы ликвидировать ошибки через доступ к устаревшей информации состояния. (Конец замечания.)

Совет разработчикам. В мульти-потоковой среде, вызов MPI_WAIT должен блокировать только вызываемый поток, позволяя планировщику потоков блокировать этот поток и начать выполнение другого потока. (Конец совета разработчикам.)

MPI_TEST(request, flag, status)

INOUT	request	запрос связи (handle)
OUT	flag	true если операция завершена (logical)
OUT	status	объект состояния (Status)

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

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Вызов MPI_TEST возвращает flag = true если операция определяемая request завершилась. В этом случае, объект состояния содержит информацию о завершенной операции; если объект связи был создан неблокирующим приемом или передачей, тогда он высвобождается

и хэндлер запроса устанавливается в MPI_REQUEST_NULL. В противном случае вызов возвращает flag = false. В этом случае, значение объекта состояния не определено. MPI_TEST - это локальная операция.

Возвращаемый объект состояния для операции приема содержит информацию, получение которой описано в разделе 3.2.5. Объект состояния для операции передачи содержит информацию, для получения которой необходимо вызвать функцию MPI_TEST_CANCELLED (часть 3.8, ниже).

Можно вызывать MPI_TEST с нулевым или неактивным аргументом request. В этом случае операция возвращает flag = false.

Функции MPI_WAIT и MPI_TEST могут быть использованы для завершения приемов и передач.

Совет пользователям. Использование неблокирующего вызова MPI_TEST позволяет пользователю предусматривать альтернативные действия с единственным потоком выполнения; событийно-управляемый планировщик потоков может быть эмулирован периодическими вызовами MPI_TEST.(Конец совета пользователям.)

Пример:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** некоторые вычисления, чтобы скрыть ожидание ****
    CALL MPI_WAIT(request, status, ierr)
  ELSE
    CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** некоторые вычисления, чтобы скрыть ожидание ****
    CALL MPI_WAIT(request, status, ierr)
END IF
```

Запрошенный объект может быть высвобожден без ожидания завершения соответствующей связи, используя следующую операцию.

```
MPI_REQUEST_FREE(request)
INOUT         request       запрос связи (handle)

int MPI_Request(MPI_Request *request)

MPI_REQUEST_FREE(REQUEST, IERROR)
       INTEGER REQUEST, IERROR
```

Помечает объект запроса для освобождения и устанавливает request в MPI_REQUEST_NULL. С выполняющейся связью происходит так, что запрос выполняется успешно; запрос будет освобожден после завершения.

Совет пользователям. Если запрос освобожден вызовом MPI_REQUEST_FREE, тогда невозможно больше проверить успешность завершения связи вызовами MPI_WAIT или MPI_TEST. Также, если ошибка произошла впоследствии во-время связи, код ошибки не будет возвращен пользователю - подобная ошибка должна трактоваться как фатальная. Активный запрос передачи должен быть освобожден только, когда логика программы такова, что приемник посылает ответ передатчику сообщения; прибывший ответ информирует передатчик, что передача завершена и буфер передатчика может использоваться снова. Активный запрос приема никогда не будет освобожден, так как приемник не имеет способа проверить, что прием завершился и приемный буфер может быть снова использован. (Конец совета пользователям.)

Пример:

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank)
if(rank.EQ.0)
```

```
  THEN
    DO i=1, n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0 ,req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
  ELSE          ! rank.EQ.1
      CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
      CALL MPI_WAIT(req, status)
    DO i=1, n-1
      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0 ,req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
    END DO
      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, req, ierr)
      CALL MPI_WAIT(req, status)
END IF
```

3.7.4 Семантика неблокирующей связи

Семантика неблокирующей связи определена подходящим продлением определений в части 3.5.

Порядок Операции неблокирующей связи задаются в соответствии с порядком выполнения вызовов, которые инициируют связь. Требования неперекрытия части 3.5 применимы к неблокирующей связи, согласно этому определению используется следующий порядок.

Пример:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (RANK.EQ.0)
  THEN
    CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
    CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
  ELSE          ! rank.EQ.1
    CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
    CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(h1, status)
CALL MPI_WAIT(h2, status)
```

Первый передача процесса ноль будет соответствовать первому приёму процесса один, даже, если оба сообщения посланы до выполнения первым процессом любой операции приема.

Развитие Вызов MPI_WAIT, который завершает прием будет возможно закончен и возвратит управление если началась соответствующая передача, или если передача удовлетворилась другим приемом. В особенности, если соответствующая передача неблокирующая, тогда прием должен завершиться даже если не выполняется вызов передатчика по завершению передачи. Аналогично, вызов MPI_WAIT, который завершает передачу будет возможно закончен, если начался соответствующий прием, если прием не удовлетворен другой передачей, и даже если нет вызовов завершающих выполнение приема.

Пример:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0)
  THEN
    CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
    CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
  ELSE          ! rank.EQ.1
    CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
```

```
    CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, ierr)
    CALL MPI_WAIT(r, status, ierr)
END IF
```

Эта программа не должна иметь тупиков в корректной реализации MPI. Первая синхронная передача процесса ноль после того, как процесс один выставит соответствующий (неблокирующий) прием, даже если процесс один не достиг еще завершающего ожидания вызова. Таким образом, процесс ноль будет продолжаться и выполнит вторую передачу, позволяющую процессу один завершить выполнение.

Если MPI_TEST, который завершает прием, повторно вызвать с теми же самыми аргументами, и начнется соответствующая передача, тогда вызов возможно вернет flag = true, если передача не удовлетворится другим приемом. Если MPI_TEST, который завершает прием, повторно вызвать с теми же самыми аргументами, и начнется соответствующий прием, тогда вызов возможно вернет flag = true, если прием не удовлетворится другой передачей.

3.7.5 Завершение множества

Удобно иметь возможность ожидать завершения любого, некоторого, или всех операций из списка, а не ожидать конкретное сообщение. Вызов MPI_WAITANY или MPI_TESTANY может быть использован для ожидания завершения одной из нескольких операций; вызов MPI_WAITALL или MPI_TESTALL может быть использован для ожидания всех ожидающих операций из списка; вызов MPI_WAITSOME или MPI_TESTSOME может быть использован для завершения всех возможных операций из списка.

```
MPI_WAITANY(count, array_of_requests, index, status)
IN            count                    длина списка (integer)
INOUT         array_of_requests        массив запросов (array of handles)
OUT           index                    индекс управления для завершающейся
                                       операции (integer)
OUT           status                   объект статуса (Status)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int
*index, MPI_Status *status)

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
      INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
STATUS(MPI_STATUS_SIZE), IERROR
```

Блокировка продолжается до тех пор, пока не завершится хотя бы одна из операций ассоциированных с активным запросом из списка. Если более, чем одна операция может быть разрешена и прервана, то выбирается произвольная. Возвращает в index индекс этого запроса в массиве и в status статус завершающейся операции. (Массив индексируется с нуля в Си, и с единицы в Fortran'е.) Если запрос был выделен неблокирующей операцией связи, тогда он высвобождается и управление запросом устанавливается в MPI_REQUEST_NULL.

Список array_of_requests может содержать нулевые или неактивные хэндлеры. Если список не содержит активных хэндлеров (список имеет нулевую длину или все вхождения нулевые или неактивны), тогда вызов немедленно завершается с index = MPI_UNDEFINED.

Выполнение MPI_WAITANY(count, array_of_requests, index, status) имеет тот же самый эффект как и выполнение MPI_WAIT(&array_of_requests[i], status), где ioe значение возвращенное index'ом.

MPI_WAITANY с массивом содержащим один активный элемент эквивалентен MPI_WAIT.

```
MPI_TESTANY(count, array_of_requests, index, flag, status)
IN            count                    длина списка (integer)
INOUT         array_of_requests        массив запросов (array of handles)
OUT           index                    индекс завершенной операции, или
```

		MPI_UNDEFINED если не завершилась (integer)
OUT	flag	true если одна операция завершилась (logical)
OUT	status	объект состояния (Status)

```
int MPI_Testany(int count, MPI_Request *array_of_requests, int
*index, int *flag, MPI_Status *status)

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
       LOGICAL FLAG
       INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
STATUS(MPI_STATUS_SIZE), IERROR
```

Проверяет завершилась ли какая-нибудь из операций связанных с активными хэндлерами. В первом случае, он вернет flag = true в index'е индекс этого запроса в массиве, и вернет в status'е статус этой операции; если запрос был выделен неблокирующим вызовом связи, тогда запрос высвобождается и хэндлер устанавливается в MPI_REQUEST_NULL. (Массив индексируется с нуля в Си, и с единицы в Fortran'е.) В последнем случае, он возвращает flag = false и значение MPI_UNDEFINED в index'е, и status также не определен. Массив может содержать нулевые или неактивные хэндлеры. Если массив не содержит активных хэндлеров, тогда вызов немедленно завершается с flag = false, index = MPIUNDEFINED, и status не определен.

Выполнение MPI_TESTANY(count, array_of_requests, index, status) имеет тот же самый эффект как и выполнение MPI_TEST(&array_of_requests[i], status), для i = 0,1,...,count-1, в том же произвольном порядке, до тех пор, пока хотя бы один вызов не вернет flag = true, или все неудачно завершаться. В первом случае, index принимает последнее значение i; в последнем случае он устанавливается в значение MPI_UNDEFINED.

MPI_TESTANY с массивом содержащим один активный элемент эквивалентен MPI_TEST.

```
MPI_WAITALL(count, array_of_requests, array_of_statuses)
```

IN	count	длина списка (integer)
INOUT	array_of_requests	массив запросов (array of handles)
OUT	array_of_statuses	массив состояний объектов (array of Status)

```
int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
       INTEGER COUNT, ARRAY_OF_REQUESTS(*),
       ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

Блокировка продолжается до тех пор, пока не закончится хотя бы одна операция связи ассоциированная с активным хэндлером из списка, и возвращает состояние всех этих операций (это включает случай, когда нет активных хэндлеров в списке). Оба массива имеют те же самые номера элементов. I-ый элемент массива array_of_statuses содержит состояние i-ой операции. Запросы которые были созданы неблокирующими операциями связи высвобождаются и соответствующие хэндлеры в массиве устанавливаются в MPI_REQUEST_NULL. Список может содержать нулевые и неактивные хэндлеры. Вызов возвращает в состоянии каждого такого элемента tag = MPI_ANY_TAG source = MPI_ANY_SOURCE, и так, что вызовы MPI_GET_COUNT и MPI_GET_ELEMENTS возвращают count = 0.

Выполнение MPI_WAITALL(count, array_of_requests, array_of_statuses) имеет тот же самый эффект как и выполнение MPI_WAIT(&array_of_requests[i], &array_of_statuses[i]), для i = 0, 1,...,count-1, в том же произвольном порядке.

MPI_WAITALL с массивом длиной один эквивалентен MPI_WAIT.

```
MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)
IN          count                   длина списка (integer)
INOUT       array_of_requests       массив запросов (array of handles)
OUT         flag                    (logical)
OUT         array_of_statuses       массив состояний объектов (array of
                                    Status)

int MPI_Testall(int count, MPI_Request *array_of_requests, int
*flag, MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES,
IERROR)
       LOGICAL FLAG
       INTEGER COUNT, ARRAY_OF_REQUESTS(*),
       ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

Возвращает flag = true, если все обмены, связанные с активными хэндлерами в массиве, завершены (это включает случай когда в списке нет активных хэндлеров). В этом случае, каждый элемент статуса, который соответствует активному хэндлеру запроса устанавливается в состояние соответствующей связи; если запрос был выделен неблокирующим вызовом связи, тогда он высвобождается, и хэндлер устанавливается в MPI_REQUEST_NULL. Каждый элемент статуса соответствующий нулевому или неактивному хэндлеру возвращает tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, и так, что вызовы MPI_GET_COUNT и MPI_GET_ELEMENTS возвращают count = 0.

