МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

С. Б. Попов

Библиотека MPI

Учебное пособие

Самара

2011

Автор: ПОПОВ Сергей Борисович

Учебное пособие содержат изложение лекционного материала темы "Библиотека MPI" по курсу «Параллельное программирование» и предназначено для бакалавров четвертого курса факультета информатики направления 010400.62 «Прикладная математика и информатика».

СОДЕРЖАНИЕ

Краткая характеристика библиотеки MPI	4
MPI для языка FORTRAN	8
Общие процедуры MPI	9
Организация приема/передачи данных между отдельными процессами	13
Коллективные функции	30
Коммуникаторы, группы и области связи	36

Краткая характеристика библиотеки MPI

MPI (*message passing interface*) - библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

MPI предоставляет программисту единый механизм взаимодействия процессов внутри параллельного приложения независимо от машинной архитектуры (однопроцессорные или многопроцессорные с общей или локальной памятью), взаимного расположения ветвей (на одном процессоре или на разных) и API (*applications programmers interface*, т.е. интерфейс разработчика приложений) операционной системы. Программа, использующая MPI, легче отлаживается (сужается простор для совершения стереотипных ошибок при параллельном программировании) и быстрее переносится на другие платформы (в идеале, простой перекомпиляцией).

Минимально в состав MPI входят: библиотека программирования (заголовочные и библиотечные файлы для языков C, C++ и FORTRAN) и загрузчик приложений. Дополнительно включаются: профилирующий вариант библиотеки (используется на стадии тестирования параллельного приложения для определения оптимальности распараллеливания); загрузчик с графическим и сетевым интерфейсом для *X-Window* и проч.

Структура каталогов MPI привычна пользователям-программистам: *bin, include, lib, man, src, ...* Минимальный набор функций прост в освоении и позволяет быстро написать надежно работающую программу. Использование же всего арсенала функций MPI позволит получить быстроработающую программу при сохранении надежности.

Несколько компьютеров могут взаимодействовать друг с другом одним из трех способов:

1. через общую память, в *Unix* существуют средства для работы с ней (разные для *BSD*- и *ATT*- клонов *Unix*);
2. через скоростную внутримашинную сеть многопроцессорных вычислительных систем;
3. через сеть, как правило, работающую по протоколу *TCP/IP*.

Программный инструментарий MPI разрабатывался как стандарт для многопроцессорных ВС, объединенных скоростной внутримашинной сетью, но MPI может работать также на базе любого из трех способов соединений. Тем не менее, первая редакция MPI стандартом не стала в силу следующего ограничения: все процессы, сообщающиеся между собой посредством функций MPI, начинают и заканчивают свое выполнение одновременно. Это

не мешает использовать MPI как скелет для параллельных приложений, но системы массового обслуживания (клиент-серверные приложения и проч.) приходится разрабатывать на базе старого инструментария.

Проблема устранена в MPI-2: он гарантировано станет стандартом для систем, взаимодействующих по типу 2. Не исключено, что системы 1 и 3 типов со временем потеряют интерес для программистов и будут использоваться только как интерфейс между аппаратурой и MPI-2.

Существует точка зрения, что низкоуровневые пересылки через разделяемую память и семафоры предпочтительнее применения таких библиотек, как MPI, потому что работают быстрее. На это можно привести следующие возражения:

1. В хорошо распараллеленном приложении на собственно взаимодействие между ветвями (пересылки данных и синхронизацию) тратится небольшая доля времени – несколько процентов от общего времени работы. Таким образом, замедление пересылок, например, в два раза, не означает общего падения производительности вдвое – она понизится на несколько процентов. Зачастую такое понижение производительности является приемлемым и с лихвой оправдывается прочими соображениями.
2. MPI – это изначально быстрый инструмент. Для повышения скорости в нем используются приемы, о которых прикладные программисты зачастую просто не задумываются. Например:
   - ✧ встроенная буферизация позволяет избежать задержек при отправке данных – управление в передающую ветвь возвращается немедленно, даже если ветвь-получатель еще не подготовилась к приему;
   - ✧ MPI использует многопоточность (*multi-threading*), вынося большую часть своей работы в потоки *(threads*) с низким приоритетом; буферизация и многопоточность сводят к минимуму негативное влияние неизбежных простоев при пересылках на производительность прикладной программы;
   - ✧ на передачу данных типа "один-всем" затрачивается время, пропорциональное не числу участвующих ветвей , а логарифму этого числа.
3. Перенос программ на другие платформы не требует переписывания и повторной отладки. Незаменимое качество для программы, которой предстоит пользоваться широкому кругу людей. Следует учитывать и то обстоятельство, что уже появляются машины, на которых из средств межпрограммного взаимодействия есть только MPI.
4. MPI не страхует ни от одной из типовых ошибок, допускаемых при параллельном программировании, но использование этой библиотеки уменьшает их вероятность. Функции работы с разделяемой памятью и с семафорами слишком элементарны, примитивны. Вероятность без ошибки реализовать с их помощью нужное программе действие

стремительно уменьшается с ростом количества инструкций; вероятность найти ошибку путем отладки близка к нулю, потому что отладочное средство вносит задержку в выполнение одних ветвей и тем самым позволяет нормально работать другим (ветви перестают конкурировать за совместно используемые данные).

Документация

Полезно распечатать для справочных целей заголовочные файлы библиотеки из подкаталога *include*. В *Convex MPI* такой файл один: ***mpi.h***, в *WinMPICH* их три: ***mpi.h*** (описания констант и типов), ***mpi_errno.h*** (коды ошибок и их описания) и ***binding.h*** (прототипы функций). В случае для *Windows* печатать рекомендуется не оригиналы, а копии, из которых вычищены все детали, относящиеся к реализации (например, `_declspec(dllimport)` и т.д.). Далее по тексту не приводятся прототипы функций, описания переменных, типов и констант.

Manual pages – еще одно хорошее подспорье. Более детальную информацию: спецификацию, учебники и различные реализации MPI можно найти на сервере *NetLib* (*http://www.netlib.org/mpi*). Здесь следует особо отметить книгу *MPI: The complete reference* издательства *MIT Press*, имеющуюся на этом сайте.

Соглашения о терминах

Параллельное приложение состоит из нескольких *процессов (ветвей, потоков, задач)*, выполняющихся одновременно. Разные процессы могут выполняться как на разных процессорах, так и на одном и том же – для программы это роли не играет, поскольку в обоих случаях механизм обмена данными одинаков. С каждым потоком связывается *номер процесса* - целое неотрицательное число, являющееся уникальным атрибутом процесса.

Процессы обмениваются друг с другом данными в виде *сообщений*. Атрибутами сообщения являются номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура *MPI_Status*, содержащая три поля: *MPI_Source* (номер процесса отправителя), *MPI_Tag* (идентификатор сообщения), *MPI_Error* (код ошибки); могут быть и дополнительные поля.

Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767. Идентификаторы позволяют программе и библиотеке связи отличать сообщения друг от друга.

Для совместного проведения тех или иных расчетов процессы внутри приложения объединяются в *группы*. Группы могут быть вложенными. Внутри группы все процессы пронумерованы. Каждый процесс может узнать у библиотеки связи свой номер внутри группы и в зависимости от номера выполнить соответствующую часть расчетов. С каждой группой ассоциирован свой *коммуникатор*. Поэтому при осуществлении пересылки

необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором *MPI_COMM_WORLD*.

Термин "*процесс*" используется также в *Unix*, и здесь нет путаницы: в MPI процесс запускается и работает как обычный процесс *Unix*, связанный через MPI с остальными процессами, входящими в приложение. В остальном процессы следует считать изолированными друг от друга: у них разные области кода, стека и данных (см. описание *Unix*-процессов).

Категории функций: блокирующие, локальные, коллективные

Блокирующие – останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. Неблокирующие функции возвращают управление немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Неблокирующие функции возвращают идентификатор операции посылки-приема сообщения (*request*), с помощью которого позднее можно проверить завершение операции. До этого завершения с переменными и массивами, которые были аргументами неблокирующей функции, ничего делать нельзя.

Локальные – не инициируют пересылок данных между ветвями. Большинство информационных функций является локальными, т.к. копии системных данных уже хранятся в каждой ветви. Функция передачи *MPI_Send* и функция синхронизации *MPI_Barrier* не являются локальными, поскольку производят пересылку. Следует заметить, что, к примеру, функция приема *MPI_Recv* (парная для *MPI_Send*) является локальной: она всего лишь пассивно ждет поступления данных, ничего не пытаясь сообщить другим ветвям.

Коллективные – должны быть вызваны всеми ветвями-абонентами того коммуникатора, который передается им в качестве аргумента. Несоблюдение этого правила приводит к ошибкам на стадии выполнения программы.

Принятая в MPI нотация записи

Регистр букв: важен в C, не играет роли в FORTRAN.

Все идентификаторы начинаются с префикса *MPI_*. Это правило без исключений. Не рекомендуется заводить пользовательские идентификаторы, начинающиеся с этой приставки, а также с приставок *MPID_*, *MPIR_* и *PMPI_*, которые используются в служебных целях.

Если идентификатор сконструирован из нескольких слов, слова в нем, как правило, разделяются подчерками: *MPI_Get_count*, *MPI_Comm_rank*. Иногда разделитель не используется: *MPI_Sendrecv*, *MPI_Alltoall*.

Порядок слов в составном идентификаторе выбирается по принципу "от общего к частному": сначала префикс *MPI_*, потом название категории (*Type*, *Comm*, *Group*, *Attr*, *Errhandler* и т.д.), потом название операции (*MPI_Errhandler_create*, *MPI_Errhandler_set*, ...). Наиболее часто

употребляемые функции выпадают из этой схемы: они имеют "антиметодические", но короткие и стереотипные названия, например *MPI_Barrier*, или *MPI_Unpack*.

Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами: *MPI_COMM_WORLD*, *MPI_FLOAT*. В именах функций первая за префиксом буква - заглавная, остальные маленькие: *MPI_Send*, *MPI_Comm_size*.