В противном случае, возвращается flag = false, запросы не модифицируются и значения элементов статуса не изменяются. Эта операция локальная.

```
MPI_WAITSOME(incount, array_of_requests, outcount,
array_of_indices, array_of_statuses)
IN          incount                 длина списка (integer)
INOUT       array_of_requests       массив запросов (array of handles)
OUT         outcount                количество завершенных запросов
                                    (integer)
OUT         array_of_indices        массив индексов завершенных операций
                                    (array of integers)
OUT         array_of_statuses       массив состояний объектов для
                                    операций, которые завершились (array
                                    of Status)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int
*outcount, int *array_of_indices, MPI_Status *array_of_statuses)

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)
       INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
ARRAY_OF_INDICES(*), ARRAY_OFSTATUSES(MPI_STATUS_SIZE,*), IERROR
```

Ожидает пока завершится хотя бы одна операция связанная с активным хэндлером из списка. Возвращает в outcount количество завершенных запросов из списка array_of_request. Возвращает в первых outcount элементах массива array_of_indices индексы этих операций (индекс в массиве array_of_request; массив индексируется с нуля в Си и с единицы в Fortran'е). Возвращает в первых outcount элементах массива array_of_status состояние для этих завершенных операций. Если завершенный запрос был выделен неблокирующим вызовом связи, тогда он высвобождается, и связанному с ним хэндлеру присваивается значение MPI_REQUEST_NULL.

Если список не содержит активных хэндлеров, то вызов завершается немедленно с outcount = 0.

```
MPI_TESTSOME(incount, array_of_requests, outcount,
array_of_indices, array_of_statuses)
IN          incount                 длина array_of_requests (integer)
INOUT       array_of_requests       массив запросов (array of handles)
```

OUT	outcount	количество завершенных запросов (integer)
OUT	array_of_indices	массив индексов завершенных операций (array of integers)
OUT	array_of_statuses	массив состояний объектов для операций, которые завершились (array of Status)

```
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int
*outcount, int *array_of_indices, MPI_Status *array_of_statuses)

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT,
ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)
       INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
ARRAY_OF_INDICES(*), ARRAY_OFSTATUSES(MPI_STATUS_SIZE,*), IERROR
```

Ведет себя как MPI_WAITSOME, кроме того, что возвращает управление немедленно, даже если нет завершенной операции; в этом случае, она возвращает outcount = 0.

MPI_TESTSOME это локальная операция, которая немедленно завершается, тогда как MPI_WAITSOME будет заблокирована до завершения связи, если ей был передан список, который содержит хотя бы один активный хэндлер. Оба вызова выполняют fairness соглашение: если запрос на прием повторно появляется в списке запросов переданных MPI_WAITSOME или MPI_TESTSOME, и соответствующая передача выставлена, тогда прием возможно будет удачен, если передача не удовлетвориться другим приемом; и аналогично для запросов передач.

> Совет пользователям. Использование MPI_TESTSOME, вероятно, будет более эффективен, чем использование MPI_TESTANY: первый возвращает информацию обо всех завершенных связях; для последнего, требуется новый вызов для каждого завершенного обмена.
>
> Для обслуживания множества клиентов можно использовать MPI_WAITSOME, чтобы не "обидеть" ни одного клиента. Клиенты посылают сообщения серверу с запросами обслуживания; сервер вызывает MPI_WAITSOME с одним запросом приема для каждого клиента и затем управляет всеми завершенными передачами. Если вместо этого будет использован вызов MPI_WAITANY, то один клиент может зависнуть, пока запросы от другого клиента всегда будут проходить первыми. (Конец совета пользователям.)
>
> Совет разработчикам. MPI_TESTSOME должна завершать столько ожидающих коммуникаций сколько возможно. (Конец совета разработчикам.)

Пример 1: клиент-серверный код (может произойти зависание)

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
if(rank > 0)
  THEN          ! код клиента
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
  ELSE          ! rank=0 -- код сервера
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, 0, tag,
                      comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
       CALL DO_SERVICE(a(1, index))     ! хэндлер одного сообщения
       CALL MPI_IRECV(a(1, index), n, MPI_REAL, 0, tag,
```

```
                    comm, request_list(index), ierr)
    END DO
END IF
```

Пример 2: тот же код с использованием MPI_WAITSOME.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
if(rank > 0)
  THEN          ! код клиента
    DO WHILE(.TRUE.)
       CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
       CALL MPI_WAIT(request, status, ierr)
    END DO
  ELSE          ! rank=0 -- код сервера
    DO i=1, size-1
       CALL MPI_IRECV(a(1,i), n, MPI_REAL, 0, tag,
                    comm, request_list(i), ierr)
    END DO
    DO WHILE(.TRUE.)
       CALL MPI_WAITSOME(size, request_list, numdone,
                         index_list, status_list, ierr)
       DO i=1, numdone
          CALL DO_SERVICE(a(1, index_list(i)))
          CALL MPI_IRECV(a(1, index_list(i)), n, MPI_REAL, 0, tag,
                    comm, request_list(i), ierr)
       END DO
    END DO
END IF
```

3.8 Проверка и отказ

Операции MPI_PROBE и MPI_IPROBE позволяют проверять приходящие сообщения без фактического их получения. Пользователь может решить как получить их, основываясь на информации полученной в результате выполнения MPI_PROBE (основная информация возвращается через status). В частности, пользователь может разместить память для приемного буфера, в соответствии с длиной просканированного сообщения.

Операция MPI_CANCEL позволяет отказываться от установленной связи. Это требуется для очистки (памяти). Передача сообщения или прием связывает ресурсы пользователя (передающий или приемный буфера), и отказ может быть необходим для освобождения этих ресурсов.

```
MPI_IPROBE(source,tag,comm,flag,status)
IN     source        ранк источника, или MPI_ANY_SOURCE (integer)
IN     tag           значение тэга или MPI_ANY_TAG (integer)
IN     comm          коммуникатор (handle)
OUT    flag          (logical)
OUT    status        объект статуса (Status)
```

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
               MPI_Status *status)
MPI_IPROBE(SOURCE, TAG, FLAG, STATUS, IERROR)
       LOGICAL FLAG
       INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_IPROBE(source, tag, comm, flag, status) возвращает значение flag = true, если есть сообщение, которое может быть получено и которое отвечает образцу, специфицируемому аргументами source, tag и comm. Вызов идентифицирует сообщение, которое должно быть получено путем вызова MPI_RECV(..., source, tag, comm, status), выполняемого в соответствующей точке программы, и возвращает в status значение, которое было бы получено вызовом MPI_RECV(). В противном случае вызов возвращает flag = false, и

оставляет status неопределенным.

Если MPI_IPROBE возвращает flag = true, то к содержимому объекта статуса можно получить доступ, как описано в разделе 3.2.5, чтобы найти источник, тэг и длину сканируемого сообщения.

Следующий приём, выполняемый с этим контекстом, источником и тэгом, возвращаемым в status путем выполнения MPI_IPROBE, получает сообщение, которое было проверено при сканировании, если никаких сообщений после этого не было получено. Если процесс приема разветвлен, тогда на пользователя ложится ответственность за сохранение предыдущих условий.

Аргумент source операции MPI_PROBE может быть MPI_ANY_SOURCE, и аргумент tag может быть MPI_ANY_TAG, так что можно сканировать сообщения от произвольного источника и/или с произвольным тэгом. Тем не менее, специфичный контекст связи должен задаваться аргументом comm.

Необязательно получать сообщение немедленно после того, как оно было просканировано, до получения оно может быть просканировано несколько раз.

```
MPI_PROBE(source, tag, comm, status)
IN     source               тип источника, или MPI_SOURCE (integer)
IN     tag                  значение тэга, или MPI_ANY_TAG (integer)
IN     comm                 коммуникатор (handle)
OUT    status               объект статуса (Status)

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

MPI_PROBE(SOURCE, TAG, COMM, IERROR)
       INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_PROBE ведет себя так, же как и MPI_IPROBE, за исключением того, что это блокирующий вызов, который возвращает управление только после того, как было найдено подходящее сообщение. MPI-реализация операций MPI_PROBE и MPI_IPROBE предполагает гарантированное развитие: если вызов MPI_PROBE выполнен процессом, и передача, которая соответствует сканированию, была начата некоторым процессом, тогда вызов MPI_PROBE вернёт управление, если сообщение не было получено другой параллельной операцией приёма (которая выполняется другой ветвью сканирующего процесса). Подобно этому, если процесс ожидает MPI_IPROBE и было найдено подходящее сообщение, то вызов MPI_IPROBE всегда будет возвращать flag = true, если сообщение не было получено другой операцией приёма.

Пример 1:

```
CALL MPI_COMM_RANK(comm, rank, ierr)
if (rank.EQ.0)
  THEN
    CALL MPI_SEND(i, 1, MPI-INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE  ! rank = 2
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE).EQ.0)
      THEN
100     CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, status, ierr)
      ELSE
200     CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, status, ierr)
    END IF
  END DO
END IF
```

Каждое сообщение получается с правильным типом.

Пример 2.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
       CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF(rank.EQ.1) THEN
       CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE
  DO i=1,2
       CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
       IF (status(MPI_SOURCE).EQ.0) THEN
100           CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, 0,
              status, ierr)
       ELSE
200           CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, status,
              ierr)
       END IF
  END DO
END IF
```

Мы небрежно модифицировали пример 1, используя MPI_ANY_SOURCE в качестве аргумента в двух вызовах приёма в операторах с метками 100 и 200. Программа сейчас некорректна: операция приёма может принимать сообщение, которое отличается от сообщения, сканированного предыдущим вызовом MPI_PROBE.

Совет разработчикам. Вызов MPI_PROBE(source,tag,comm,status) будет идентифицировать сообщение, которое должно было быть получено вызовом MPI_RECV(..., source, tag, comm, status), выполняемым в том же месте. Предположим, что это сообщение имеет источник s, тэг t и контекст c. Если аргумент тэга в вызове сканирования имеет значение MPI_ANY_TAG, тогда просканированное сообщение будет самым ранним ожидаемым сообщением с источником s, контекстом c и любым тэгом; в любом случае, это сообщение будет самым ранним среди сообщений с источником s, тэгом t и контекстом c (сообщение, которое должно было быть получено, так чтобы соблюсти порядок сообщения). Это сообщение будет оставаться самым ранним ожидаемым сообщением с источником s, с тэгом t и контекстом c, пока не будет получено. Операция получения, следующая за сканированием, которая использует такой же контекст как в сканировании, и использует значения тэга и источника, полученные через сканирование, должна получить сообщение, если оно уже не было получено другой операцией приёма. (Конец совета разработчикам).

```
MPI_CANCEL(request)
IN     request              запрос связи (handle)

int MPI_Cancel(MPI_Request *request)

MPI_CANCEL(REQUEST, IERROR)
       INTEGER REQUEST, IERROR
```

Вызов MPI_CANCEL помечает для отказа отложенную неблокирующую операцию (передачи или приёма). Вызов отказа - локальный. Он возвращает управление немедленно, возможно даже до того как обмен фактически отменен. Ещё необходимо завершить обмен, который был помечен для отмены, используя вызов MPI_WAIT или MPI_TEST (или любые другие производные операции).

Если обмен помечен для отмены, тогда вызов MPI_WAIT для этого обмена гарантированно возвратит управление, независимо от деятельности других процессов (т.е. MPI_WAIT ведет себя как локальная функция); подобно этому, если MPI_TEST многократно вызывается в цикле активного ожидания для отмененной связи, тогда MPI_TEST постоянно будет успешным.

MPI_CANCEL может быть использована для отмены связи, которая

использует настойчивый запрос, аналогично она может быть использована для отмены связи с ненастойчивым запросом. Успешная отмена отменяет активную связь, но не собственно запрос. После вызова MPI_CANCEL и последующего вызова MPI_WAIT или MPI_TEST, запрос становится неактивным и может быть активизирован для новой связи.

Успешная отмена буферизирующего обмена освобождает буферное пространство, выделенное для ожидаемого сообщения.

Либо происходит отказ, либо связь, но не то и другое вместе. Если передача помечена для отмены, тогда возможен случай, когда либо передача завершится нормально, и переданное сообщение будет принято принимающим процессом, либо передача нормально отменена, в этом случае никакая часть сообщения не будет получена приемником. Тогда, любой подходящий прием должен быть удовлетворен другой передачей. Если прием помечен для отмены, тогда должен быть случай, когда прием осуществится нормально или, когда прием будет успешно отменен, в этом случае никакая часть приемного буфера не изменится. Тогда, любая подходящая передача удовлетворится другим приемом. Если операция была отменена, тогда информация о произведенном действии будет возвращена в аргументе status той операции, которая выполняет связь.

```
MPI_TEST_CANCELLED(status,flag)
IN     status               объект статуса (Status)
OUT    flag                 (logical)

int MPI_Test_cancelled(MPI_Status status, int *flag)

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
       LOGICAL FLAG
       INTEGER STATUS(MPI_STATUS_SIZE), IERROR
```

Возвращает flag = true, если обмен, связанный с объектом статуса, был успешно отменён. В этом случае, все другие поля статуса (такие как count и tag) не определены. В противном случае возвращается flag = false. Если операция приёма могла быть отменена, тогда следовало вначале вызвать MPI_TEST_CANCELLED, чтобы проверить была ли операция отменена, перед проверкой других полей возвращенного статуса.

> Совет пользователям. Операция отказа может иметь далеко идущие последствия, поэтому пользоваться ею следует лишь в исключительных случаях. (Конец совета пользователям).

> Совет разработчикам. Если операция передачи использует протокол "eager" (данные передаются в приемник перед тем, как стало ясно, что они подходят), тогда отмена этой передачи может потребовать связи с назначенным приемником, чтобы освободить размещенные буфера. В некоторых системах может потребоваться прерывание для назначенного приемника. Заметьте, что хотя обмены еще нужны для реализации MPI_CANCEL, это локальная операция, в том смысле, что ее выполнение не зависит от программ, выполняемых другими процессами. Если требуется обработка в другом процессе, то она должна быть прозрачной для прикладной программы (следовательно есть необходимость в прерывании и обработчике прерываний). (Конец совета разработчикам.)

3.9 Запросы настойчивой связи

Часто связь с соответствующим списком аргументов многократно выполняется во внутреннем кольце параллельных вычислений. В такой ситуации, может оказаться возможным оптимизировать передачу связыванием списка аргументов передачи с настойчивым запросом связи один раз, затем повторно используя запрос для инициации и передачи сообщений. Настойчивый запрос, созданный таким образом, можно считать как порт связи или "полуканал". Он не будет обеспечивать

полной функциональности обычного канала, так как здесь нет связи передающего порта с приемным портом: такая конструкция позволяет снизить затраты на связь между процессором и контроллером связи, но не затраты на связь между одним контроллером связи и другим. Необязательно, чтобы сообщения, передаваемые с настойчивым запросом, принимались операцией приёма, использующей настойчивый запрос, и наоборот.

Настойчивый запрос связи создается, используя один из четырех следующих вызовов. Эти вызовы не используют обмены.

```
MPI_SEND_INT(buf, count, datatype, dest, tag, comm, request)
IN     buf                  адрес начала передаваемого буфера (choice)
IN     count                количество посылаемых элементов (integer)
IN     datatype             тип каждого элемента (handle)
IN     dest                 ранк приемника (integer)
IN     tag                  тэг сообщения (integer)
IN     comm                 коммуникатор (handle)
OUT    request              запрос связи (handle)

int MPI_Send_init(void* buf, int count, MPI_Datatype datatype,
              intdest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
IERROR)
       <type> BUF(*)
       INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
       IERROR
```

Создает запрос настойчивой связи для операции передачи стандартного режима, и связывает с ним все аргументы операции передачи.

```
MPI_BSEND_INIT(buf,count,datatype,dest,comm,request)
IN     buf                  адрес начала передаваемого буфера (choice)
IN     count                количество посылаемых элементов (integer)
IN     datatype             тип каждого элемента (handlr)
IN     dest                 ранк приемника (integer)
IN     tag                  тэг сообщения (integer)
IN     comm                 коммуникатор (handle)
OUT    request              запрос связи (handle)

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype,
              intdest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
IERROR)
       <type> BUF(*)
       INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
              IERROR
```

Создает настойчивый запрос связи для передачи буферизующего режима.

```
MPI_SSEND_INIT(buf,count,datatype,dest,tag,comm,request)
IN     buf                  адрес начала передаваемого буфера (choice)
IN     count                количество посылаемых элементов (integer)
IN     datatype             тип каждого элемента (handle)
IN     dest                 ранк приемника (integer)
IN     tag                  тэг сообщения (integer)
IN     comm                 коммуникатор (handle)
OUT    request              запрос связи (handle)

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

MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
IERROR)
```

```
       <type> BUF(*)
       INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
       IERROR
```

Создает объект настойчивой связи для операции передачи синхронного режима.

```
MPI_RSEND_INIT(buf,count,datatype,dest,tag,comm,request)
IN     buf                 адрес начала передаваемого буфера (choice)
IN     count               количество посылаемых элементов (integer)
IN     datatype            тип каждого элемента (handle)
IN     dest                ранк приемника (integer)
IN     tag                 тэг сообщения (integer)
IN     comm                коммуникатор (handle)
OUT    request             запрос связи (handle)

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

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Создает объект настойчивой связи для операции передачи в режиме по готовности.

```
MPI_RECV_INIT(buf,count,datatype,source,tag,comm,request)
IN     buf                 адрес начала передаваемого буфера (choice)
IN     count               количество посылаемых элементов (integer)
IN     datatype            тип каждого элемента (handle)
IN     dest                ранк приемника или MPI_ANY_SOURCE(integer)
IN     tag                 тэг сообщения или MPI_ANY_TAG (integer)
IN     comm                коммуникатор (handle)
OUT    request             запрос обмена (handle)

MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Request request)

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST)
       <type>BUF(*)
       INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST
```

Создаёт настойчивый запрос связи для операции приёма.
Настойчивый запрос на обмен неактивен после создания - активные обмены не присоединяются к запросу.
Обмены приём передача, которые используют настойчивый запрос инициируются функцией MPI_START.

```
MPI_START(request)
INOUT         request              запрос на обмен (handle)

int MPI_Start(MPI_Request *request)

MPI_START(REQUEST, IERROR)
       INTEGER REQUEST, IERROR
```

request - операнд, возвращаемый одним из предыдущих четырех вызовов. Связанный запрос является неактивным. Он становится активным, после выполнения вызова.
Если есть запрос для передачи с режимом по готовности, тогда соответствующий запрос на приём должен быть выставлен, перед тем как сделан вызов. Буфер связи должен быть доступен и после вызова, и в то время как выполняется операция.
Вызов локален, с семантикой схожей для операций неблокированной связи, описанных в разделе 3.7.5. То есть, вызов MPI_START с запросом, созданным MPI_SEND_INIT, начинает обмен таким

же точно образом, как вызов MPI_ISEND; вызов MPI_START с запросом, созданным MPI_BSEND_INIT начинает обмен таким же точно образом как вызов MPI_IBSEND; и так далее.

MPI_STARTALL(count, array_of_requests)

IN	count	длина списка (integer)
INOUT	array_of_request	массив запросов (array of handle)

```
int MPI_Startall(int count, MPI_Request *array_of_request)

MPI_STARTALL(COUNT, ARRAY_OF_REQUEST, IERROR)
       INTEGER COUNT, ARRAY_OF_REQUEST(*), IERROR
```

Начинает все обмены, объединенные с запросами из array_of_request. Вызов MPI_STARTALL(count, array_of_requests) имеет тот же эффект, что и вызовы MPI_START(&array_of_request[i]), выполненные для i=0, .., count-1, в некотором произвольном порядке.

Обмен, начатый вызовом MPI_START или MPI_STARTALL, завершается при помощи вызова MPI_WAIT, MPI_TEST, или одной из производных функций, описанных в разделе 3.7.5. Запрос становится неактивным после успешного выполнения такого вызова. Этот запрос не удаляется и может быть активизирован вновь путем вызова MPI_START или MPI_STARTALL.

Настойчивый запрос удаляется вызовом MPI_REQUEST_FREE (раздел 3.7.3).

Вызов MPI_REQUEST_FREE можно производить в любой точке программы, после того, как создан настойчивый запрос. Тем не менее, запрос должен удаляться только после того, как станет неактивным. Активные запросы приёма, не следует освобождать, в противном случае будет невозможно проверить, что запрос был завершен. Предпочтительнее, в общем случае, освобождать запросы, когда они неактивные. Если следовать этому правилу, то функции, описанные в этой части задействуются в следующей последовательности форм

```
Create (Start Complete)* Free,
```

где * означает 0 или более повторений. Если некий объект обмена используется в нескольких параллельных ветвях, то на программиста ложится ответственность за координацию вызовов так, чтобы соблюдалась корректная последовательность.

Операция передачи, инициируемая с помощью MPI_START, может быть сопоставлена с любой операцией приёма, и также, операция приёма, инициируемая с помощью MPI_START может получать сообщения, сгенерированные любой операцией передачи.

3.10 Передача-прием

Операции передачи-приема объединяют в один вызов передачу сообщения одному процессу и получение другого сообщения, от другого процесса, возможно того же самого. Операцию передачи-приема хорошо использовать для выполнения операций сдвига вдоль цепочки процессов. Если блокированные передачи и приемы используются для такого сдвига, тогда необходимо корректно упорядочить передачи и приемы (например, четные процессы передают, затем принимают, нечетные процессы сначала принимают, затем передают) так, чтобы предотвратить циклические взаимо-эффекты, которые приводят к тупику. Когда используется операция передачи-приема, подсистема сама связи следит за их выполнением. Операция передачи-приема может быть использована совместно с функциями, описанными в Главе 6, чтобы выполнять сдвиги по различным логическим топологиям. Также операция передачи-приема полезна для реализации вызовов удаленных процедур.

Сообщение, посланное операцией передачи-приема, может быть получено обычной операцией приема или просканироваться операцией сканирования; операция передачи-приема может получать сообщение, посланное обычной операцией передачи.

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
recvcount, recvtype, source, recvtag, comm, status)
IN     sendbuf              адрес начала передаваемого буфера (choise)
IN     sendcount            число элементов передаваемого буфера (integer)
IN     sendtype             тип элементов передаваемого буфера (handle)
IN     dest                 ранк приемника (integer)
IN     sendtag              тэг передачи(integer)
OUT    recvbuf              адрес начала приемного буфера (choice)
IN     recvcount            число элементов приемного буфера (integer)
IN     recvtype             тип элементов приемного буфера (handle)
IN     source               ранк источника (integer)
IN     recvtag              тэг приемника (integer)
IN     comm                 коммуникатор (handle)
OUT    status               объект статус (Status)

int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
sendtype, int dest, int sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
       RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
       <type> SENDBUF(*), RECVBUF(*)
       INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,
       RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE),
       IERROR
```

Выполняют блокирующую операцию передачи и приема. Как передача так и прием используют один и тот же коммуникатор, но возможны различные тэги. Передающий буфер и приемный буфер разнесены, и могут иметь различную длину и тип данных.

```
MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source,
recvtag, comm, status)
INOUT         buf           адрес начала буфера передачи и приема (choice)
IN            count         число элементов в буфере передачи и приема
                            (integer)
IN            datatype      тип элементов и буфере передачи и приема
                            (handle)
IN            dest          ранк приемника (integer)
IN            sendtag       тэг передаваемого сообщения (integer)
IN            source        ранк источника (integer)
IN            recvtag       тэг принимаемого сообщения (integer)
IN            comm          коммуникатор (handle)
INOUT         status        объект статуса (Status)

int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag,
MPI_Comm com, MPI_Status *status)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE,
                     RECVTAG, COMM, STATUS, IERROR)
       <type> BUF(*)
       INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
       STATUS(MPI_STATUS_SIZE), IERROR
```

Выполняют блокирующие передачу и прием; один и тот же буфер используется как для передачи так и для приема, так что переданное сообщение замещается принятым сообщением.

Семантика операции передачи-приема такая, какая получилась бы, если инициатор вызова разветвился на две параллельные ветви, и одну использовал для выполнения передачи, и другую для выполнения приема, заканчивающихся слиянием этих двух ветвей. Дополнительная промежуточная буферизация необходима для "замещения" вариантов.

3.11 Нулевые процессы

Во многих примерах, удобно специфицировать "макетный" источник или приёмник для обмена. Это упрощает код, который необходим для операций с границами, например, в случае нециклического сдвига, производимого путем вызова операции передачи-приема.

Специальное значение MPI_PROC_NULL может быть использовано вместо номера, где в вызове требуется аргумент источника или приёмника. Связь с процессом MPI_PROC_NULL не производит никаких действий: передача в MPI_PROC_NULL выполняется и возвращает управление на столько быстро на сколько возможно. Приём от MPI_PROC_NULL выполняется настолько быстро на сколько возможно не изменяя приемный буфер.