MPI для языка FORTRAN

Изложим в сжатом виде основные отличия и особенности реализации MPI для языка FORTRAN. С их учетом программисты, пишущие на языке FORTRAN, смогут приступить к изучению дальнейшего материала.

- ✧ Регистр символов в исходном тексте для FORTRAN значения не имеет. *MPI_Comm_rank* и *MPI_COMM_RANK* для него, в отличие от С, является одним и тем же идентификатором.
- ✧ В начале исходного текста должна быть строка

```
include 'mpif.h'
```

Этот файл содержит описания переменных и констант.

- ✧ Все, что в С является функциями, в FORTRAN сделано подпрограммами. В С код ошибки MPI возвращается пользовательской программе как код завершения функции; в FORTRAN код ошибки возвращается в дополнительном параметре:

С:
```
errcode = MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

FORTRAN:
```
      CALL MPI_COMM_RANK ( MPI_COMM_WORLD, rank, ierror )
```

- ✧ Все параметры в FORTRAN передаются не по значению, а по ссылке. Соответственно, там, где в С используется символ "&" для вычисления адреса переменной, в FORTRAN не надо писать ничего. Это иллюстрирует пример для предыдущего пункта, где в переменную *rank* функция/подпрограмма записывает номер вызвавшей ее задачи. Там, где MPI требуется знать местонахождение в памяти таких данных, которые по ссылке не передашь (например, при конструировании пользовательских типов), используется подпрограмма *MPI_ADDRESS*.
- ✧ В FORTRAN нет структур. Там, где в С для описания данных в MPI применяется структура, в FORTRAN применяется целочисленный массив, размер и номера ячеек которого описаны символьными константами.

В С пишем:

```
MPI_Status status;  /* переменная типа структура */
```

```
MPI_Probe( MPI_ANY_SOURCE,
           MPI_ANY_TAG,
           MPI_COMM_WORLD,
           &status );
/* теперь проверяем поля заполненной структуры */
if( status.MPI_SOURCE == 1 )     { ... }
if( status.MPI_TAG == tagMsg1 )     { ... }
```

В FORTRAN, соответственно:

```
INTEGER status(MPI_STATUS_SIZE)
CALL MPI_PROBE( MPI_ANY_SOURCE,
                MPI_ANY_TAG,
                MPI_COMM_WORLD,
                status )
IF( status(MPI_SOURCE) .EQ. 1 ) ....
IF( status(MPI_TAG) .EQ. tagMsg1 ) ...
```

и так далее...

Имя типа-структуры переходит в размер массива, дополняясь суффиксом *_SIZE*, а имена полей переходят в индексы ячеек в этом целочисленном массиве.

✧ Для компиляции и компоновки вместо команды `mpicc` используются команды `mpif77` или `mpif90` для FORTRAN77 и FORTRAN90 соответственно. Это скрипты (командные файлы), которые после соответствующей настройки окружения (пути к библиотекам и т.п.) вызывают стандартные компиляторы FORTRAN.

Общие процедуры MPI

Существует несколько функций, которые используются в любом, даже самом коротком приложении MPI. Занимаются они не столько собственно передачей данных, сколько ее обеспечением.

MPI_Init - инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова *MPI_Init*.

```
int MPI_Init( int* argc, char*** argv);
```

Функции передаются адреса аргументов, стандартно получаемых программой от операционной системы и хранящих параметры командной строки. В конец командной строки программы MPI-загрузчик mpirun добавляет ряд информационных параметров, которые требуются *MPI_Init*. Это показывается в примере 3.1.

Возвращает: в случае успешного выполнения - *MPI_SUCCESS*, иначе - код ошибки. (То же самое возвращают и все остальные функции, рассматриваемые в данном пособии.)

MPI_Finalize - завершение параллельной части приложения. Все последующие обращения к любым MPI-процедурам, в том числе к *MPI_Init*, запрещены. К моменту вызова *MPI_Finalize* некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

```
int MPI_Finalize( void );
```

MPI_Abort – аварийное закрытие библиотеки. Вызывается, если пользовательская программа завершается по причине ошибок времени выполнения, связанных с MPI:

```
void MPI_Abort(MPI_Comm comm, int error );
```

Вызов MPI_Abort из любой задачи принудительно завершает работу всех задач, подсоединенных к области связи *comm*. Если указан описатель *MPI_COMM_WORLD*, будет завершено все приложение (все его задачи) целиком, что, по-видимому, и является наиболее правильным решением. Используйте код ошибки *MPI_ERR_OTHER*, если не знаете, как охарактеризовать ошибку в классификации MPI.

Две следующие информационных функции: сообщают размер группы (то есть общее количество задач, подсоединенных к ее области связи) и порядковый номер вызывающей задачи:

```
int MPI_Comm_size( MPI_Comm comm, int* size);
```

Определение общего числа параллельных процессов в группе *comm*. Параметры: *comm* - идентификатор группы; *size* - размер группы (выходной параметр).

```
int MPI_Comm_rank( MPI_comm comm, int* rank);
```

Определение номера процесса в группе *comm*. Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от 0 до *size_of_group-1*. Параметры: *comm* – идентификатор группы; *rank* – номер вызывающего процесса в группе *comm* (выходной параметр).

```
double MPI_Wtime(void)
```

Функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Гарантируется, что этот момент не будет изменен за время существования процесса.

Использование *MPI_Init*, *MPI_Finalize*, *MPI_Comm_size*, *MPI_Comm_rank* и *MPI_Abort* показано далее, в примере 1.

Пример 1

```
/*
 *  Простейшая приемопередача:
 *     MPI_Send, MPI_Recv
 *  Завершение по ошибке:
 *     MPI_Abort
 */

#include <mpi.h>
#include <stdio.h>
```

```
/* Идентификаторы сообщений */
#define tagFloatData  1
#define tagDoubleData 2

/* Этот макрос введен для удобства,
 * он позволяет указывать длину массива в количестве я чеек
 */
#define ELEMS(x)   ( sizeof(x) / sizeof(x[0]) )

int main( int argc, char **argv )
{
    int size, rank, count;
    float floatData[10];
    double doubleData[20];
    MPI_Status status;

/* Инициализируем библиотеку */
    MPI_Init( &argc, &argv );
    /* Узнаем количество задач в запущенном приложении */
    MPI_Comm_size( MPI_COMM_WORLD, &size );
/* ... и свой собственный номер: от 0 до (size-1) */
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
/* пользователь должен запустить ровно две задачи,
   иначе ошибка */
    if( size != 2 ) {
/* задача с номером 0 сообщает пользователю об ошибке */
        if( rank==0 )
            printf("Error: two processes required",
                   "instead of %d, abort\n", size );
/* Все задачи-абоненты области связи MPI_COMM_WORLD
 * будут стоять, пока задача 0 не выведет сообщение.
 */
        MPI_Barrier( MPI_COMM_WORLD );
/* Без точки синхронизации одна из задач может
 * вызвать MPI_Abort раньше, чем успеет отработать printf()
 * в задаче 0, MPI_Abort немедленно принудительно завершит
 * все задачи и сообщение выведено не будет
 */
/* все задачи аварийно завершают работу */
        MPI_Abort(
            MPI_COMM_WORLD,  /* Описатель области связи, на
                 которую распространяется действие ошибки */
            MPI_ERR_OTHER ); /* Целочисленный код ошибки */
        return -1;
    }

    if( rank==0 ) {
      /* Задача 0 что-то такое передает задаче 1 */
```

```
        MPI_Send(
          floatData,  /* 1) адрес передаваемого массива */
          5,          /* 2) сколько: 5 ячеек, т.е.
                            floatData[0]..floatData[4] */
          MPI_FLOAT,  /* 3) тип ячеек */
          1,          /* 4) кому: задаче 1 */
          tagFloatData,     /* 5) идентификатор сообщения */
          MPI_COMM_WORLD ); /* 6) описатель области связи,
                               через которую
                               происходит передача */
/* и еще одна передача: данные другого типа */
        MPI_Send( doubleData, 6, MPI_DOUBLE, 1,
                  tagDoubleData, MPI_COMM_WORLD );

    } else {
/* Задача 1 что-то такое принимает от задачи 0 */
/* ждем сообщение и помещаем пришедшие данные в буфер */
        MPI_Recv(
            doubleData,     /* 1) адрес массива, куда
                               складывать принятое */
            ELEMS( doubleData ), /* 2) фактическая длина
                               приемного массива в
                               числе ячеек */
            MPI_DOUBLE,     /* 3) сообщаем MPI, что
                               пришедшее сообщение
                               состоит из чисел типа
                               'double' */
            0,              /* 4) от кого: от задачи 0 */
            tagDoubleData,  /* 5) ожидаем сообщение с таким
                               идентификатором */
            MPI_COMM_WORLD, /* 6) описатель области связи,
                               через которую ожидается
                               приход сообщения */
            &status );      /* 7) сюда будет записан статус
                               завершения приема */
/* Вычисляем фактически принятое количество данных */
        MPI_Get_count(
            &status,     /* статус завершения */
            MPI_DOUBLE,  /* сообщаем MPI, что пришедшее
                            сообщение состоит из чисел
                            типа 'double' */
            &count );    /* сюда будет записан результат */
/* Выводим фактическую длину принятого на экран */
        printf("Received %d elems\n", count );
/* Аналогично принимаем сообщение с данными типа float
 * Обратите внимание: задача-приемник имеет возможность
 * принимать сообщения не в том порядке, в котором они
 * отправлялись, если эти сообщения имеют разные
 * идентификаторы
 */
```

```
        MPI_Recv( floatData, ELEMS( floatData ), MPI_FLOAT,
            0, tagFloatData, MPI_COMM_WORLD, &status );
        MPI_Get_count( &status, MPI_FLOAT, &count );
    }
/* Обе задачи завершают выполнение */
    MPI_Finalize();
    return 0;
    }
```

Организация приема/передачи данных между отдельными процессами

Самый простой тип связи между задачами: одна ветвь вызывает функцию передачи данных, а другая - функцию приема. В MPI это выглядит, например, так:
Задача 1 передает:

```
int buf[10];
MPI_Send( buf, 5, MPI_INT, 1, 0, MPI_COMM_WORLD );
```

Задача 2 принимает:

```
int buf[10];
MPI_Status status;
MPI_Recv( buf, 10, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
```

Проще всего это осуществляется функциями приема/передачи сообщений с блокировкой:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int msgtag, MPI_Comm comm);
```

Параметры: *buf* - адрес начала буфера посылки сообщения;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
dest - номер процесса-получателя;
msgtag - идентификатор сообщения;
comm - идентификатор группы.