Когда прием производится с источника source = MPI_PROC_NULL, объект статуса возвращает source = MPI_PROC_NULL, tag = MPI_ANY_TAG и count = 0.

3.12 Производные типы данных

Как было объяснено выше, все двухточечные обмены задействуют только непрерывные буфера, содержащие последовательность элементов определенного типа. Это сильное ограничение по двум соображениями. Во первых, часто желательно передать сообщение, которое содержит значения с различными типами данных (например, целое число, за которым следуют последовательность действительных чисел); во вторых, бывает необходимо послать разбросанные данные (например, подблок матрицы). Одно решение - упаковать фрагментированные данные в непрерывный буфер в передатчике на одном конце и распаковать их обратно в приемнике на другом. Это приводит к накладкам из-за потребности в дополнительных операциях копирования из памяти в память на обоих концах, даже когда подсистема связи имеет возможности разброса-сбора данных. Вместо этого MPI обеспечивает механизм для специфицирования более обобщенно смешанных и фрагментированных буферов связи. Поднимемся на уровень разработчиков, чтобы решить должны ли данные быть упакованы сразу в непрерывный буфер или они могут быть собраны непосредственно с того места, где они находятся.

Обобщенный механизм предлагаемый здесь позволяет передавать непосредственно без копирования объекты различной формы и размера. Опустим тот факт, что библиотека MPI распознает объекты, описанные во включающем языке; следовательно если надо передать структуру, или секцию массива, будет необходимым обеспечить в MPI определение буфера связи, который имитирует определение структуры или секции массива в запросе. Эти возможности могут быть использованы разработчиками библиотеки для определения функций связи, которые могут передавать объекты, определенные во включающем языке - путем декодирования этих определений как фактических в таблице символов или дискрипторе массива. Такие функции высокоуровневой связи не являются составной частью MPI.

Более общие буфера связи специфицируются при помощи замены основных типов данных, которые пока использовались на производные типы данных, сконструированные из основных типов данных на основе конструкторов, описанных в этой части. Такие методы конструирования производных типов данных могут применяться рекурсивно.

Общий тип данных является непрозрачным объектом, определяющим две вещи:

¦ последовательность базовых типов данных
¦ последовательность целых (байтовых) смещений

Смещения не обязательно должны быть положительными, отличными от своих соседей, или упорядоченными по возрастанию; следовательно не требуется, чтобы порядок элементов совпадал с их порядком в памяти, и элементы могут появляться более чем один раз. Мы называем

такую пару последовательностей (или последовательность пар) картой типа. Последовательность базовых типов данных (без смещений) является сигнатурой типа данного типа данных.

Пусть

Typemap = {(type0, disp0),...,(typen-1, dispn-1)},

будет такой картой типа, где typei являются базовыми типами, а dispi являются смещениями. Пусть

Typesig = {type0,...,typen-1}

будет связанной сигнатурой типа. Такая карта типа, вместе с базовым адресом buf, определяет буфер связи: такой буфер связи, состоящий из n входов, где i-ый вход есть адрес buf + dispi, имеет тип typei. Сообщение связанное с таким буфером связи будет состоять из n значений, типы которых определены в Typesig.

Мы можем использовать прототип для общего типа данных как аргумент в операциях передачи или приема, при замене аргументов базового типа данных. Операция MPI_SEND(buf,1,datatype,...) будет использовать передающий буфер, определенный базовым адресом buf и общим типом данных, связанным с datatype; она будет вырабатывать сообщение, с сигнатурой типа, определенной аргументом datatype. MPI_RECV(buf,1,datatype,...) будет использовать приемный буфер, определенный базовым адресом buf и общим типом данных, связанным с datatype.

Общие типы данных могут быть использованы во всех операциях передачи или приёма. Ниже в разделе 3.12.5 мы описываем случай, когда аргумент count имеет значение > 1.

Базовые типы данных, представленные в разделе 3.2.2 являются специальными случаями общих типов данных, и предопределяются. Следовательно MPI_INT есть прототип, предопределенный для типа данных с картой типа {(int,0)}, с одним элементом типа int и смещением ноль. Аналогично для всех других базовых типов.

Протяжённость типа данных по определению есть кратчайшее расстояние от первого до последнего байта, занятых элементами этого типе данных, округленным в большую сторону для выполнения требований выравнивания. То есть, если

Typemap = {(type0, disp0),...,(typen-1, dispn-1)},

тогда

lb(Typemap) = min dispj,

ub(Typemap) = max (dispj + sizeof(typej)), и

extent(Typemap) = ub(Typemap) - lb(Typemap) + ¦.

Если typei требует выравнивания для адреса, который является кратным ki, тогда ¦ есть наименьшая положительная добавка, необходимая для округления extent(Typemap) до следующего кратного maxiki.

Пример: Предположим, что Type = {(double ,0), (char,8)} (double со смещением ноль, после которого следует char со смещением восемь). Предположим, далее, что double должны быть точно выровнены по адресам, которые кратны восьми. Тогда, протяжённость этого типа данных - 16 (9 округляется до следующего кратного 8). Тип данных, состоящий из символа, за которым непосредственно следует double, также имеет размер 16.

Целесообразность. Определение протяжённости мотивируется тем предположением, что содержимое, добавляемое в конец каждой структуры в массиве структур есть наименьшее необходимое для соблюдения ограничений по выравниванию. Более точный контроль размера дается в разделе 3.12.3. Такой точный контроль

необходим в случаях, когда это предположение не соблюдается, например, где используются объединенные типы. (Конец замечания.)

3.12.1 Конструкторы типов данных

Непрерывный Простейший конструктор типа данных это MPI_TYPE_CONTIGUOUS, который позволяет размножение типов данных в непрерывные области.

```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
IN     count          число копий (nonnegative integer)
IN     oldtype        старый тип данных (handle)
OUT    newtype        новый тип данных (handle)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
       INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

newtype это тип данных, полученный конкатенацией count копий oldtype. Конкатенация определяется используя параметр extent как размер связываемых копий.

Пример: пусть oldtype имеет карту типа {(double,0),(char,8)} с протяжённостью 16, пусть count = 3. Карта типа, возвращенная в newtype будет следующая

```
 {(double,0),(char,8),(double,16),(char,24),(double,32),(char,40)};
```

то есть, это чередующиеся double и char элементы, со смещениями 0, 8, 16, 24, 32, 40.
Вообще, предположим, что карта типа oldtype

```
       {(type0,disp0),...,(typen-1, dispn-1)},
```

имеет протяжённость extent. Тогда newtype имеет карту типа с count¦n входами, определенную так:

```
       {(type0, disp0),...,(typen-1, dispn-1), (type0, disp0+extent),...
..,(typen-1, dispn-1+extent),...,(type0, disp0+extent¦(count-1)),...,
,(typen-1, dispn-1+extent¦(count-1))}.
```

Вектор Функция MPI_TYPE_VECTOR более общий конструктор, который позволяет размножение типов данных в ячейки памяти, которые состоят из равных блоков. Каждый блок получается конкатенацией определенного числа копий старого типа данных. Интервал между блоками есть множество протяжённостью старого типа данных.

```
MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype)
IN     count               число блоков (nonnegative integer)
IN     blocklength         число элементов в каждом блоке (nonnegative
                           integer)
IN     stride              число элементов между началом каждого блока
                           (integer)
IN     oldtype             старый тип данных (handle)
OUT    newtype             новый тип данных (handle)

int MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatypeoldtype, MPI_Datatype *newtype)

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR)
       INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

Пример: Предположим, опять, что oldtype имеет карту типа {(double,0) ,(char,8)}, с протяжённостью 16. Вызов

MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype) создаст тип данных с
такой картой

{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40),
(double,64),(char,72),(double,80),(char,88),(double,96),
(char,104)}:

два блока с тремя копиями старого типа, отстоящими на 4¦16 байтов
друг от друга.
Вызов MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) создаст тип
данных

{(double,0), (char,8), (double,-32), (char,-24), (double,-64),
(char,-56)}.

Вообще, предположим, что oldtype имеет карту типа

{(type0, disp0),...,(typen-1, dispn-1)},

с протяжённостью extent. Вновь созданный тип данных имеет карту
типа с count¦blocklength¦n входами:

{(type0, disp0),...,(typen-1, dispn-1),
 (type0, disp0 + extent),...,(typen-1, dispn-1 + extent),..,
 (type0, disp0 + (blocklength-1)¦extent),...,
 (typen-1, dispn-1 + (blocklength-1)¦extent),
 (type0, disp0 + stride¦extent),...,
 (typen-1, dispn-1 + stride¦extent),...,
 (type0, disp0 + (stride + blocklength-1)¦extent),...,
 (typen-1, dispn-1 + (stride + blocklenth - 1)¦extent),...,
 (type0, disp0 + stride¦(count-1)¦extent),...,
 (typen-1, dispn-1 + stride¦(count-1)¦extent),...,
 (type0, disp0 + (stride¦(count-1) + blocklength-1)¦extent),...,
 (typen-1, dispn-1 + (stride¦(count-1) + blocklength-1)¦extent)}

Вызов MPI_TYPE_CONTIGUOUS(count,oldtype,newtype) эквивалентен
вызову MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), или вызову
MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), где n -
произвольное.

Нвектор Функция MPI_TYPE_HVECTOR идентична MPI_TYPE_VECTOR, за
исключением того, что stride дается в байтах, а не в элементах.
Использование обоих типов вектор конструкторов проиллюстрировано в
разделе 3.12.7. (H используется для обозначения "heterogeneous"
(разнородный)).

MPI_TYPE_HVECTOR(count, blocklength, stride, oldtype, newtype)
IN   count   число блоков (nonnegative integer)
IN   blocklength   число элементов в каждом блоке (nonnegative
                            integer)
IN   stride   число байт между началом каждого блока
                            (integer)
IN   oldtype   старый тип данных (handle)
OUT   newtype   новый тип данных (handle)

int MPI_Type_hvector(int count, int blocklength, int stride,
                MPI_Datatypeoldtype, MPI_Datatype *newtype)

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

Предположим, что oldtype имеет карту типа

{(type0, disp0),...,(typen-1, dispn-1)},

с протяжённостью extent. Вновь созданный тип данных имеет карту типа с count¦blocklength¦n входами:

```
{(type0, disp0),...,(typen-1, dispn-1),
 (type0, disp0 + extent),...,(typen-1, dispn-1 + extent),..,
 (type0, disp0 + (blocklength - 1)¦extent),...,
 (typen-1, dispn-1 + (blocklength - 1)¦extent),
 (type0, disp0 + stride),..., (typen-1, dispn-1 + stride),...,
 (type0,disp0 + (stride + blocklength - 1)¦extent),...,
 (typen-1, dispn-1 + (stride + blocklenth - 1)¦extent),...,
 (type0, disp0 + stride¦(count - 1)),...,
 (typen-1, dispn-1 + stride¦(count - 1)),...,
 (type0, disp0 + stride¦(count-1) + (blocklength - 1)¦extent),...,
 (typen-1, dispn-1 + stride¦(count-1) + (blocklength - 1)¦extent)}
```

Индексированные Функция MPI_TYPE_INDEXED позволяет размножение старого типа в последовательность непрерывных блоков, где каждый блок может содержать различное число копий и иметь различное смещение. Все смещения блоков кратны протяжённости старого типа.

```
MPI_TYPE_INDEXED(count, array_of_blocklength,
       array_of_displacements, oldtype, newtype)
IN     count                              число блоков - также число входов в
                                          array_of_displacement и
                                          array_of_blocklength (nonnegative
                                          integer)
IN     array_of_blocklengths              число элементов в каждом блоке
                                          (массив nonegative integer)
IN     array_of_displacements             смещение каждого блока, в количестве
                                          протяжённостей oldtype (массив
                                          integer)
IN     oldtype                            старый тип данных (handle)
OUT    newtype                            новый тип данных (handle)
```

```
int MPI_Type_indexed(int count, int *array_of_blocklength,
               int *array_of_displacements, MPI_Datatype oldtype,
               MPI_Datatype *newtype)
```

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTH,
       ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
       INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
       ARRAY_OF_DISPLACEMENTS(*),OLDTYPE, NEWTYPE, IERROR
```

Пример: пусть oldtype имеет карту типа {(double,0),(char,8)}, с размером 16. Пусть B=(3,1) и пусть D=(4,0). Вызов MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) возвращает тип данных с картой типа

```
   {(double,64), (char,72), (double,80), (char,88), (double,96),
(char,104), (double,0), (char,8)}:
```

три карты старого типа, начинающиеся со смещения 64, и одна копия, начинающаяся со смещения 0.

Вообще, предположим, что oldtype имеет карту типа

```
       {(type0, disp0),...,(typen-1, dispn-1)},
```

с протяжённостью extent. Пусть B будет аргументом array_of_blocklength и D будет аргументом array_of_displacements. Вновь созданный тип данных имеет n¦¦i=0count-1B[i] элементов:

```
{(type0, disp0 + D[0]¦extent),...,(typen-1, dispn-1 + D[0]¦extent),
 (type0, disp0 + (D[0] + B[0]-1)¦extent),...,
 (typen-1, dispn-1 + D[0] + B[0]-1)¦extent),..,
 (type0, disp0 + D[count-1]¦extent),...,
 (typen-1, dispn-1 + D[count-1]¦extent),
```

```
 (type0, disp0 + (D[count-1] + B[count-1])¦extent),...,
 (typen-1, dispn-1 + (D[count-1]+B[count-1]-1)¦extent)}.
```

Вызов MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) эквивалентен вызову MPI_TYPE_INDEXED(count, B, D, oldtype, newtype), где

```
D[j] = j¦stride, j = 0,...,count-1,
```

и

```
B[j] = blocklength, j = 0,...,count-1.
```

Ниндексированный Функция MPI_TYPE_HINDEXED идентична MPI_TYPE_INDEXED, за исключением того, что смещения блоков в array_of_displacements определены в байтах, а не в значениях, кратных протяжённости oldtype. (H используется для обозначения "heterogeneous" (разнородный)).

```
MPI_TYPE_HINDEXED(count, array_of_blocklengths,
array_of_displacrments, oldtype, newtype)
```

IN	count	число блоков - также число входов в array_of_displacement и array_of_blocklength (nonnegative integer)
IN	array_of_blocklengths	число элементов в каждом блоке (массив nonegative integer)
IN	array_of_displacements	смещение каждого блока, в байтах (массив integer)
IN	oldtype	старый тип данных (handle)
OUT	newtype	новый тип данных (handle)

```
int MPI_Type_hindexed(int count, int *array_of_blocklength,
               int *array_of_displacements, MPI_Datatype oldtype,
               MPI_Datatype *newtype)

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTH,
ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
       INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
       ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR
```

Предположим, что oldtype имеет карту типа

```
{(type0, disp0),...,(typen-1, dispn-1)},
```

с протяжённостью extent. Пусть B будет аргументом array_of_blocklength и D будет аргументом array_of_displacements. Вновь созданный тип данных имеет n¦¦i=0count-1B[i] элементов:

```
{(type0, disp0 + D[0]),...,(typen-1, dispn-1 + D[0]),
 (type0, disp0 + D[0] + (B[0]-1)¦extent),...,
 (typen-1, dispn-1 + D[0] + (B[0]-1)¦extent),..,
 (type0, disp0 + D[count-1]),..., (typen-1, dispn-1 + D[count-1]),
 (type0, disp0 + D[count-1] + (B[count-1] - 1)¦extent),...,
 (typen-1, dispn-1 + (D[count-1] + (B[count-1]-1)¦extent)}.
```

Структурный MPI_TYPE_STRUCT наиболее общий конструктор: он обобщает предыдущие конструкторы таким образом, что позволяет каждому блоку состоять из различных типов данных.

```
MPI_TYPE_STRUCT(count, array_of_blocklengths,
       array_of_displacements, array_of_types, newtype)
```

IN	count	количество блоков(integer) - также число элементов в массивах array_of_types, array_of_displacements и

		array_of_blocklengths
IN	array_of_blocklengths	число элементов в каждом блоке (массив integer)
IN	array_of_displacements	смещение каждого блока в байтах (массив integer)
IN	array_of_types	тип элемента в каждом блоке (массив handler типов данных)
OUT	newtype	новый тип данных (handle)

```
int MPI_Type_struct(int count, int *array_of_blocklength,
               MPI_Aint *array_of_displacements,
               MPI_Datatype*array_of_types,
               MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
       INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
       ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

Пример: Пусть type1 имеет карту типа

{(double,0),(char,8)},

с протяжённостью 16. Пусть B=(2,1,3), D=(0,16,26), и T=(MPI_FLOAT, type1, MPI_CHAR). Тогда вызов MPI_TYPE_STRUCT(3, B, D, T, newtype) возвращает тип данных с картой,

{(float,0), (float,4), (double,16), (char,24), (char,26), (char,27), (char,28)}:

эти две копии MPI_FLOAT начинаются с 0, затем копия типа type1 начинается с 16, далее идут три копии MPI_CHAR, начинающиеся с 26. (Мы считаем, что float занимает 4 байта)

Предположим в общем случае, что T - аргумент array_of_types, где T[i] - прототипы для

typemapi = {(type0i, disp0i),...,(typein-1, dispin-1)},

с протяжённость extenti. Пусть B - аргумент array_of_blocklength и D - аргумент array_of_displacements. Тогда вновь созданный тип данных будет иметь карту с ¦i=0count-1B[i]¦ni элементами

Вызов MPI_TYPE_HINDEXED(count,B,D,oldtype,newtype) эквивалентен вызову MPI_TYPE_STRUCT(count,B,D,T,newtype), где каждый элемент T равен oldtype.

3.12.2 Функции адреса и протяжённости

Смещения в общих типах данных вычисляются относительно некоторого исходного адреса буфера. Эти смещения могут быть заменены абсолютными адресами: мы трактуем их как смещения относительно "нулевого адреса", то есть адреса начала адресного пространства. Этот первоначальный нулевой адрес обозначается константой MPI_BOTTOM. Таким образом, типы данных могут выражаться абсолютными адресами входов в буфер связи, в таком случае аргумент buf принимает значение MPI_BOTTOM. Адреса размещения в памяти могут быть найдены с помощью функции MPI_ADRESS.

MPI_ADRESS (location, address)

IN	location	местоположение в памяти (choice)
OUT	adress	адрес расположения (integer)

```
int MPI_Adress (void *location, MPI_Aint *address)

MPI_ADRESS (LOCATION, ADDRESS, IERROR)
       <type>        LOCATION(*)
```

```
       INTEGER       ADDRESS, IERROR
```

Возвращает (байтовый) адрес расположения.

Пример:

```
       REAL A(100, 100)
       INTEGER I1, I2, DIFF
       CALL MPI_ADDRESS (A(1, 1), I1, IERROR)
       CALL MPI_ADDRESS (A(10, 10), I2, IERROR)
       DIFF = I2 - I1
       ! Значения DIFF равно 909 * sizeofreal, значения I1 и I2
зависят от реализации.
```

Совет пользователям. Си пользователи могут избежать применения MPI_ADDRESS и полагаться на имеющийся адресный оператор &. Однако &выражение является указателем, а не адресом. ANSI C не требует, чтобы значение указателя (или указателя на int) было абсолютным адресом объекта, хотя это уже общий случай. Далее, ссылки могут и не иметь уникальные определения на машинах с сегментным адресным пространством. Использование MPI_ADDRESS со ссылками на Си переменные гарантирует совместимость с такими машинами. Следующие вспомогательные функции обеспечивают полную информацию о получаемых типах данных. (Конец совета пользователям.)

```
MPI_TYPE_EXTENT (datatype, extent)
IN     datatype             тип данных (handle)
OUT    extent               протяженность типа данных (integer)

int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent)

MPI_TYPE_EXTENT (DATATYPE, EXTENT, IERROR)
       INTEGER DATATYPE, EXTENT, IERROR
```

Возвращает протяжённость типа данных, как она определена в части 3.1.

```
MPI_TYPE_SIZE (datatype, size)
IN     datatype             тип данных (handle)
OUT    size                 размер типа данных (integer)

int MPI_Type_size (MPI_Datatype datatype, MPI_Aint *size)

MPI_TYPE_size (DATATYPE, SIZE, IERROR)
       INTEGER DATATYPE, SIZE, IERROR
```

Возвращает число байт занятых данными, то есть сумму размеров элементов типов данных.

```
MPI_TYPE_COUNT (datatype, count)
IN     datatype             тип данных (handle)
OUT    count                количество типов данных (integer)

int MPI_TYPE_COUNT (MPI_Datatype datatype, int *count)

MPI_TYPE_COUNT (DATATYPE, COUNT, IERROR)
       INTEGER       DATATYPE, COUNT, IERROR
```

Возвращает число высокоуровневых типов данных.

3.12.3 Маркеры нижней и верхней границ

Часто удобно точно определять нижнюю и верхнюю границы карты типов и переназначать определение, данное в части 3.1. Это позволяет определить типы данных, которые имеют "дырки" в начале и в конце; или типы данных, которые простираются выше верхней границы или ниже нижней границы. Примеры даны в разделе 3.12.7. Достигая

этого, мы складываем два "псевдо-типа" MPI_LB и MPI_UB, которые могут быть использованы, соответственно, для того, чтобы отметить нижнюю или верхнюю границу типа данных. Эти "псевдо-типы" не занимают памяти (extent(MPI_LB) = extent(MPI_UB) = 0). Они не влияют на размер или количество типов данных, и не занимают сообщений, создаваемых вместе с типом данных. Однако, они влияют на точность протяженности типа данных и, следовательно, влияют на результат дублирования этого типа данных конструктором типа данных.

Пример: пусть D = (-3, 0, 6); T = (MPI_LB, MPI_INT, MPI_UB) и B = (1, 1, 1). Когда производится вызов MPI_TYPE_STRUCT (3, B, D, T, type1), то создается новый тип данных, который имеет протяженность 9 (от -3 до 5 включительно) и содержит целое со смещением 0. Этот тип данных определен последовательностью {(lb, -3), (int, 0), (ub, 6)}. Если этот тип дублируется вызовом MPI_TYPE_CONTIGUOUS (2, type1, type2) то заново созданный тип, может быть описан последовательностью {(lb, -3), (int, 0), (int, 9), (ub, 15)}. (Записи типов lb или ub могут быть уничтожены, если они находятся не в конце типа данных).

В общем, если

Typemap = {(type0, disp0), ..., (typen-1, dispn-1)},

то нижняя граница Typemap определяется так:

если нет элементов lb
иначе

Подобно этому верхняя граница Typemap определяется

если нет элементов ub
иначе

Тогда extent(Typemap) = ub(Typemap) - lb(Typemap) + ¦

Если типу typej требуется выравнивание на адрес байта и адрес кратен ki, то ¦ есть наименьшее неотрицательное приращение необходимое для округления протяженности (extent (Typemap)) к следующему кратному maxiki.

Формальное определение, данное для конструкторов разнообразных типов данных, применяют теперь с улучшенным определением extent.

Две следующие функции могут быть использованы для вычисления нижней или верхней границы типа данных.

MPI_TYPE_LB (datatype, displacement)
IN datatype тип данных (handle)
OUT displacement смещение нижней границы от начала в байтах (integer)

int MPI_Type_lb (MPI_Datatype datatype, int *displacement)

MPI_TYPE_LB (DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB (datatype, displacement)
IN datatype тип данных (handle)
OUT displacement смещение верхней границы от начала в байтах (integer)

int MPI_Type_ub (MPI_Datatype datatype, int *displacement)

MPI_TYPE_UB (DATATYPE, DISPLACEMENT, IERROR)

```
       INTEGER DATATYPE, DISPLACEMENT, IERROR
```

Пример:

```
       MPI_Type_extent (type, extent)
       MPI_Type_lb (type, lb)
       MPI_Type_ub (type, ub)
       if (extent != ub - lb)
       printf ("MPI library failed validation test\n");
```

3.12.4 Фиксация и освобождение

Объект типа данных запоминается перед тем, как он может быть использован в обмене. Система может "видоизменять" внутреннее представление типа данных, которое способствует передаче, например, изменять представление от компактного к расширенному и выбирать более удобный механизм передачи. Такой тип данных может быть использован как аргумент в конструкторе типа данных.