Блокирующая посылка сообщения с идентификатором *msgtag*, состоящего из *count* элементов типа *datatype*, процессу с номером *dest*. Все элементы сообщения расположены подряд в буфере *buf*. Значение *count* может быть нулем, задается не в байтах, а в количестве ячеек. Тип передаваемых элементов *datatype* должен указываться с помощью предопределенных констант типа. Разрешается передавать сообщение самому себе.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу *dest*, остается за MPI. Следует специально отметить, что возврат из подпрограммы *MPI_Send* не означает ни того, что сообщение уже передано процессу *dest*, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, выполнивший *MPI_Send*.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int msgtag, MPI_comm comm,
             MPI_Status *status);
```

Параметры:

buf - адрес начала буфера приема сообщения (выходной параметр);
count - максимальное число элементов в принимаемом сообщении;
datatype - тип элементов принимаемого сообщения;
source - номер процесса-отправителя;
msgtag - идентификатор принимаемого сообщения;
comm - идентификатор группы;
status - параметры принятого сообщения (выходной параметр).

Прием сообщения с идентификатором *msgtag* от процесса *source* с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой *MPI_Probe*.

Блокировка гарантирует, что после возврата из подпрограммы все элементы сообщения приняты и расположены в буфере *buf*.

Если процесс посылает два сообщения другому процессу и оба эти сообщения соответствуют одному и тому же вызову *MPI_Recv*, то первым будет принято то сообщение, которое было отправлено раньше.

В качестве номера процесса-отправителя можно указать предопределенную константу *MPI_ANY_SOURCE* - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу *MPI_ANY_TAG* - признак того, что подходит сообщение с любым идентификатором. После такого приема данных фактические номер процесса и идентификатор сообщения записываются в поля *MPI_SOURCE* и *MPI_TAG* из структуры *status*.

Поле *MPI_ERROR*, как правило, проверять необязательно – обработчик ошибок, устанавливаемый MPI по умолчанию, в случае сбоя завершит выполнение программы до возврата из *MPI_Recv*. Таким образом, после возврата из *MPI_Recv* поле *status.MPI_ERROR* может быть равно только 0 (или, если угодно, *MPI_SUCCESS*).

Тип *MPI_Status* не содержит поля, в которое записывалась бы фактическая длина пришедшего сообщения. Длину можно узнать так:

```
MPI_Status status;
int count;
MPI_Recv( ... ,  MPI_INT, ... , &status );
MPI_Get_count( &status, MPI_INT, &count );
/* ... теперь count содержит количество принятых ячеек */
```

```
int MPI_Get_Count( MPI_Status *status,
                   MPI_Datatype datatype, int *count);
```

Параметры: *status* - параметры принятого сообщения;
datatype - тип элементов принятого сообщения;
count - число элементов сообщения (выходной параметр).

По значению параметра *status* данная подпрограмма определяет число уже принятых (после обращения к *MPI_Recv*) или принимаемых (после обращения к *MPI_Probe* или *MPI_IProbe*) элементов сообщения типа *datatype*.

Обратите внимание, что аргумент-описатель типа у *MPI_Recv* и *MPI_Get_count* должен быть одинаковым, иначе, в зависимости от реализации: в *count* вернется неверное значение или произойдет ошибка времени выполнения.

Итак, по возвращении из *MPI_Recv* поля структуры *status* содержат информацию о принятом сообщении, а функция *MPI_Get_count* возвращает количество фактически принятых данных. Однако имеется еще одна функция, которая позволяет узнать о характеристиках сообщения до того, как сообщение будет помещено в приемный пользовательский буфер: *MPI_Probe*. За исключением адреса и размера пользовательского буфера, она имеет такие же параметры, как и *MPI_Recv*. Она возвращает заполненную структуру *MPI_Status* и после нее можно вызвать *MPI_Get_count*. Стандарт MPI гарантирует, что следующий за *MPI_Probe* вызов *MPI_Recv* с теми же параметрами (имеются в виду номер задачи-передатчика, идентификатор сообщения и коммуникатор) поместит в буфер пользователя именно то сообщение, которое было принято функцией *MPI_Probe*.

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

Параметры:

source - номер процесса-отправителя или *MPI_ANY_SOURCE;*
msgtag - идентификатор ожидаемого сообщения или *MPI_ANY_TAG;*
comm - идентификатор группы;
status - параметры обнаруженного сообщения (выходной параметр).

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из подпрограммы не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра *status*. Следует обратить внимание, что подпрограмма определяет только факт прихода сообщения, но реально его не принимает.

MPI_Probe нужна в двух случаях:

✧ Когда задача-приемник не знает заранее длины ожидаемого сообщения. Пользовательский буфер заводится в динамической памяти:

```
MPI_Probe( MPI_ANY_SOURCE, tagMessageInt,
           MPI_COMM_WORLD, &status );
/* MPI_Probe вернет управление после того как примет */
/* данные в системный буфер */
MPI_Get_count( &status, MPI_INT, &bufElems );
buf = malloc( sizeof(int) * bufElems );
MPI_Recv( buf, bufElems, MPI_INT, ...
/* дальше параметры у MPI_Recv такие же, как в MPI_Probe */
        );
 /* MPI_Recv останется просто скопировать */
 /* данные из системного буфера в пользовательский */
```

Вместо этого, конечно, можно просто завести на приемной стороне буфер заведомо большой, чтобы вместить в себя самое длинное из возможных сообщений, но такой стиль не является оптимальным, если длина сообщений изменяется в слишком широких пределах.

Когда задача-приемник собирает сообщения от разных отправителей с содержимым разных типов. Без *MPI_Probe* порядок извлечения сообщений в буфер пользователя должен быть задан в момент компиляции:

```
MPI_Recv(floatBuf, floatBufSize, MPI_FLOAT, MPI_ANY_SOURCE,
         tagFloatData, ... );
MPI_Recv(intBuf,   intBufSize,   MPI_INT,   MPI_ANY_SOURCE,
         tagIntData,   ... );
MPI_Recv(charBuf,  charBufSize,  MPI_CHAR,  MPI_ANY_SOURCE,
         tagCharData,  ... );
```

Теперь, если в момент выполнения сообщение с идентификатором *tagCharData* придет раньше двух остальных, MPI будет вынужден "законсервировать" его на время выполнения первых двух вызовов *MPI_Recv*. Это чревато непроизводительными расходами памяти. *MPI_Probe* позволит задать порядок извлечения сообщений в буфер пользователя равным по рядку их поступления на принимающую сторону, делая это не в момент компиляции, а непосредственно в момент выполнения:

```
for( i=0; i<3; i++ ) {
    MPI_Probe(
MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status );
    switch( status.MPI_TAG ) {
      case tagFloatData:
         MPI_Recv( floatBuf, floatBufSize, MPI_FLOAT, ... );
         break;
      case tagIntData:
         MPI_Recv( intBuf, intBufSize, MPI_INT, ... );
         break;
      case tagCharData:
         MPI_Recv( charBuf, charBufSize, MPI_CHAR, ... );
         break;
    } /* конец switch */
} /* конец for */
```

Многоточия здесь означают, что последние 4 параметра у *MPI_Recv* такие же, как и у предшествующей им *MPI_Probe*.

Использование *MPI_Probe* продемонстрировано в примере 2.

Пример 2

```
/*
 *  Прием сообщений неизвестной длины:
 *      MPI_Probe
 *
 *  Прием сообщений от разных отправи телей с разными
 *  идентификаторами
 *   ( с содержимым разных типов ) в произвольном порядке:
 *      MPI_ANY_SOURCE, MPI_ANY_TAG      ( джокеры )
 */

#include <mpi.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


/* Идентификаторы сообще ний */
#define tagFloatData  1
#define tagLongData   2

  /* Длина передаваемых сообщений может быть
   * случайной от 1 до maxMessageElems
   */
#define maxMessageElems 100
```

```
int main( int argc, char **argv )
{
    int    size, rank, count, i, n, ok;
    float *floatPtr;
    int   *longPtr;
    char  *typeName;
    MPI_Status status;

/* Инициализация и сообщение об ошибке
 * целиком перенесены из первого примера
 */
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

/* пользователь должен запустить ровно ТРИ задачи,
   иначе ошибка */
    if( size != 3 ) {
        if( rank==0 )
            printf("Error: 3 processes required ",
                   "instead of %d\n", size );
        MPI_Barrier( MPI_COMM_WORLD );
        MPI_Abort( MPI_COMM_WORLD, MPI_ERR_OTHER );
        return -1;
    }

/* Каждая задача инициализирует генератор случайных чисел */
    srand( ( rank + 1 ) * (unsigned )time(0) );

    switch( rank ) {

        case 0:
              /* создаем сообщение случайной длины */
            count = 1 + rand() % maxMessageElems;
            floatPtr = malloc( count * sizeof(float) );
            for( i=0; i<count; i++ )
                floatPtr[i] = (float)i;

/* Посылаем сообщение в задачу 2 */
            MPI_Send( floatPtr, count, MPI_FLOAT,
                2, tagFloatData, MPI_COMM_WORLD );
            printf("%d. Send %d float items to process 2\n",
                rank, count );
            break;

        case 1:
              /* создаем сообщение случайной длины */
            count = 1 + rand() % maxMessageElems;
            longPtr = malloc( count * sizeof(long) );
```

```
            for( i=0; i<count; i++ )
                longPtr[i] = i;