Замечание: операция по фиксации типа данных сохраняет этот тип данных, то есть сохраняет его формальный описатель коммуникационного буфера, но не содержание этого буфера. Таким образом, после того, как тип данных был зафиксирован он может быть использован повторно для того, чтобы передать измененное содержание буфера или содержание нескольких различных буферов с разными стартовыми адресами.

```
MPI_TYPE_COMMIT (datatype)
INOUT         datatype             тип данных, который запоминается (handle)

int MPI_Type_commit (MPI_Datatype *datatype)

MPI_TYPE_COMMIT (DATATYPE, IERROR)
       INTEGER DATATYPE, IERROR
```

Не нужно запоминать базовые типы данных, они являются "предсохраненными".

```
MPI_TYPE_FREE (datatype)
INOUT         datatype             тип данных, который освобождается (handle)


int MPI_Type_free (MPI_Datatype *datatype)

MPI_TYPE_FREE (DATATYPE, IERROR)
       INTEGER DATATYPE, IERROR
```

Отмечает тип данных ассоциированный с datatype для освобождения и устанавливает datatype в MPI_DATATYPE_NULL. Любой обмен, который использует этот тип данных, будет завершен нормально. Производный тип данных, который был определен от освобожденного типа, не эффективен.

Пример:

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS (5, MPI_REAL, type1, ierr)
                     ! создан новый тип объекта
CALL MPI_TYPE_COMMIT (type1, ierr)
                     ! теперь type1 может быть использован для обмена
type2 = type1
                     ! type2 может быть использован для связи
                     ! (он является указателем на тот же объект, на
                     !  который указывает и type1
CALL MPI_TYPE_VECTOR (3, 5, 4, MPI_REAL, type1, ierr)
                     ! создан новый не сохраненный объект
CALL MPI_TYPE_COMMIT (type1, ierr)
                     ! теперь type1 заново может быть использован
                     ! для обмена
```

Освобождение типа данных не влияет на любой другой тип, который был создан от освобожденного типа данных. Система поступает так, как если бы входные аргументы типа данных наследуются конструкторами, переданными по значению.

Совет разработчикам. Реализация может сохранять количество ссылок на активные обмены, которые использует тип данных, и таким образом, знать в каком порядке их освобождать. Кроме того, могут поддерживаться конструкторы производных типов данных, таких, которые содержат указатели к аргументам, а не копируют их. В таком случае нужно отслеживать ссылки на определение активного типа данных для того чтобы знать, когда объект этого типа данных может быть освобожден. (Конец совета разработчикам.)

3.12.5 Использование общих типов данных

Хэндлеры выведенных типов данных, могут быть переданы коммуникационным вызовам, где требуется аргумент типа данных. Вызов вида MPI_SEND (buf, count, datatype, ...), где count > 1 рассматривается так, как если вызов обрабатывал новый тип данных, который соединяет count копий этого типа. Таким образом, MPI_SEND (buf, count, datatype, dest, tag, comm) эквивалентен

```
MPI_TYPE_CONTIGUOUS (count, datatype, newtype)
MPI_TYPE_COMMIT (newtype)
MPI_SEND (buf, 1, newtype, dest, tag, comm)
```

Возможно такое же употребление всех других коммуникационных функций, которые имеют count и datatype аргументы.

Предположим, что операция пересылки MPI_SEND (buf, count, datatype, dest, tag, comm) выполняется, когда datatype имеет тип карты

```
{ (type0, disp0), ..., (typen-1, dispn-1) },
```

с протяженностью extent(пустые элементы удалены). Тогда буфер используемый операцией передачи, содержит n¦count элементов, где элемент i¦n+j располагается по addri,j = buf + extent¦i + dispj и с типом typej для i=0,...,count-1 и j=0,...,n-1. Эти элементы не обязаны располагаться ни непрерывно, ни отдельно друг от друга, их порядок вообще может быть произвольным.

Переменная, хранящаяся по адресу addri,j в вызывающей программе, должна соответствовать типу typej. Соответствие типов определено в разделе 3.3.1. Посылаемое сообщение содержит n¦count элементов, где элемент i¦n+j имеет тип typej.

Подобно этому можно предположить, что выполняется операция приёма сообщения MPI_RECV (buf, count, datatype, source, tag, comm, status), datatype имеет тип карты

```
{ (type0, disp0), ..., (typen-1, dispn-1) },
```

с протяженностью extent(пустые элементы удалены). Буфер для приёма сообщения состоит из n¦count элементов, где элемент i¦n +j начинается с buf + extent¦i + dispj и имеет тип typej. Если полученное сообщение состоит из k элементов, то мы должны иметь k ¦ n*count, тогда i¦n + jый элемент сообщения должен иметь тип, который соответствует типу typej.

Важно иметь ввиду, что соответствие типов определяется согласно сигнатуре посылаемого типа данных, то есть последовательностью компонентов базового типа и, следовательно, не зависит от определения типа данных.

Пример:

```
...
CALL MPI_TYPE_CONTIGUOUS (2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS (4, MPI_REAL, type4, ...)
CALL MPI_TYPE_CONTIGUOUS (2, type2, type22, ...)
```

```
...
CALL MPI_SEND (a, 4, MPI_REAL, ...)
CALL MPI_SEND (a, 2, type2, ...)
CALL MPI_SEND (a, 1, type22, ...)
CALL MPI_SEND (a, 1, type4, ...)
...
CALL MPI_RECV (a, 4, MPI_REAL, ...)
CALL MPI_RECV (a, 2, type2, ...)
CALL MPI_RECV (a, 1, type22, ...)
CALL MPI_RECV (a, 1, type4, ...)
```

Каждая операция передачи соответствует операции приёма.

Тип данных может указывать на перекрывающиеся записи. Если такой тип данных используется в операции приёма, то есть если некоторая часть приёмного буфера переписывается более чем один раз этой операцией, то этот вызов (операция) ошибочен.

Снова предположим, что MPI_RECV (buf, count, datatype, dest, tag, comm, status) в данный момент выполняется и datatype имеет карту

{ (type0, disp0), ..., (typen-1, dispn-1) }.

Полученное сообщение не обязано ни заполнять весь приёмный буфер, ни даже занимать место размер, которого кратен n. Может быть получено любое количество k базовых элементов, где 0 ¦ k ¦ count¦n.

Количество полученных элементов может быть восстановлено из status с помощью функции MPI_GET_ELEMENTS:

```
MPI_GET_ELEMENTS (status, datatype, count)
IN     status        возвращает статус операции приёма (Status)
IN     datatype      тип, использованный этой операцией (handle)
OUT    count         число полученных базовых элементов (integer)

int MPI_Get_elements (MPI_Status status, MPI_Datatype datatype, int
*count)

MPI_GET_ELEMENTS (STATUS, DATATYPE, COUNT, IERROR)
       INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Функция MPI_GET_COUNT ведет себя по другому: она возвращает количество полученных "элементов". В предыдущем примере MPI_GET_COUNT может возвратить любое целое значение k, где 0 ¦ k ¦ count. Если MPI_GET_COUNT возвращает k, то число полученных базовых элементов (и значение, возвращенное MPI_GET_ELEMENTS) равно n¦k. Если число полученных базовых элементов не кратно n, то есть операция приёма приняла не полное число "копий" типа данных datatype, то MPI_GET_COUNT возвращает значение MPI_UNDEFINED.

Пример:

```
...
CALL MPI_TYPE_CONTIGUOUS (2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT (Type2, ierr)
...
CALL MPI_COMM_RANK (comm, rank, ierr)
IF (rank.EQ.0) THEN
       CALL MPI_SEND (a, 2, MPI_REAL, 1, 0, comm, ierr)
       CALL MPI_SEND (a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE
       CALL MPI_RECV (a, 2, Type2, 0, 0, comm, stat, ierr)
       CALL MPI_GET_COUNT (stat, Type2, i, ierr) !возвр. i=2
       CALL MPI_GET_ELEMENTS (stat, Type2, i, ierr) !i=2
       CALL MPI_RECV (a, 2, Type2, 0, 0, comm, stat, ierr)
       CALL MPI_GET_COUNT (stat, Type2, i, ierr) !возвращает
                    ! i=MPI_UNDEFINED
       CALL MPI_GET_ELEMENTS (stat, Type2, i, ierr) !i=3
END IF
```

Функция MPI_GET_ELEMENTS может также быть использована после

попытки найти число элементов в проверенном сообщении. Отметим, что функции MPI_GET_COUNT и MPI_GET_ELEMENTS возвращают одинаковые значения, когда они используются с базовыми типами данных.

> Целесообразность. Дополнение данное к определению MPI_GET_COUNT кажется естественным: можно будет ожидать, что эта функция возвратит значение count, когда заполнится принимающий буфер. Иногда тип данных datatype представляется базовой единицей информации, которую нужно передать, например, записью в массиве записей (структур). Нужно иметь возможность определять количество компонентов, которое будет получено, и сделать это без деления на количество элементов в каждом компоненте. Однако при других обстоятельствах тип данных используется для того, чтобы определить комплексную схему информации в памяти получателя, и он не представляется единицей информации в передаче. В таком случае нужно использовать функцию MPI_GET_ELEMENTS. (Конец замечания.)

> Совет пользователям. Это определение подразумевает, что операция приёма не может изменить значения в памяти вне коммуникационного буфера. В частности определение означает, что заполненное пространство в структуре не будет изменено при копировании ее от одного процесса к другому. Это будет предотвращать очевидную оптимизацию копирования такой структуры вместе с ее заполнением, как одного сплошного блока. Реализация вольна делать эту оптимизацию тогда, когда она не влияет на результат вычислений. Пользователь может "усилить" эту оптимизацию, непосредственно включая заполнение в часть сообщения. (Конец совета пользователям.)

3.12.6 Корректное использование адресов

Последовательно объявленные переменные в Си или Frortran'е не обязательно хранятся последовательно. Таким образом, не нужно беспокоится о том, что произойдет пересечение при смещении от одной переменной к другой. Также в машинах с сегментным адресным пространством, адреса не являются уникальными и адресная арифметика имеет некоторые специфические свойства. Таким образом, использование адресов, то есть смещений относительно стартового адреса MPI_BOTTOM, ограничено.

Переменные располагаются в памяти последовательно, если они относятся к массиву, к COMMON блоку в языке Fortran или структуре в языке C. Действительными являются адреса, определенные рекурсивно, следующим образом:

> 1. Функция MPI_ADDRESS возвращает действительный адрес, который затем передается как аргумент вызывающей программе.
>
> 2. Аргумент buf функции обмена расценивается как действительный адрес, когда передается как аргумент вызывающей программе.
>
> 3. Если v является действительным адресом и i - это целое, то v+i - действительный адрес; v и v+i располагаются в последовательной памяти.
>
> 4. Если v - это действительный адрес, то MPI_BOTTOM+v тоже действительный адрес.

Правильная программа использует только действительные адреса, которые отождествляются с расположением записей в буфере связи. Далее, если u и v - два действительные адреса, то (целая) разность u-v может быть вычислена только если u и v находятся в одной последовательной памяти; других арифметических операций, которые могли бы иметь какое-то значение при выполнении вычислений над адресами нет.

Приведенные выше правила не принуждают к использованию

производных типов, для которых можно определить такой буфер связи, который весь содержится в последовательной памяти. Однако конструкция буфера связи, который содержит переменные, не находящиеся в той же памяти, должна подчиняться определенным ограничениям. Переменные в коммуникационном буфере, которые не находятся в той же памяти, могут использоваться только с помощью определения в коммуникационном вызове аргументов buf = MPI_BOTTOM, count = 1 и типа данных datatype, где все смещения есть действительные (абсолютные) адреса.

Совет пользователям. Не ожидается, что MPI реализации будут способны обнаруживать ошибочные "вне граничные" смещения. Если это переполнение адресного пространства пользователя, то MPI-вызов может не знать размерность массивов и записей в главной программе. (Конец совета пользователям.)

Совет разработчикам. Не нужно различать абсолютные адреса и относительные смещения на машинах со сплошным адресным пространством: MPI_BOTTOM равен нулю, а адреса и смещения целые. На машинах, где требуется различие, адреса преобразуются в выражения, которые включают в себя MPI_BOTTOM. (Конец совета разработчикам.)