              /* Посылаем сообщение в задачу 2 */
            MPI_Send( longPtr, count, MPI_LONG,
                      2, tagLongData, MPI_COMM_WORLD );
            printf("%d. Send %d long items to process 2\n",
                   rank, count );
            break;

        case 2:
            /* задача 2 принимает сообщения неизвестной
             * длины,используя MPI_Probe
             */
            for( n=0; n<2; n++ )  /* Всего ожидаются два*/
            {                     /* сообщения */
                MPI_Probe(
                    MPI_ANY_SOURCE,  /* Джокер: ждем от
                                        любой задачи */
                    MPI_ANY_TAG,     /* Джокер: ждем с любым
                                        идентификатором */
                    MPI_COMM_WORLD,
                    &status );
/* MPI_Probe вернет управление, когда сообщение б удет
 * уже на приемной стороне в служебном буфере
 */

/* Проверяем идентификатор и размер пришедшего сообщения */
                if( status.MPI_TAG == tagFloatData )
                {
                    MPI_Get_count( &status, MPI_FLOAT,
                                   &count );
/* Принятое будет размещено в динамической пам яти:
 * заказываем в ней буфер соответствующей дл ины
 */
                    floatPtr=malloc(count * sizeof(float));
                    MPI_Recv( floatPtr, count, MPI_FLOAT,
                              MPI_ANY_SOURCE, MPI_ANY_TAG,
                              MPI_COMM_WORLD, &status );
/* MPI_Recv просто скопирует уже принятые да нные
 * из системного буфера в пользовательский
 */
                    /* Проверяем принятое */
                    for( ok=1, i=0; i<count; i++ )
                        if( floatPtr[i] != (float)i )
                            ok = 0;
                    typeName = "float";
                }
                else if( status.MPI_TAG == tagLongData )
                {
```

```
                        /* Действия, аналогичные вышеописанным
                         */
                        MPI_Get_count(&status,MPI_LONG, &count);
                        longPtr = malloc(count * sizeof(long));

                        MPI_Recv( longPtr, count, MPI_LONG,
                            MPI_ANY_SOURCE, MPI_ANY_TAG,
                            MPI_COMM_WORLD, &status );
                        for( ok=1, i=0; i<count; i++ )
                            if( longPtr[i] != i )
                                ok = 0;
                        typeName = "long";
                    }

/* Докладываем о завершении приема */
                    printf( "%d. %d %s items are received ",
                            "from %d : %s\n", rank, count,
                            typeName,status.MPI_SOURCE,
                            (ok ? "OK" : "FAILED") );

                } /* for(n) */
                break;

        } /* switch(rank) */

         /* Завершение работы */
        MPI_Finalize();
        return 0;
}
```

В этом примере используются параметры *MPI_ANY_SOURCE* для номера задачи-отправителя ("принимай от кого угодно") и *MPI_ANY_TAG* для идентификатора получаемого сообщения ("принимай что угодно"). Пользоваться ими следует с осторожностью, потому что по ошибке таким вызовом *MPI_Recv* может быть захвачено сообщение, которое должно приниматься в другой части задачи-получателя.

Если логика программы достаточно сложна, использовать такие параметры-джокеры желательно только в функциях *MPI_Probe* и *MPI_Iprobe*, чтобы перед фактическим приемом узнать тип и количество данных в поступившем сообщении (на худой конец, можно принимать, и не зная количества - был бы приемный буфер достаточно вместительным, но тип для *MPI_Recv* надо указывать явно – а он может быть разным в сообщениях с разными идентификаторами).

Достоинство джокеров: приходящие сообщения извлекаются по мере поступления, а не по мере вызова *MPI_Recv* с нужными идентификаторами задач/сообщений. Это экономит память и увеличивает скорость работы.

Кратко опишем основные функции приема/передачи сообщений без блокировки:

```
int MPI_ISend(void *buf, int count, MPI_Datatype datatype,
              int dest, int msgtag, MPI_Comm comm,
              MPI_Request *request);
```

Параметры:

buf - адрес начала буфера посылки сообщения;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
dest - номер процесса-получателя;
msgtag - идентификатор сообщения;
comm - идентификатор группы;
request - идентификатор асинхронной передачи (выходной параметр).

Передача сообщения, аналогичная *MPI_Send*, однако возврат из подпрограммы происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере *buf*. Это означает, что нельзя повторно использовать данный буфер для других целей без получения дополнительной информации о завершении данной посылки. Окончание процесса передачи (т.е. того момента, когда можно переиспользовать буфер *buf* без опасения испортить передаваемое сообщение) можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

Сообщение, отправленное любой из процедур *MPI_Send* и *MPI_ISend*, может быть принято любой из процедур *MPI_Recv* и *MPI_IRecv*.

```
int MPI_IRecv(void *buf, int count, MPI_Datatype datatype,
              int source, int msgtag, MPI_comm comm,
              MPI_Request *request);
```

Параметры:

buf - адрес начала буфера приема сообщения (выходной параметр)
count - максимальное число элементов в принимаемом сообщении
datatype - тип элементов принимаемого сообщения
source - номер процесса-отправителя
msgtag - идентификатор принимаемого сообщения
comm - идентификатор группы
request - идентификатор асинхронного приема сообщения (выходной)

Прием сообщения, аналогичный *MPI_Recv*, однако возврат из подпрограммы происходит сразу после инициализации процесса приема без ожидания получения сообщения в буфере *buf*. Окончание процесса приема можно определить с помощью параметра *request* и процедур *MPI_Wait* и *MPI_Test*.

```
int MPI_Wait( MPI_Request *request, MPI_Status *status);
```

Параметры:

request - идентификатор асинхронного приема или передачи;
status - параметры сообщения (выходной параметр).

Ожидание завершения асинхронных процедур *MPI_ISend* или *MPI_IRecv*, ассоциированных с идентификатором *request*. В случае приема, атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*.

```
int MPI_WaitAll( int count, MPI_Request *requests,
                 MPI_Status *statuses);
```

Параметры:

count - число идентификаторов;
requests - массив идентификаторов асинхронного приема/передачи;
statuses - параметры сообщений (выходной параметр).

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива *statuses* будет установлено в соответствующее значение.

```
int MPI_WaitAny( int count, MPI_Request *requests,
                 int *index, MPI_Status *status);
```

Параметры:

count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
index - номер завершенной операции обмена (выходной параметр)
status - параметры сообщений (выходной параметр)

Выполнение процесса блокируется до тех пор, пока какая-либо операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если несколько операций могут быть завершены, то случайным образом выбирается одна из них. Параметр *index* содержит номер элемента в массиве *requests*, содержащего идентификатор завершенной операции.

```
int MPI_WaitSome( int incount, MPI_Request *requests,
                  int *outcount, int *indexes,
                  MPI_Status *statuses);
```

Параметры:

incount - число идентификаторов
requests - массив идентификаторов асинхронного приема/передачи

outcount - число идентификаторов завершившихся операций обмена (выходной параметр)
indexes - массив номеров завершившихся операции обмена (выходной параметр)
statuses - параметры завершившихся сообщений (выходной параметр)

Выполнение процесса блокируется до тех пор, пока по крайней мере одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр *outcount* содержит число завершенных операций, а первые *outcount* элементов массива *indexes* содержат номера элементов массива *requests* с их идентификаторами. Первые *outcount* элементов массива *statuses* содержат параметры завершенных операций.

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

Параметры:
request - идентификатор асинхронного приема или передачи
flag - признак завершенности операции обмена (выходной параметр)
status - параметры сообщения (выходной параметр)

Проверка завершенности асинхронных процедур *MPI_ISend* или *MPI_IRecv*, ассоциированных с идентификатором *request*. В параметре *flag* возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра *status*.

```
int MPI_TestAll( int count, MPI_Request *requests,
                 int *flag, MPI_STatus *statuses);
```

Параметры:
count - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
flag - признак завершенности операций обмена (выходной параметр)
statuses - параметры сообщений (выходной параметр)

В параметре *flag* возвращает значение *1*, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве *statuses*). В противном случае возвращается *0*, а элементы массива *statuses* неопределены.

```
int MPI_TestAny(int count, MPI_Request *requests,
                int *index, int *flag, MPI_Status *status);
```

Параметры:
count - число идентификаторов

requests - массив идентификаторов асинхронного приема или передачи
index - номер завершенной операции обмена (выходной параметр)
flag - признак завершенности операции обмена (выходной параметр)
status - параметры сообщения (выходной параметр)

Если к моменту вызова подпрограммы хотя бы одна из операций обмена завершилась, то в параметре *flag* возвращается значение *1*, *index* содержит номер соответствующего элемента в массиве *requests*, а *status* - параметры сообщения.

```
int MPI_TestSome( int incount, MPI_Request *requests,
                  int *outcount, int *indexes,
                  MPI_Status *statuses);
```

Параметры:
incount - число идентификаторов
requests - массив идентификаторов асинхронного приема или передачи
outcount - число идентификаторов завершившихся операций обмена (выходной параметр)
indexes - массив номеров завершившихся операции обмена (выходной параметр)
statuses - параметры завершившихся операций (выходной параметр)

Данная подпрограмма работает так же, как и *MPI_TestSome*, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение *outcount* будет равно нулю.

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

Параметры:
source - номер процесса-отправителя или *MPI_ANY_SOURCE*
msgtag - идентификатор ожидаемого сообщения или *MPI_ANY_TAG*
comm - идентификатор группы
flag - признак завершенности операции обмена (выходной параметр)
status - параметры обнаруженного сообщения (выходной параметр)

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре *flag* возвращает значение *1*, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично *MPI_Probe*), и значение *0*, если сообщения с указанными атрибутами еще нет.

Объединение запросов на взаимодействие

Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и

перемещении необходимой информации между процессом и сетевым контроллером. Несколько запросов на прием и/или передачу могут объединяться вместе для того, чтобы далее их можно было бы запустить одной командой. Способ приема сообщения никак не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

```
int MPI_Send_Init( void *buf, int count,
                MPI_Datatype datatype, int dest, int msgtag,
                MPI_Comm comm, MPI_Request *request);
```

Параметры:

buf - адрес начала буфера посылки сообщения
count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
dest - номер процесса-получателя
msgtag - идентификатор сообщения
comm - идентификатор группы
request - идентификатор асинхронной передачи (выходной параметр)

Формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы *MPI_ISend*, однако в отличие от нее пересылка не начинается до вызова подпрограммы *MPI_StartAll*.

```
int MPI_Recv_Init( void *buf, int count,
                MPI_Datatype datatype, int source,
                int msgtag, MPI_Comm comm,
                MPI_Request *request);
```

Параметры:

buf - адрес начала буфера приема сообщения (выходной параметр)
count - число принимаемых элементов в сообщении
datatype - тип принимаемых элементов
source - номер процесса-отправителя
msgtag - идентификатор сообщения
comm - идентификатор группы
request - идентификатор асинхронного приема (выходной параметр)

Формирование запроса на выполнение приема данных. Все параметры точно такие же, как и у подпрограммы *MPI_IReceive*, однако в отличие от нее реальный прием не начинается до вызова подпрограммы *MPI_StartAll*.

```
MPI_Start_All( int count, MPI_Request *requests);
```

Параметры:

count - число запросов на взаимодействие
requests - массив идентификаторов приема/передачи (выходной)

Запуск всех отложенных взаимодействий, ассоциированных вызовами подпрограмм *MPI_Send_Init* и *MPI_Recv_Init* с элементами массива запросов *requests*. Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью процедур *MPI_Wait* и *MPI_Test*.

Совмещенные прием/передача сообщений

Некоторые конструкции с приемо-передачей применяются очень часто. Например, обмен данными с соседями по группе (в группе четное количество ветвей!):

```
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if( rank % 2 ) {
    /* Ветви с четными номерами сначала
     * передают следующим нечетным ветвям,
     * потом принимают от предыдущих
     */
    MPI_Send(..., ( rank+1 ) % size ,...);
    MPI_Recv(..., ( rank+size-1 ) % size ,...);
} else {
    /* Нечетные ветви поступают наоборот:
     * сначала принимают от предыдущих ветвей,
     * потом передают следующим.
     */
    MPI_Recv(..., ( rank-1 ) % size ,...);
    MPI_Send(..., ( rank+1 ) % size ,...);
}
```

Посылка данных и получение подтверждения:

```
MPI_Send(..., anyRank ,...);    /* Посылаем данные */
    MPI_Recv(..., anyRank ,...);    /* Принимаем
                                       подтверждение */
```

Ситуация настолько распространенная, что в MPI специально введены две функции, осуществляющие одновременно посылку одних данных и прием других. Первая из них - *MPI_Sendrecv*.

```
int MPI_Sendrecv( void *sbuf, int scount,
                  MPI_Datatype stype, int dest, int stag, void
                  *rbuf, int rcount,
```

```
MPI_Datatype rtype, int source, MPI_DAtatype
rtag, MPI_Comm comm, MPI_Status *status)
```

Параметры:

sbuf - адрес начала буфера посылки сообщения
scount - число передаваемых элементов в сообщении
stype - тип передаваемых элементов
dest - номер процесса-получателя
stag - идентификатор посылаемого сообщения
rbuf - адрес начала буфера приема сообщения (выходной параметр)
rcount - число принимаемых элементов сообщения
rtype - тип принимаемых элементов
source - номер процесса-отправителя
rtag - идентификатор принимаемого сообщения
comm - идентификатор группы
status - параметры принятого сообщения (выходной параметр)

Данная операция объединяет в едином запросе посылку и прием сообщений. Принимающий и отправляющий процессы могут являться одним и тем же процессом. Сообщение, отправленное операцией *MPI_Sendrecv*, может быть принято обычным образом, и точно также операция *MPI_Sendrecv* может принять сообщение, отправленное обычной операцией *MPI_Send*. Буфера приема и посылки обязательно должны быть различными.

Следует учесть, что и прием, и передача используют один и тот же коммуникатор ; порядок приема и передачи данных *MPI_Sendrecv* выбирает автоматически, при этом гарантируется, что автоматический выбор не приведет к дедлоку (взаимной блокировке различных процессов).

Другая функция *MPI_Sendrecv_replace* помимо общего коммуникатора использует еще и общий для приема-передачи буфер. Не очень удобно то, что параметр *count* получает двойное толкование: это и количество отправляемых данных, и предельная емкость входного буфера. Показания к применению:

- ✧ принимаемые данные должны быть заведомо не длиннее отправляемых
- ✧ принимаемые и отправляемые данные должны иметь одинаковый тип ;
- ✧ отправляемые данные затираются принимаемыми.

MPI_Sendrecv_replace так же гарантированно не вызывает дедлока.

Что такое дедлок? Это взаимная блокировка процессов при использовании разделяемых ресурсов.

Вариант 1:

```
-- Процесс 1 --            -- Процесс 2 --
Recv( из процесса 2 )      Recv( из процесса 1 )
Send( в процесс 2 )        Send( в процесс 1 )
```

Вариант 1 вызовет дедлок, какой бы инструментарий не использовался: функция приема не вернет управления до тех пор, пока не получит данные;

поэтому функция передачи не может приступить к отправке данных; поэтому функция приема... и так до самого SIG_KILL).

Вариант 2:

```
-- Процесс 1 --            -- Процесс 2 --
Send( в процесс 2 )        Send( в процесс 1 )
Recv( из процесса 2 )      Recv( из процесса 1 )
```

Вариант 2 заблокирует процессы в том случае, если функция передачи возвращает управление только после того, как данные попали в пользовательский буфер на приемной стороне. Скорее всего, именно так и возьмется реализовывать передачу через разделяемую память/семафоры программист-проблемщик.

Однако при использовании MPI зависания во втором варианте не произойдет. *MPI_Send*, если на приемной стороне нет готовности (не вызван *MPI_Recv*), не станет ее дожидаться, а положит данные во временный буфер и вернет управление программе немедленно. Когда *MPI_Recv* будет вызван, данные он получит не из пользовательского буфера напрямую, а из промежуточного системного. Буферизация – дело громоздкое, и не всегда сильно экономит время (особенно на SMP-машинах), зато повышает надежность: делает программу более устойчивой к ошибкам программиста.

MPI_Sendrecv и *MPI_Sendrecv_replace* также делают программу более устойчивой: с их использованием программист лишается возможности перепутать варианты 1 и 2.

Зачем MPI знать тип передаваемых данных?

Стандартные функции пересылки данных прекрасно обходятся без подобной информации – им требуется знать только размер в байтах. Вместо одного такого аргумента функции MPI получают два: количество элементов некоторого типа и символический описатель указанного типа (*MPI_INT*, и т.д., см. табл. 1). Причин тому несколько:

1. MPI позволяет описывать пользовательские типы данных, которые располагаются в памяти не непрерывно, а с разрывами, или наоборот, с перехлестом. Переменная такого типа характеризуется не только размером, и эти характеристики MPI хранит в описателе типа.
2. Приложение MPI может работать на гетерогенном вычислительном комплексе (коллективе ЭВМ с разной архитектурой). Одни и те же типы данных на разных машинах могут иметь разное представление, например: на плавающую арифметику существует 3 разных стандарта (IEEE, IBM, Cray); тип *char* в терминальных приложениях *Windows* представлен альтернативной кодировкой ГОСТ, а в *Unix* - кодировкой KOI-8r ; в многобайтовых числах на процессорах фирмы *intel* младший байт занимает младший адрес, на всех остальных – наоборот. Если приложение работает в гетерогенной сети, через сеть задачи

обмениваются данными в формате XDR (eXternal Data Representation), принятом в Internet. Перед отправкой и после приема данных задача конвертирует их в/из формата XDR. Естественно, при этом MPI должен знать не просто количество передаваемых байт, но и тип содержимого.

3. Обязательным требованием к MPI была поддержка языка FORTRAN в силу его инерционной популярности. В этом языке тип CHARACTER требует особого обращения, поскольку переменная такого типа содержит не собственно текст, а адрес текста и его длину. Функция MPI, получив адрес переменной, должна извлечь из нее адрес текста и копировать сам текст. Это и произойдет, если в поле аргумента-описателя типа стоит MPI_CHARACTER. Ошибка в указании типа приведет: при отправке – к копированию служебных данных вместо текста, при приеме – к записи текста на место служебных данных. И то, и другое приводит к ошибкам времени выполнения.
4. Такие часто используемые в C типы данных, как структуры, могут содержать в себе некоторое пустое пространство, чтобы все поля в переменной такого типа размещались по адресам, кратным некоторому четному числу (часто 2, 4 или 8) – это ускоряет обращение к ним. Причины тому чисто аппаратные. Выравнивание данных настраивается ключами компилятора. Разные задачи одного и того же приложения, выполняющиеся на одной и той же машине (даже на одном и том же процессоре), могут быть построены с разным выравниванием, и типы с одинаковым текстовым описанием будут иметь разное двоичное представление. MPI будет вынужден позаботиться о правильном преобразовании. Например, переменные такого типа могут занимать 9 или 16 байт:

```
typedef struct { char    c;
                 double  d;
               } CharDouble;
```

Таблица 1. Предопределенные константы типа элементов сообщений

Константы MPI	Тип в C
MPI_BYTE	
MPI_CHAR	*signed char*
MPI_SHORT	*signed int*
MPI_INT	*signed int*
MPI_LONG	*signed long int*
MPI_UNSIGNED_CHAR	*unsigned char*
MPI_UNSIGNED_SHORT	*unsigned int*
MPI_UNSIGNED	*unsigned int*
MPI_UNSIGNED_LONG	*unsigned long int*
MPI_FLOAT	*float*

MPI_DOUBLE	*double*
MPI_LONG_DOUBLE	*long double*

MPI_BYTE – это особый описатель типа; который не описывает тип данных для конкретного языка программирования (в C он ближе всего к *unsigned char*). Использование *MPI_BYTE* означает, что содержимое соответствующего массива ***не должно*** подвергаться ***никаким*** преобразованиям - и на приемной, и на передающей стороне массив будет иметь одну и ту же длину и одинаковое ***двоичное*** представление.

Коллективные функции

Под термином "коллективные" в MPI подразумеваются три группы функций:

- ✧ функции коллективного обмена данными;
- ✧ точки синхронизации, или барьеры;
- ✧ функции поддержки распределенных операций.

Коллективная функция одним из аргументов получает описатель области связи (коммуникатор). Вызов коллективной функции является корректным, только если произведен из всех процессов-абонентов соответствующей области связи, и именно с этим коммуникатором в качестве аргумента (хотя для одной области связи может иметься несколько коммуникаторов, подставлять их вместо друг друга нельзя). В этом и заключается коллективность: либо функция вызывается всем коллективом процессов, либо никем; третьего не дано.

Если требуется ограничить область действия коллективной функции только частью присоединенных к коммуникатору задач, или наоборот - расширить область действия, то необходимо создать временную группу/область связи/коммуникатор на базе существующих.

Функция синхронизации процессов

Этим занимается всего одна функция:

```
int MPI_Barrier( MPI_Comm comm);
```

Параметр: *comm* - идентификатор группы.

Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы *comm* также не выполнят эту процедуру.

Гарантирует, что к выполнению следующей за *MPI_Barrier* инструкции каждая задача приступит одновременно с остальными.

Это единственная в MPI функция, вызовами которой гарантированно синхронизируется во времени выполнение различных процессов. Некоторые другие коллективные функции в зависимости от реализации могут обладать,

а могут и не обладать свойством одновременно возвращать управление всем процессам; но для них это свойство является побочным и необязательным. Если в программе необходима синхронность, используйте только *MPI_Barrier*.

Когда может потребоваться синхронизация? В примере 3.1 синхронизация используется перед аварийным завершением: там процесс 0 рапортует об ошибке, и чтобы ни один из оставшихся процессов вызовом *MPI_Abort* не завершила нулевой досрочно-принудительно, перед *MPI_Abort* поставлен барьер.

Алгоритмической необходимости в барьерах, как представляется, нет. Параллельный алгоритм для своего описания требует по сравнению с алгоритмом классическим всего лишь двух дополнительных операций – приема и передачи данных между процессами. Точки синхронизации несут чисто технологическую нагрузку вроде той, что описана в предыдущем абзаце.

Функции коллективного обмена данными.

Основные особенности и отличия от обычных межпроцессных коммуникаций типа "точка-точка":

- ✧ на прием и/или передачу работают одновременно ***все*** задачи-абоненты указываемого коммуникатора, соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров;
- ✧ коллективная функция выполняет одновременно и прием, и передачу; она имеет большое количество параметров, часть которых нужна для приема, а часть для передачи; в разных задачах та или иная часть игнорируется;
- ✧ как правило, значения ***всех*** параметров (за исключением адресов буферов) должны быть идентичными во всех задачах;
- ✧ MPI назначает идентификатор для сообщений автоматически; кроме того, сообщения передаются не по указываемому коммуникатору, а по временному коммуникатору-дупликату; тем самым потоки данных коллективных функций надежно изолируются друг от друга и от потоков, созданных функциями "точка-точка";
- ✧ возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено, как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

```
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype,
               int source, MPI_Comm comm);
```

Параметры:

buf - адрес начала буфера посылки сообщения (выходной параметр);

count - число передаваемых элементов в сообщении
datatype - тип передаваемых элементов
source - номер рассылающего процесса
comm - идентификатор группы

Выполняется рассылка сообщения от процесса *source* всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера *buf* процесса *source* будет скопировано в локальный буфер процесса. Значения параметров *count*, *datatype* и *source* должны быть одинаковыми у всех процессов.

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype,
               void *rbuf, int rcount, MPI_Datatype rtype,
               int dest, MPI_Comm comm);
```

Параметры:
sbuf - адрес начала буфера посылки
scount - число элементов в посылаемом сообщении
stype - тип элементов отсылаемого сообщения
rbuf - адрес начала буфера сборки данных (выходной параметр)
rcount - число элементов в принимаемом сообщении
rtype - тип элементов принимаемого сообщения
dest - номер процесса, на котором происходит сборка данных
comm - идентификатор группы
ierror - код ошибки (выходной параметр)

Данная функция ("совок") выполняет сборку данных со всех процессов в буфере *rbuf* процесса *dest*. Каждый процесс, включая *dest*, посылает содержимое своего буфера *sbuf* процессу *dest*. Собирающий процесс сохраняет данные в буфере *rbuf*, располагая их в порядке возрастания номеров процессов. Параметр *rbuf* имеет значение только на собирающем процессе и на остальных игнорируется, значения параметров *count*, *datatype* и *dest* должны быть одинаковыми у всех процессов.

Векторный вариант функции сбора данных – *MPI_Gatherv* – позволяет задавать ***разное*** количество отправляемых данных в разных задачах-отправителях. Соответственно, на приемной стороне задается массив позиций в приемном буфере, по которым следует размещать поступающие данные, и максимальные длины порций данных от всех задач. Оба массива содержат позиции/длины не в байтах, а в количестве ячеек типа *rcount*.

Функция *MPI_Scatter* ("разбрызгиватель") выполняет обратную "совку" операцию – части передающего буфера из задачи root распределяются по приемным буферам всех задач.

Векторный вариант – *MPI_Scatterv*, рассылает части неодинаковой длины в приемные буфера неодинаковой длины.

Функция *MPI_Allgather* аналогична *MPI_Gather*, но прием осуществляется не в одной задаче, а во ***всех***: каждая имеет специфическое содержимое в передающем буфере, и все получают одинаковое содержимое в буфере приемном. Как и в *MPI_Gather*, приемный буфер последовательно заполняется данными изо всех передающих. Вариант с неодинаковым количеством данных называется *MPI_Allgatherv*.

MPI_Alltoall : каждый процесс нарезает передающий буфер на куски и рассылает куски остальным процессам; каждый процесс получает куски от всех остальных и поочередно размещает их приемном буфере. Это "совок" и "разбрызгиватель" в одном флаконе. Векторный вариант называется *MPI_Alltoallv*.

В учебнике, изданном MIT Press, есть хорошая схема для всех перечисленных в этом разделе функций.

Помните, что коллективные функции несовместимы с коммуникационными функциями типа "точка-точка": недопустимым, например, является вызов в одной из принимающих широковещательное сообщение задач *MPI_Recv* вместо *MPI_Bcast*.

Распределенные операции.

Идея проста: в каждой задаче имеется массив. Над нулевыми ячейками всех массивов производится некоторая операция (сложение/произведение/ поиск минимума/максимума и т.д.), над первыми ячейками производится такая же операция и т.д. Четыре функции предназначены для вызлва этих операций и отличаются способом размещения результата в задачах.

```
int MPI_AllReduce( void *sbuf, void *rbuf, int count,
                   MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm);
```

Параметры:

sbuf - адрес начала буфера для аргументов
rbuf - адрес начала буфера для результата (выходной параметр)
count - число аргументов у каждого процесса
datatype - тип аргументов
op - идентификатор глобальной операции
comm - идентификатор группы

Выполнение *count* глобальных операций *op* с возвратом *count* результатов во всех процессах в буфере *rbuf*. Операция выполняется независимо над соответствующими аргументами всех процессов. Значения параметров *count* и *datatype* у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция *op* обладает свойствами ассоциативности и коммутативности.

```
int MPI_Reduce( void *sbuf, void *rbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root,
                MPI_Comm comm);
```

Параметры:

sbuf - адрес начала буфера для аргументов
rbuf - адрес начала буфера для результата (выходной параметр)
count - число аргументов у каждого процесса
datatype - тип аргументов
op - идентификатор глобальной операции
root - процесс-получатель результата
comm - идентификатор группы

Функция аналогична предыдущей, но результат будет записан в буфер *rbuf* только у процесса *root*.

```
int vector[16];
int resultVector[16];
MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
for( i=0; i<16; i++ )
    vector[i] = myRank*100 + i;
MPI_Reduce(
    vector,  /* каждая задача в коммуникаторе предоставляет
                вектор */
    resultVector,   /* задача номер 'root' собирает данные
                       сюда */
    16,      /* количество ячеек в исходном и результирующем
                массивах */
    MPI_INT, /* и тип ячеек */
    MPI_SUM, /* описатель операции: поэлементное сложение
                векторов */
    0,       /* номер задачи, собирающей результаты в
                'resultVector' */
    MPI_COMM_WORLD  /* описатель области связи */
);
if( myRank==0 )
    /* печатаем resultVector, равный сумме векторов */
```

Предопределенных описателей операций в MPI насчитывается 12:

- *MPI_MAX* и *MPI_MIN* ищут поэлементные максимум и минимум;
- *MPI_SUM* вычисляет сумму векторов;
- *MPI_PROD* вычисляет поэлементное произведение векторов;
- *MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR* - логические и двоичные операции И, ИЛИ, исключающее ИЛИ;
- *MPI_MAXLOC, MPI_MINLOC* - поиск индексированного минимума/максимума - здесь не рассматриваются.

Каждая функция работает с массивами определенных типов (таблица 2). Этим типам операндов могут соответствовать описатели, приведенные в таблице 3.

Таблица 2. Соответствие операций и операндов

Операция	Допустимый тип операндов
MPI_MAX, MPI_MIN	целые и вещественные
MPI_SUM, MPI_PROD	целые, вещественные, комплексные
MPI_LAND, MPI_LOR, MPI_LXOR	целые и логические
MPI_LAND, MPI_LOR, MPI_LXOR	целые (в т.ч. байтовые)

Таблица 3. Соответствие операндов и описателей

Тип	Описатель в С	Описатель в FORTRAN
Целый	*MPI_INT, MPI_UNSIGNED_INT, MPI_LONG, MPI_UNSIGNED_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT*	MPI_INTEGER
Целый Байтовый	**MPI_BYTE**	(нет)
Вещественный	*MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE*	MPI_REAL, MPI_DOUBLE_ PRECISION
Логический	(нет, пользуйтесь типом int)	MPI_LOGICAL
Комплексный	(нет)	MPI_COMPLEX

Количество поддерживаемых операциями типов для ячеек векторов строго ограничено вышеперечисленными. Никакие другие встроенные или пользовательские описатели типов использоваться не могут.