3.12.7 Примеры

Следующие примеры иллюстрируют применение производных типов.

Первый пример: послать и получить часть трех-мерного массива (3D).

```
      REAL a (100, 100, 100), e (9, 9, 9)
      INTEGER oneslice, twoslice, threeslice, sizeofreal,
              myrank, ierr

C     выбрать часть a (1:17:2, 3:11, 2:10)
C     и сохранить ее в e (:,:,:).

      CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank)
      CALL MPI_TYPE_EXTENT (MPI_REAL, sizeofreal, ierr)

C     создать тип данных для 1D части.
      CALL MPI_TYPE_VECTOR (9, 1, 2, MPI_REAL, oneslice, ierr)

C     создать тип данных для 2D части.
      CALL MPI_TYPE_HVECTOR (9, 1, 100*sizeofreal, oneslice,
                             twoslice, ierr)

C     создать тип данных для секции записи
      CALL MPI_TYPE_HVECTOR (9, 1, 100*sizeofreal, twoslice,
threeslice, ierr)

      CALL MPI_TYPE_COMMIT (threeslice, ierr)
      CALL MPI_SENDRECV (a(1, 3, 2), 1, threeslice, myrank, 0,
                         e, 9*9*9, MPI_REAL, myrank, 0,
                         MPI_COMM_WORLD, status, ierr)
```

Второй пример: скопировать нижнюю треугольную часть матрицы.

```
      REAL a(100, 100), b(100, 100)
      INTEGER disp(100), blocklen(100), ltype, myrank, ierr
      MPI_STATUS status

C     скопировать нижнюю треугольную часть массива a
C     на нижнюю треугольную часть массива b

      CALL MPI_CALL_RANK (MPI_COMM_WORLD, myrank)
```

```
C       вычислить начало и размер каждой колонки
        DO i = 1, 100
          disp(i) = 100*(i-1) + i
          block(i) = 100-i
        END DO
C       создать тип данных для нижней треугольной части
        CALL MPI_TYPE_INDEXED (100, block, disp, MPI_REAL, ltype,
        ierr)
        CALL MPI_TYPE_COMMIT (ltype, ierr)
        CALL MPI_SENDRECV (a, 1, ltype, myrank, 0, b, 1, ltype,
                           myrank, 0, MPI_COMM_WORLD, status, ierr)

        Третий пример: транспонировать матрицу.

        REAL a(100, 100), b(100, 100)
        INTEGER row, xpose, sizeofreal, myrank, ierr
        MPI_STATUS status

C       транспонировать матрицу a на b

        CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank)
        CALL MPI_TYPE_EXTENT (MPI_REAL, sizeofreal, ierr)

C       создать тип данных для одной строки
        CALL MPI_TYPE_VECTOR (100, 1, 100, MPI_REAL, row, ierr)

C       создать тип данных для матрицы в порядке возрастания номеров
C       строк
        CALL MPI_TYPE_HVECTOR (100, 1, sizeofreal, row, xpose, ierr)

        CALL MPI_TYPE_COMMIT (xpose, ierr)

C       послать матрицу в порядке старших номеров строк и получить
C       ее в порядке возрастания номеров столбцов

        CALL MPI_SENDRECV (a, 1, xpose, myrank, 0,
                           b, 100*100, MPI_REAL, myrank, 0,
                           MPI_COMM_WORLD, status, ierr)

        Другой подход к проблеме транспонирования:

        REAL a(100, 100), b(100, 100)
        INTEGER disp(2), blocklen(2), type(2), row, row1,
                sizeofreal, myrank, ierr
        MPI_STATUS status

        CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank)

C       транспонировать матрицу a на b

        CALL MPI_TYPE_EXTENT (MPI_REAL, sizeofreal, ierr)

C       создать тип данных для одной строки
        CALL MPI_TYPE_VECTOR (100, 1, 100, MPI_REAL, row, ierr)

C       создать тип данных для одной строки с размером в одно
C       реальное число

        disp(1) = 0
        disp(2) = sizeofreal
        type(1) = row
        type(2) = MPI_UB
        blocklen(1) = 1
        blocklen(2) = 1

        CALL MPI_TYPE_STRUCT (2, blocklen disp, type, row1, ierr)
        CALL MPI_TYPE_COMMIT (row1, ierr)
```

```
C      послать 100 строк и получить в порядке возрастания номеров
C      столбцов

       CALL MPI_SENDRECV (a, 100, row1, myrank, 0,
                          b, 100*100, MPI_REAL, myrank, 0,
                          MPI_COMM_WORLD, status, ierr)


       Четвертый пример. Манипуляция с массивом структур.

struct Partstruct
       {
       int           class; /* отдельный класс */
       double        d[6];  /* отдельные координаты */
       char          b[7];  /* некоторая вспомогательная информация */
       };

struct Partstruct particle[1000];

int                  i, dest, rank;
MPI_COMM             comm;

/* строим тип данных, описывающий структуру */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int             blocklen[3] = {1, 6, 7};
MPI_Aint        disp[3];

/* вычислим смещения компонентов структуры */
MPI_Address (patricle, disp);
MPI_Address (patricle[0].d, disp+1);
MPI_Address (patricle[0].b, disp+2);
for (i=0; i<3; i++) disp[i] -= disp[0];

MPI_Type_struct (3, blocklen, disp, type, &Particletype);

/* Если компилятор не делает заполнения используя мистический путь,
то следующий путь может быть безопасным */

MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR,
              MPI_UB};
int          blocklen1[4] = {1, 6, 7, 1};
MPI_Aint     disp1[4];

/* вычислим смещение компонентов структуры */
MPI_Address (patricle, disp1);
MPI_Address (patricle[0].d, disp1+1);
MPI_Address (patricle[0].b, disp1+2);
MPI_Address (patricle+1, disp1+3);
for (i=0; i<4; i++) disp1[i] -= disp1[0];

/* построим тип данных, описывающий структуру */

MPI_Type_struct (4, blocklen1, disp1, type1, &Partcletype);

/* 4.1:
послать весь массив */

MPI_Type_commit (&Particletype);
MPI_Send (particle, 1000, Particletype, dest, tag, comm);

/* 4.2:
послать записи нулевого класса, предшествующих числу этих
записей */

MPI_Datatype Zparticles;  /* тип данных, описывающий все части
```

```
                          нулевого класса (нужен для
                          пересчета, если классы изменены)
                         */
MPI_Datatype Ztype;

MPI_Aint             zdisp[1000];
int                  zblock[1000], j, k;
int                  zzblock[2] = {1, 1};
MPI_Aint             zzdisp[2];
MPI_Datatype         zztype[2];

/* вычислить смещения частей нулевого класса */
j = 0;
for (i=0; i < 1000; i++)
  if (particle[i].class == 0)
     {
      zdisp[j] = i;
      zblock[j] = 1;
      j++;
     }


/* создать тип данных для частей нулевого класса */
MPI_Type_indexed (j, zblock, zdisp, Particletype,
                  &Zparticles);

/* подвесить счет частей */
MPI_Address (&j, zzdisp);
MPI_Address (particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct (2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit (&Ztype);
MPI_Send (MPI_BOTTOM, 1, Ztype, dest, tag, comm);

/* проблематично найти более эффективный путь определения
Zparticles */

/* последовательные части с нулевым индексом обозначаются как один
блок */
j=0;
for (i=0; i<1000; i++)
  if (particle[i].index == 0)
       {
       for (k=i+1; (k<1000) && (particle[k].index == 0); k++);
       zdisp[j] = i;
       zblock[j] = k-i;
       j++;
       i = k;
       }
MPI_Type_indexed (j, zblock, zdisp, Particletype,
                  &Zparticles);

/* 4.3:
послать первые две координаты всех записей */

MPI_Datatype Allpairs;  /* тип данных для всех записей */
MPI_Aint     sizeofentry;
MPI_Type_extent (Particletype, sizeofentry);

/* значение sizeofentry может также быть вычислено вычитанием
адреса particle[0] из адреса particle[1] */

MPI_Type_hvector (1000, 2, sizeofentry, MPI_REAL, &Allpairs);
MPI_Type_commit (&Allpairs);
MPI_Send (particle.d, 1, Allpairs, dest, tag, comm);
```

```
/* альтернативное решение 4.3 */

MPI_Datatype Onepair; /* тип данных для одной пары координат с
                         размером в одну часть записи */
MPI_Aint     disp[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int          blocklen2[3] = {1, 2, 1};

MPI_Address (particle, disp2};
MPI_Address (particle[0].d, disp2+1);
MPI_Address (particle+1, disp2+2);
for (i=0; i<2; i++) disp2[i] -= disp2[0];

MPI_Type_struct (3, blocklen2, disp2, type2, &Onepair);
MPI_Type_commit (&Onepair);
MPI_Send (particle[0].d, 1000, Onepair, dest, tag, comm);
```

Пятый пример: такая же манипуляция, как и в предыдущем примере, но с использованием абсолютных адресов типов данных.

```
struct Partstruct
       {
       int           class; /* отдельный класс */
       double        d[6];  /* отдельные координаты */
       char          b[7];  /* некоторая вспомогательная информация */
      };
struct Partstruct particle[1000];

/* строим тип данных, описывающий первый массив записей */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int             blocklen[3] = {1, 6, 7};
MPI_Aint        disp[3];

MPI_Address (patricle, disp);
MPI_Address (patricle[0].d, disp+1);
MPI_Address (patricle[0].b, disp+2);
MPI_Type_struct (3, block, disp, type, &Particletype);

/* Particletype описывает первый массив записей, используя
абсолютные адреса */

/* 5.1:
посылает полный массив */

MPI_Type_commit (&Particletype);
MPI_Send (MPI_BOTTOM, 1000, Particletype, dest, tag, comm);

/* 5.2:
послать записи нулевого класса, предшествующих числу этих
записей */
MPI_Datatype Zparticles, Ztype;
MPI_Aint             zdisp[1000];
int                  zblock[1000], j, k;
int                  zzblock[2] = {1, 1};
MPI_Aint             zzdisp[2];
MPI_Datatype         zztype[2];

j = 0;
for (i=0; i < 1000; i++)
  if (particle[i].index == 0)
     {
      for (k=i+1; (k<1000) && (particle[k].index == 0); k++);
      zdisp[j] = i;
      zblock[j] = k-i;
```

```
      j++;
      i = k;
     }
MPI_Type_indexed (j, zblock, zdisp, Particletype,
                  &Zparticles);
/* Zparticles описывает части с нулевым классом, используя их
абсолютные адреса */

/* подвесить счет частей */
MPI_Address (&j, zzdisp);
zzdisp[1] = MPI_BOTTOM;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct (2, zzblock, zzdisp, zztype, &Ztype);

MPI_Type_commit (&Ztype);
MPI_Send (MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

Шестой пример: указание объединений.

```
union {
   int        ival;
   float      fval;
      } u[1000];

int   utype;
MPI_Datatype type[2];
int          blocklen[2] = {1, 1};
MPI_Aint     disp[2];
MPI_Datatype mpi_utype[2];
MPI_Aint     i, j;

/* вычислить MPI адрес для каждого возможного типа объединения;
   присваиваемые значения выравниваются влево в памяти объединений
*/

MPI_Address (u, &i);
MPI_Address (u+1, &j);
disp[0] = 0;
disp[1] = j-i;
type[1] = MPI_UB;
type[0] = MPI_INT;
MPI_Type_struct (2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct (2, blocklen, disp, type, &mpi_utype[1]);
for (i=0; i<2; i++) MPI_Type_commit (&mpi_utype[i]);

/* актуальная связь */
MPI_Send (u, 1000, mpi_utype[utype], dest, tag, comm);
```

3.13 Упаковывание и распаковывание

Некоторые существующие коммуникационные библиотеки поддерживают упаковывающие/распаковывающие функции для передачи отделенной друг от друга информации: пользователь упаковывает информацию в единый буфер перед ее передачей и распаковывает ее из единого буфера после получения. Производные типы данных, которые описаны в части 3.12, позволяют в большинстве случаев избегать упаковывания и распаковывания: пользователь указывает схему посылаемой и получаемой информации, и тогда коммуникационная библиотека использует не цельный буфер. Процедуры упаковки/ распаковки информации предусмотрены для совместимости с предыдущими библиотеками. Кроме того, они предусмотрены для некоторых функциональных возможностей, которые без них (упаковки/распаковки) не были бы доступны в MPI: сообщение может быть получено отдельными частями. Операция получения производимая при приеме последней части

может зависеть от содержания предшествующей части; и уходящие сообщения могут быть буферизованы в памяти пользователя. Так обходится политика системного буферирования. Наконец доступность операций упаковки/распаковки облегчает развитие дополнительных коммуникационных библиотек, нанизанных на вершину MPI.