MPI_Reduce_scatter : каждая задача получает не весь массив-результат, а его часть. Длины этих частей находятся в массиве-третьем параметре функции. Размер исходных массивов во всех задачах одинаков и равен сумме длин результирующих массивов.

MPI_Scan : аналогична функции *MPI_Allreduce* в том отношении, что каждая задача получает результрующий массив. Главное отличие: здесь содержимое массива-результата в задаче i является результатом выполнение операции над массивами из задач с номерами от 0 до i включительно.

Помимо встроенных, пользователь может вводить свои собственные операции, но механизм их создания здесь не рассматривается. Для этого служат функции *MPI_Op_create* и *MPI_Op_free*, а также тип *MPI_User_function*.

Коммуникаторы, группы и области связи

Группа - это некое множество процессов. Один процесс может быть членом нескольких групп. В распоряжение программиста предоставлен тип *MPI_Group* и набор функций, работающих с переменными и константами этого типа. Констант, собственно, две: *MPI_GROUP_EMPTY* может быть возвращена, если группа с запрашиваемыми характеристиками в принципе может быть создана, но пока не содержит ни одной ветви; *MPI_GROUP_NULL* возвращается, когда запрашиваемые характеристики противоречивы. Согласно концепции MPI, после создания группу нельзя дополнить или усечь – можно создать только новую группу под требуемый набор ветвей на базе существующей.

Область связи (*communication domain*) – это нечто абстрактное. В распоряжении программиста нет типа данных, описывающего непосредственно области связи, как нет и функций по управлению ими. Области связи автоматически создаются и уничтожаются вместе с коммуникаторами. Абонентами одной области связи являются ***все*** задачи либо одной, либо двух групп.

Коммуникатор, или описатель области связи – это верхушка трехслойного пирога (группы, области связи, описатели областей связи), с которым работают задачи. Именно с коммуникаторами программист имеет дело, вызывая функции пересылки данных, а также подавляющую часть вспомогательных функций. Одной области связи могут соответствовать несколько коммуникаторов. Коммуникаторы являются "несообщающимися сосудами": если данные отправлены через один коммуникатор, ветвь-получатель сможет принять их только через этот же самый коммуникатор, но ни через какой-либо другой.

Зачем нужны разные группы, разные области связи и разные их описатели?

- ✧ По существу, они служат той же цели, что и идентификаторы сообщений - помогают ветви-приемнику и ветви-получателю надежнее определять друг друга, а также содержимое сообщения;
- ✧ Ветви внутри параллельного приложения могут объединяться в подколлективы для решения промежуточных задач - посредством создания групп, и областей связи над группами. Пользуясь описателем этой области связи, ветви гарантированно ничего не примут извне подколлектива, и ничего не отправят наружу. Параллельно при этом они могут продолжать пользоваться любым другим имеющимся в их распоряжении коммуникатором для пересылок вне подколлектива, например, *MPI_COMM_WORLD* для обмена данными внутри всего приложения;
- ✧ Коллективные функции создают дубликат от полученного аргументом коммуникатора, и передают данные через дубликат, не опасаясь, что их

сообщения будут случайно перепутаны с сообщениями функций "точка-точка", распространяемыми через оригинальный коммуникатор;

- ✧ Программист с этой же целью в разных кусках кода может передавать данные между ветвями через разные коммуникаторы, один из которых создан копированием другого.
- ✧ Коммуникаторы распределяются автоматически (функциями семейства "Создать новый коммуникатор"), и для них не существует джокеров ("принимай через какой угодно коммуникатор") – вот еще два их существенных достоинства перед идентификаторами сообщений. Идентификаторы (целые числа) распределяются пользователем вручную, и это служит источником двух частых ошибок вследствие путаницы на приемной стороне:
- ✧ сообщениям, имеющим разный смысл, вручную по ошибке назначается один и тот же идентификатор;
- ✧ функция приема с джокером сгребает все подряд, в том числе и те сообщения, которые должны быть приняты и обработаны в другом месте ветви.

Важно помнить, что все функции, создающие коммуникатор, являются *коллективными*. Именно это качество позволяет таким функциям возвращать в разные ветви *один и тот же* описатель. Коллективность заключается в следующем:

- ✧ одним из аргументов функции является коммуникатор;
- ✧ функцию должны вызывать *все* ветви-абоненты указываемого коммуникатора.

Создание коммуникаторов и групп

Копирование. Самый простой способ создания коммуникатора - скопировать "один-в-один" уже имеющийся:

```
int MPI_Comm_dup( MPI_Comm comm, MPI_Comm *newcomm);
```

Параметры:

comm - идентификатор группы

newcomm - идентификатор новой группы (выходной параметр)

Новая группа при этом не создается - набор задач остается прежним. Новый коммуникатор наследует все свойства копируемого.

Разбиение. Соответствующая коммуникатору группа разбивается на непересекающиеся подгруппы, для каждой из которых заводится свой коммуникатор.

```
int MPI_Comm_Split( MPI_Comm comm, int color, int key,
                    MPI_Comm *newcomm);
```

Параметры:

comm - идентификатор группы
color - признак разделения на группы
key - параметр, определяющий нумерацию в новых группах
newcomm - идентификатор новой группы (выходной параметр)

Данная процедура разбивает все множество процессов, входящих в группу *comm*, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра *color* (неотрицательное число). Каждая новая подгруппа содержит все процессы одного цвета. Если в качестве *color* указано значение *MPI_UNDEFINED*, то в *newcomm* будет возвращено значение *MPI_COMM_NULL*.

Создание через группы. В предыдущих двух случаях коммуникатор создается от существующего коммуникатора напрямую, без явного создания группы: группа либо та же самая, либо создается автоматически. Самый же общий способ таков:

1. функцией *MPI_Comm_group* определяется группа, на которую указывает соответствующий коммуникатор;
2. на базе существующих групп функциями семейства *MPI_Group_xxx* создаются новые группы с нужным набором ветвей;
3. для итоговой группы функцией *MPI_Comm_create* создается коммуникатор; не забудьте, что она должна быть вызвана во ВСЕХ ветвях-абонентах коммуникатора, передаваемого первым параметром;
4. все описатели созданных групп очищаются вызовами функции *MPI_Group_free*.

Такой механизм позволяет, в частности, не только расщеплять группы подобно *MPI_Comm_split*, но и объединять их. Всего в MPI определено 7 разных функций конструирования групп.

Может ли задача обратиться к области связи, абонентом которой не является? Нет. Описатель области связи передается в задачу функциями MPI, которые одновременно делают эту задачу абонентом описываемой области. Таков единственный существующий способ получить описатель. Попытки "пиратскими" средствами обойти это препятствие (например, получить описатель, посредством *MPI_Send/MPI_Recv* переслать его в другую задачу, не являющуюся его абонентом, и там им воспользоваться) не приветствуются, и исход их, скорее всего, будет определяться деталями реализации.

Пример 3

```
/*
 *
 *  Создание коммуникатора-дупликата, надежное разделение
 *  потоков сообщений:
 *     MPI_Comm_dup, MPI_Comm_free
 *
 *  Измененный подход к обработке ошибок через
```

```
 *     MPI_Abort, MPI_Barrier
 */

#include <mpi.h>
#include <stdio.h>

#define  tag1  1
#define  tag2  2
#define  tag3  1   /* сознательная "ошибка": идентификатор
                      равен 'tag1' */

#define  ELEMS(x)  (sizeof( x )/sizeof( x[0] ))

int main( int argc, char **argv )
{
    MPI_Comm myComm;
    int rank, size;

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

/* Обработка ошибки "неверное кол-во запущенных задач" */
    if( size != 2 && rank==0 ) {
        printf( "ERROR: 2 processes required ",
                "instead of %d, abort.\n", size );
        MPI_Abort( MPI_COMM_WORLD, MPI_ERR_OTHER );
    }
/* Барьер нужен, чтобы в случае ошибки, пока ветвь 0
 * рапортует о ней и вызывает MPI_Abort, остальные ветви
 * не смогли приступить к выполнению содержательной части
 * программы.
 */
    MPI_Barrier( MPI_COMM_WORLD );

/**  Общая часть закончена,
 ** начинается содержательная часть...
 **/

/* Создаем еще один коммуникатор - копию MPI_COMM_WORLD */
    MPI_Comm_dup( MPI_COMM_WORLD, &myComm );

    if( rank == 0 )   /*  Ветвь 0 передает  */
    {
        static char buf1[] = "Contents of first message";
        static char buf2[] = "Contents of second message";
        static char buf3[] = "Contents of third message";

/* Обратите внимание: хотя для сообщений выбран один и тот
 * же идентификатор, описатели области связи разные
```

```
 */
        MPI_Send( buf1, ELEMS(buf1), MPI_CHAR, 1,
                  tag1, myComm );
        MPI_Send( buf2, ELEMS(buf2), MPI_CHAR, 1,
                  tag2, MPI_COMM_WORLD );
        MPI_Send( buf3, ELEMS(buf3), MPI_CHAR, 1,
                  tag3, MPI_COMM_WORLD );
    }
    else     /*  Ветвь 1 принимает  */
    {
        char buf1[100], buf2[100], buf3[100];
        MPI_Status st;

/* Вызов НЕ перехватит первое сообщен ие из-за того,
   что tag1=tag3 */

        MPI_Recv( buf3, ELEMS(buf3), MPI_CHAR, 0,
                  tag3, MPI_COMM_WORLD, &st );

/* Вызов НЕ перехватит первое сообщение джокером */

        MPI_Recv( buf2, ELEMS(buf2), MPI_CHAR, 0,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &st );

/* Первое сообщение будет успешно принято там, где надо */

        MPI_Recv( buf1, ELEMS(buf1), MPI_CHAR, 0,
                  tag1, myComm, &st );

          /* Печатаем результаты приема */
        printf("Received in buf1 = \'%s\'\n", buf1 );
        printf("Received in buf2 = \'%s\'\n", buf2 );
        printf("Received in buf3 = \'%s\'\n", buf3 );

    } /* rank 1 */

/* Уведомляем MPI, что больше коммуникатором не пользуемся.
 * После этого myComm будет сброшен в MP I_COMM_NULL
 * (то есть в 0),а соответствующие ему данные будут помечены
 * на удаление.
 */

    MPI_Comm_free( &myComm );

    MPI_Finalize();
}
```

Пример 4

```
/*
```

```
 *
 *  Создание подкоммуникаторов. Неявное разбиение группы
 *  на подгруппы:
 *     MPI_Comm_split
 *
 *  Рекомендация:
 *    запустите этот пример несколько раз с разным числом
 *    ветвей N ( -np N ), например: N = 6,7,8
 */

#include <mpi.h>
#include <stdio.h>

#define  tag1  1
#define  tag2  2
#define  tag3  1   /* сознательная "ошибка": идентификатор
                      равен 'tag1' */

#define  ELEMS(x)  (sizeof( x )/sizeof( x[0] ))

int main( int argc, char **argv )
{
    MPI_Comm subComm;
    int rank, size, subCommIndex, subRank, subSize;

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

/* Распределяем по следующему правилу:
 * каждые три ветви - в подгруппу,
 * каждую четвертую - "в никуда"
 */
    subCommIndex = rank % 4;
    if( subCommIndex == 3 )
        subCommIndex = MPI_UNDEFINED;

      /* Желательная нумерация внутри подгрупп:
       * обратная той, что имеется в MPI_COMM_WORLD
       */
    subRank = size - rank;

      /* Вызываем во ВСЕХ ветвях-абонентах MPI_COMM_WORLD,
       * указанного первым аргументом функции
       */
    MPI_Comm_split( MPI_COMM_WORLD, subCommIndex,
                    rank, &subComm );
      /* Каждая ветвь пишет, к чему она теперь относится
       */
    printf("My rank in MPI_COMM_WORLD = %d. ", rank);
```

```
    if( subComm == MPI_COMM_NULL )
        printf("I\'m not attached ",
               "to any subcommunicator\n");
    else {
        MPI_Comm_size( subComm, &subSize );
        MPI_Comm_rank( subComm, &subRank );
        printf("My local rank = %d, local size = %d\n",
               subRank, subSize );
    }

    MPI_Comm_free( &subComm );

    MPI_Finalize();
}
```

Полезная нагрузка коммуникатора: атрибуты.

Помимо характеристик области связи, тело коммуникатора содержит в себе некие дополнительные данные (атрибуты). Механизм хранения атрибутов называется *"caching"*. Атрибуты могут быть системные и пользовательские; в системных, в частности, хранятся:

- ✧ адрес функции-обработчика ошибок;
- ✧ описание пользовательской топологии;
- ✧ максимально допустимый идентификатор для сообщений.

Атрибуты идентифицируются целыми числами, которые MPI назначает автоматически. Некоторые константы для описания системных атрибутов: *MPI_TAG_UB*, *MPI_HOST*, *MPI_IO*, *MPI_WTIME_IS_GLOBAL*. К этим атрибутам программист обращается редко, и менять их не может; а для таких часто используемых атрибутов, как обработчик ошибок или описание топологии, существуют персональные наборы функций, например, *MPI_Errhandler_xxx*.

Атрибуты - удобное место хранения совместно используемой информации; помещенная в атрибут одной из ветвей, такая информация становится доступной всем использующим коммуникатор ветвям без пересылки сообщений (вернее, на MPP-машине, к примеру, сообщения будут, но на системном уровне, т.е. скрытые от глаз программиста).

Пользовательские атрибуты создаются и уничтожаются функциями *MPI_Keyval_create* и *MPI_Keyval_free*; модифицируются функциями *MPI_Attr_put*, *MPI_Attr_get* и *MPI_Attr_delete*. При создании коммуникатора на базе существующего атрибуты из последнего тем или иным образом копируются или нет в зависимости от функции копирования типа *MPI_Copy_function*, адрес которой является параметром функции создания атрибута.

То же и для удаления атрибутов при уничтожении коммуникатора: задается пользовательской функцией типа *MPI_Delete_function*, указываемой при создании атрибута.

Корректное удаление отслуживших описателей.

```
int MPI_Comm_Free( MPI_Comm comm);
```

Параметр: *comm* - идентификатор группы (выходной параметр)

Уничтожает группу, ассоциированную с идентификатором *comm*, который после возвращения устанавливается в *MPI_COMM_NULL*.

Здесь имеются в виду все типы системных данных, для которых предусмотрена функция *MPI_Xxx_free* (и константа *MPI_XXX_NULL*). В MPI-I их семь (см. *mpi.h*):

- ✧ коммуникаторы;
- ✧ группы;
- ✧ типы данных;
- ✧ распределенные операции;
- ✧ квитанции (request's);
- ✧ атрибуты коммуникаторов;
- ✧ обработчики ошибок (errhandler's).

Дальше все описывается на примере коммуникаторов и групп, но изложенная схема является общей для всех типов ресурсов.

Не играет роли, в каком порядке уничтожать взаимосвязанные описатели. Главное - не забыть вызвать функцию удаления ресурса *MPI_Xxx_free* вовсе. Соответствующий ресурс не будет удален немедленно, он прекратит существование только если будут выполнены два условия:

- ✧ программе пользователя никогда не предоставлялись ссылки на ресурс, или все пользовательские ссылки очищены вызовами *MPI_Xxx_free* ;
- ✧ ресурс перестает использоваться другими ресурсами MPI, то есть удаляются все системные ссылки.

Взаимосвязанными описателями являются описатели коммуникатора и группы (коммуникатор ссылается на группу); или описатели типов, если один создан на базе другого (порожденный ссылается на исходный).

Пример 5

```
MPI_Comm subComm;
MPI_Group subGroup;
int rank;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

MPI_Comm_split( MPI_COMM_WORLD, rank / 3,
                rank % 3, &subComm );
/* Теперь создан коммуникатор subComm, и автоматически
 * создана группа, на которую распространяется его область
 * действия.
 * На коммуникатор заведена ссылка из программы - subComm.
 * На группу заведена системная ссылка из коммуникатора.
```

```
 */

MPI_Comm_group( subComm, &subGroup );
 /* Теперь на группу имеется две ссылки - системная
  * из коммуникатора, и пользовательская subGroup.
  */

MPI_Group_free( &subGroup );
 /* Пользовательская ссылка на группу уничтожена,
  * subGroup сброшен в MPI_GROUP_NULL.
  * Собственно описание группы из системных данных
  * не удалено,так как на него еще ссылается коммуникатор.
  */

MPI_Comm_free( &subComm );
 /* Удалена пользовательская ссылка на коммуникатор,
  * subComm сброшен в MPI_COMM_NULL. Так как других ссылок
  * на коммуникатор нет, его описание удаляется из системных
  * данных.
  * Вместе с коммуникатором удалена системная ссылка
  * на группу.Так как других ссылок на группу нет,
  * ее описание удаляется из системных данных.
  */
```

Еще раз отметим, что для MPI не играет роли, в каком порядке будут вызваны завершающие вызовы *MPI_Xxx_free*, это дело программы.

Не пытайтесь уничтожать константные описатели вроде *MPI_COMM_WORLD* или *MPI_CHAR*: их создание и уничтожение - дело самого MPI.

В заключение обсудим некоторые детали, не связанные напрямую с организацией межпроцессорных взаимодействий.

Обработчики ошибок. По умолчанию, если при выполнении функции MPI обнаружена ошибка, выполнение всех ветвей приложения завершается. Это сделано в расчете на программиста, не привыкшего проверять коды завершения (*malloc,open,write,...*;), и пытающегося распространить такой стиль на MPI. При аварийном завершении по такому сценарию на консоль выдается очень скудная информация: в лучшем случае, там будет название функции MPI и название ошибки. Обработчик ошибок является принадлежностью коммуникатора, для управления обработчиками служат функции семейства *MPI_Errhandler_xxx* (http://www.csa.ru).

Многопоточность. Сам MPI неявно использует многопоточность очень широко, и не мешает программисту делать то же самое. Однако: разные задачи имеют с точки зрения MPI ***обязательно*** разные номера, а разные потоки (*threads*) внутри одной задачи для него ничем не отличаются. Программист сам идентификаторами сообщений и коммуникаторами должен устанавливать такую дисциплину для потоков, чтобы один поток не стал,

допустим, вызывая *MPI_Recv*, джокером перехватывать сообщения, которые должен принимать и обрабатывать другой поток той же задачи. Другим источником ошибок может быть использование разными потоками коллективных функций над одним и тем же коммуникатором: используйте *MPI_Comm_dup*.

Работа с файлами. В MPI-2 средства перенаправления работы с файлами появились, в MPI-1 их нет. Все вызовы функций напрямую передаются операционной системе (*Unix/Parix/NFS/...*) на той машине, или на том процессорном узле MPP-компьютера, где находится вызывающая ветвь. Теоретически возможность подключения средств расширенного управления вводом/выводом в MPI-1 есть - каждый коммуникатор хранит атрибут с числовым кодом *MPI_IO* - это номер ветви, в которую перенаправляется ввод/вывод от всех остальных ветвей в коммуникаторе; сам MPI ничего с ним не делает и никак не использует. Для MPI существует ряд дополнительных библиотек такого рода, как для конкретных платформ, так и свободно распространяемые многоплатформенные, но я не видел ни одной.

Работа с консолью также отдается на откуп системе; это может приводить к перемешиванию вывода нескольких задач, поэтому рекомендуется весь вывод на экран производить либо из какой-то одной задачи (нулевой?), либо в начале функции main() написать:

```
setvbuf( stdout, NULL, _IOLBF, BUFSIZ );
setvbuf( stderr, NULL, _IOLBF, BUFSIZ );
```

И не забудьте написать "#include <stdio.h>" в начале программы.