```
MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position,
              comm)
IN            inbuf         входной стартовый буфер (choice)
IN            incount       число входных параграфов информации (integer)
IN            datatype      тип данных каждого входного параграфа
                            (integer)
OUT           outbuf        выходной стартовый буфер (choice).
IN            outsize       размер выходного буфера в байтах (integer)
INOUT         position      текущая позиция в буфере (integer)
IN            comm          коммуникатор для упакованного сообщения
                            (handle)

int MPI_Pack (void *inbuf, int incount, MPI_Datatype datatype,
              void *outbuf, int outsize, int *position, MPI_Comm comm)

MPI_Pack (INBUF, INCOUT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM,
IERROR)
       <type> INBUF(*), OUTBUF(*)
       INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

Чтобы передать упакованное сообщение из одного буфера, надо точно определить inbuf, incount, datatype, а чтобы получить это сообщение, надо определить outbuf и outsize. Входной буфер может быть любым буфером, допускаемый MPI_SEND. Выходной буфер является непрерывной областью памяти, содержащей outcount байтов, начинающуюся по адресу outbuf (длина подсчитывается в байтах, как если бы это был коммуникационный буфер для сообщения типа MPI_PACKED).

Входное значение position является первым месторасположением в выходном буфере использующимся для упаковки. Параметр position увеличивается на размер упакованного сообщения. Выходное значение position есть первое свободное место в выходном буфере, которое следует за расположением упакованного сообщения. Аргумент comm - это коммуникатор, который впоследствии используется для передачи упакованного сообщения.

```
MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype,
comm)
IN            inbuf         начало входного буфера (choice)
IN            insize        размер входного буфера в характерных единицах
                            (integer)
INOUT         position      текущая позиция в буфере (integer)
OUT           outbuf        начало выходного буфера (choice)
IN            outcount      число параграфов, которое будет распаковано
                            (integer)
IN            datatype      тип данных каждой выходящей единицы данных
                            (integer)
IN            comm          коммуникатор для упакованного сообщения
                            (handle)

int MPI_Unpack (void *inbuf, int insize, int *position,
       void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm
comm)

MPI_UNPACK (INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
              DATATYPE, SOURCE, COMM, IERROR)
       <type> INBUF(*), OUTBUF(*)
       INTEGER INSIZE, DATATYPE, OUTCOUNT, POSITION, COMM, IERROR
```

Чтобы распаковать сообщение в принимающем буфере, надо определить значения outbuf, outcount, datatype и обозначить

параметры входного буфера с помощью значений inbuf и insize. Выходной буфер может быть любым буфером, допустимым MPI_RECV. Выходной буфер является непрерывной областью памяти, содержащей insize байт, начинающуюся по адресу inbuf. Входное значение position является первым месторасположением в выходном буфере, используемым упакованным сообщением. Параметр position увеличивается на размер упакованного сообщения, поэтому выходное значение position есть первое свободное место в выходном буфере, которое следует за расположением упакованного сообщения. Аргумент comm - это коммуникатор, который впоследствии используется для приема упакованного сообщения.

> Советы пользователям. Имеются различия между MPI_RECV и MPI_UNPACK: в MPI_RECV аргумент count обозначает максимальное число элементов, которое может быть получено. Реальное число полученных элементов обуславливается длиной пришедшего сообщения. В MPI_UNPACK аргумент count обозначает реальное количество элементов, которые распаковываются. "Размер" соответствующего сообщения является приращением для position. Основанием для его изменения является то, что "размер получаемого сообщения" не предопределен с того момента, как пользователь решил, сколько информации ему придется распаковывать, также не легко определить "размер сообщения" по числу элементов, которые будут распаковываться. Фактически в разнородных системах это число может не быть определено априори. (Конец совета пользователям.)

Чтобы понять поведение операций упаковывания и распаковывания, удобно думать о том, что информационная часть сообщения будто бы есть последовательность, полученная слиянием последовательных значений, посланных в сообщении. Операция упаковки сохраняет эту последовательность в буфере, как если бы сообщение посылалось в этот буфер. Операция распаковки восстанавливает эту последовательность из буфера, как будто бы сообщение принималось из этого буфера. (При этом полезно думать о внутренних FORTRAN файлах или функции sscanf в языке Си).

Несколько сообщений могут быть последовательно упакованы в одну упакованную единицу. Это достигается с помощью нескольких связанных вызовов MPI_PACK, где первый вызов резервирует position = 0, и для каждого последующего вызова вводится значение position, которое является выходным для предыдущего вызова, то же самое для значений outbuf, outsize и comm. Эта упакованная единица содержит теперь эквивалентную информацию о том, что будет сохранено в сообщении вызовом передачи с буфером передачи, который является "цепочкой" индивидуальных буферов передачи.

Упакованная единица может быть послана при использовании типа MPI_PACKED. Обмен точка с точкой или групповые обмены могут быть использованы для передачи последовательности байтов - упакованной единицы - от одного процесса к другому. Эта упакованная единица может теперь быть получена с помощью операции приёма с любым типом данных: правило соответствия типа ослабляется для сообщения, посланного с типом MPI_PACKED.

Сообщение, посланное с любым типом (включая MPI_PACKED), может быть получено с помощью использования типа MPI_PACKED. Такое сообщение может быть затем распаковано вызовом MPI_UNPACK.

Упакованная единица (или сообщение, созданное регулярной - "типовой" посылкой) может быть распаковано в несколько последовательных сообщений. Это достигается с помощью нескольких связанных вызовов MPI_UNPACK, где первый вызов устанавливает position = 0, и для каждого последующего вызова вводится значение position, которое является выходным у предыдущего вызова, то же самое для значений inbuf, insize и comm.

Слияние двух упакованных единиц не является необходимым действием для упакованной единицы, и упакованная единица не является подцепочкой одной упакованной единицы. Таким образом, одна единица не может быть образована из двух упакованных единиц и результат распаковки не может быть одной единицей, также не может

быть распакована одна подцепочка упакованной единицы как отдельная упакованная единица. Каждая упакованная единица, которая была создана связанной последовательностью вызовов упаковки, должна быть распакована как одна единица с помощью связанной последовательности распаковывающих вызовов.

Целесообразность. Ограничение на "атомную" упаковку и распаковку упакованной единицы требует присоединения к голове упакованных единиц дополнительной информации, такой, как описание архитектуры передаваемого (используется для преобразования типов в неоднородном окружении). (Конец замечания.)

Следующий вызов предоставляет пользователю возможность узнать, сколько памяти требуется, чтобы упаковать сообщение и, таким образом, организовать размещение буферов в памяти.

```
MPI_PACK_SIZE(incount, datatype, size)
IN     incount              количество (integer)
IN     datatype             тип данных (handle)
IN     comm                 коммуникатор (handle)
OUT    size                 размер упакованного сообщения в байтах
                            (integer)

int MPI_Pack_size (int incount, MPI_Datatype datatype,
                   MPI_Comm comm, int *size)

MPI_PACK_SIZE (INCOUNT, DATATYPE, COMM, SIZE, IERROR)
       INTEGER       INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

Вызов MPI_PACK_SIZE (incount, datatype, size) возвращает в size верхнюю границу для приращения position, к которому может привести вызов MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position).

Целесообразность. Вызов возвращает верхнюю границу, а не точную границу, поскольку точное количество памяти, необходимое для того, чтобы упаковать сообщение, может зависеть от контекста (например, первое сообщение, упакованное в упакованную единицу, может взять больше памяти). (Конец замечания.)

Пример 1:

```
int position, i, j, a[2];
char buff[1000];

       ...

MPI_Comm_rank (MPI_Comm_world, myrank);
if (myrank == 0)
       {
       /* код посылающего */
       position = 0;
       MPI_Pack (buff, 1000, position, i, 1, MPI_INT);
       MPI_Pack (buff, 1000, position, j, 1, MPI_INT);
       MPI_Pack (buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
       }
else /* код получателя */
       MPI_Recv (a, 2, MPI_INTEGER, 0, 0, MPI_COMM_WORLD);

}
```

Пример 2:

```
int position, i;
float a[1000];
char buff[1000];
```

```
       ...

MPI_Comm_rank (MPI_Comm_world, myrank);
if (myrank == 0)
       {
       /* код посылающего */
       int len[2];
       MPI_Aint disp[2];
       MPI_Datatype type[2], newtype;

       /* построить тип данных для i, за которой следуют a[0] ...
a[i-1] */
       len[0] = 1;
       len[1] = i;
       MPI_Address (&i, disp);
       MPI_Address (a, disp+1);
       type[0] = MPI_INT;
       type[1] = MPI_FLOAT;
       MPI_Type_struct (2, len, disp, type, &newtype);
       MPI_Type_commit (&newtype);

       /* упаковать i, за которой следуют a[0] ... a[i-1] */

       position = 0;
       MPI_Pack (MPI_BOTTOM, 1, newtype, buff, position,
MPI_COMM_WORLD);

       /* Послать */
       MPI_Send (buff, position, MPI_Byte, 1, 0, MPI_Comm_world);

       /* *****
       Можно заменить последние три строки
       MPI_Send (MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
       ***** */
       }
else /* myrank == 1 */
       {
       /* код получателя */
       MPI_Status status;

       /* получить */
       MPI_Recv (buff, 1000, MPI_Byte, 0, 0, status);

       /* распаковать i */
       position = 0;
       MPI_Unpack (buff, 1000, position, i, 1, MPI_Int);

       /* распаковать a[0] ... a[i-1] */
       MPI_Unpack (buff, 1000, position, a, i, MPI_Float);
       }
```

Пример 3: каждый процесс посылает количество символов, выраженное значением count, корневому процессу, который соединяет их в одну строку.

```
int count, gsize, counts[64], totalcount, k1, k2, k, displs[64],
position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
       ...

MPI_Comm_size (comm, gsize);
MPI_Comm_rank (comm, myrank);

/* разместить локальный буфер */
MPI_Pack_size (1, MPI_INT, comm, k1);
MPI_Pack_size (count, MPI_CHAR, k2);
```

```
k = k1 + k2;
lbuf = (char *)malloc (k);

/* упаковать count, за которым следует count символов */
position = 0;
MPI_Pack (&count, 1, MPI_INT, &lbuf, k, &position, comm);
MPI_Pack (chr, count, MPI_CHAR, &lbuf, k, &position, comm);

if (myrank != root)
       {
       /* собрать в корне размеры всех упакованных сообщений */
       MPI_Gather (&position, 1, MPI_INT, NULL, NULL, NULL, root,
comm);
       /* собрать в корне упакованные сообщения */
       MPI_Gather (&buf, position, MPI_PACKED, NULL, NULL, NULL,
              NULL, root, comm);
       }
else
       { /* корневой код */
       /* собрать в корне размеры всех упакованных сообщений */
       MPI_Gatherv (&position, 1, MPI_INT, counts, MPI_INT, root,
comm);
       /* собрать все упакованные сообщения */
       displs[0] = 0;
       for (i=1; i<gsize; i++)
              displs[i] = displs[i-1] + counts[i-1];
       totalcount = displs[gsize-1] + counts[gsize-1];
       rbuf = (char *)malloc (totalcount);
       rcuf = (char *)malloc (totalcount);
       MPI_Gatherv (&buf, position, MPI_PACKED, &rbuf, counts,
displs, MPI_PACKED, root, comm);
             /* распаковать все сообщения и соединить строки */
       for (i=0; i<gsize; i++)
              {
              position = 0;
              concat_pos = 0;
              MPI_Unpack (&rbuf+displs[i], totalcount-displs[i],
                     position, count, 1, MPI_INT, comm);
              MPI_Unpack (&rbuf+displs[i], totalcount-displs[i],
                     position, &cbuf+concat_pos, count, MPI_CHAR, comm);
              }
       concat_pos += count;
       cbuf[count] = '\0';
       }
```