Итак, вы приступаете к созданию параллельной программы. Желание есть, задача ясна, метод выбран, целевой компьютер, скорее всего, тоже определен. Осталось только все мысли выразить в той или иной форме, понятной для этого компьютера. Чем руководствоваться, если собственного опыта пока мало, а априорной информации о доступных технологиях параллельного программирования явно недостаточно? Наводящих соображений может быть много, но в результате вы все равно будете вынуждены пойти на компромисс, делая выбор между временем разработки программы, ее эффективностью и переносимостью, интенсивностью последующего использования программы, необходимостью ее дальнейшего развития. Не вдаваясь в детали выбора, попробуйте для начала оценить, насколько важны для вас следующие три характеристики.

Основное назначение параллельных компьютеров — это быстро решать задачи. Если технология программирования по разным причинам не позволяет в полной мере использовать весь потенциал вычислительной системы, то нужно ли тратить усилия на ее освоение? Не будем сейчас обсуждать причины. Ясно то, что *возможность создания эффективных программ* является серьезным аргументом в выборе средств программирования.

Технология может давать пользователю полный контроль над использованием ресурсов вычислительной системы и ходом выполнения его программы. Для этого ему предлагается набор из нескольких сотен конструкций и функций, предназначенных "на все случаи жизни". Он может создать действительно эффективную программу, если правильно воспользуется предложенными средствами. Но захочет ли он это делать? Не сто́ит забывать, что он должен решать свою задачу из своей предметной области, где и своих проблем хватает. Маловероятно, что физик, химик, геолог или эколог с большой радостью захочет осваивать новую специальность, связанную с параллельным программированием. *Возможность быстрого создания параллельных программ* должна приниматься в расчет наравне с другими факторами.

Вычислительная техника меняется очень быстро. Предположим, что была найдена технология, позволяющая быстро создавать эффективные параллельные программы. Что произойдет через пару лет, когда появится новое поколение компьютеров? Возможных вариантов развития событий два. Первый вариант — разработанные прежде программы были "одноразовыми" и сейчас ими уже никто не интересуется. Бывает и так. Однако, как правило, в параллельные программы вкладывается слишком много средств (времени, усилий, финансовых затрат), чтобы просто так об этом забыть и начать разработку заново. Хочется перенести накопленный багаж на новую компьютерную платформу. Скорее всего, на новом компьютере старая программа рано или поздно работать будет, и даже будет давать правильный результат. Но дает ли выбранная технология *гарантии сохранения эффективности параллельной программы* при ее переносе с одного компьютера на другой? Скорее всего, нет. Программу для новой платформы нужно оптимизировать за-

ново. А тут еще разработчики новой платформы предлагают вам очередную новую технологию программирования, которая опять позволит создать выдающуюся программу для данного компьютера. Программы переписываются, и так по кругу в течении многих лет.

Выбор технологии параллельного программирования – и в самом деле очень непростой вопрос. Даже поверхностный анализ и обзор средств, которые могут помочь в решении задач на параллельном компьютере, приведёт к списку из более сотни наименований.

В некоторых случаях выбор определяется просто. Например, вполне жизненной является ситуация, когда воспользоваться можно только тем, что установлено на доступном вам компьютере. Другой аргумент звучит так: "... все используют MPI, поэтому и я тоже буду...". Если есть возможность и желание сделать осознанный выбор, это обязательно нужно делать. Посоветуйтесь со специалистами. Проблемы в дальнейшем возникнут в любом случае, вопрос только насколько быстро и в каком объеме. Если выбор будет правильным, проблем будет меньше. Если неправильным, то тоже не отчаивайтесь, будет возможность подумать о выборе еще раз. Сделав выбор несколько раз, вы станете специалистом в данной области, забудете о своих прежних интересах, скажем, о квантовой химии или вычислительной гидродинамике. Не исключено, что в итоге вы сможете предложить свою технологию и найти ответ на центральный вопрос параллельных вычислений: "Как создавать эффективные программы для параллельных компьютеров?"

Параллельные программы по определению являются такими, которые имеют параллельно работающие части. Среди средств параллельного программирования чаще всего выделяются библиотеки (MPI, PVM, PThreads, Shmem и т.д.); расширения последовательных языков (OpenMP, Cilk+, UPC, HPF и т.д.) и непосредственно параллельные языки (X10, Chapel, Occam, Erlang и т.д.)

Широкое распространение компьютеров с распределенной памятью определило и появление соответствующих систем программирования. Как правило, в таких системах отсутствует единое адресное пространство, и для обмена данными между параллельными процессами используется явная передача сообщений через коммуникационную среду. Отдельные процессы описываются с помощью традиционных языков программирования, а для организации их взаимодействия вводятся дополнительные функции. По этой причине практически все системы программирования, основанные на явной передаче сообщений, существуют не в виде новых языков, а в виде интерфейсов и библиотек.

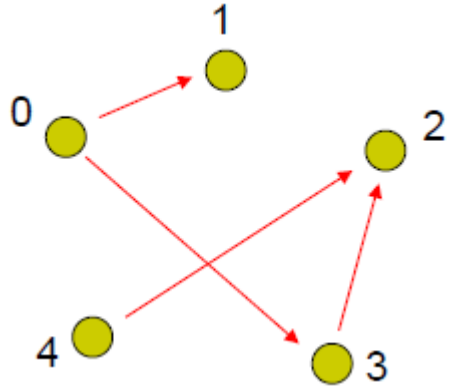
К настоящему времени примеров известных систем программирования на основе передачи сообщений накопилось довольно много: Shmem, Linda, PVM, MPI. Мы остановимся на последней.

Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это отражено в названии технологии – Message Passing Interface. Стандарт фиксирует интерфейс, который соблюдается как системой программирования MPI на каждой вычислительной системе, так и пользователем при создании программ. Мы будем рассматривать примеры и описания

функций с использованием языка C, однако основные идеи MPI и правила оформления отдельных конструкций для разных языков в целом схожи.
Полная версия интерфейса содержит более 120 функций, и мы не будем ставить задачей описать его полностью, объяснены будут идея технологии и необходимые на практике компоненты с наиболее употребляемыми функциями. За прочими вопросами можно обратиться к документации.
Интерфейс поддерживает создание параллельных программ в стиле MIMD, что подразумевает объединение процессов с различными исходными текстами. Однако на практике программисты гораздо чаще используют SPMD-модель, в рамках которой для всех параллельных процессов используется один и тот же код. В настоящее время все больше и больше реализаций MPI поддерживают работу с нитями.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс `MPI_`. Например, функция посылки сообщения от одного процесса другому имеет имя `MPI_Send`. Если пользователь не будет использовать в программе имен с таким префиксом, то конфликтов с объектами MPI заведомо не будет. Все описания интерфейса MPI собраны в файле mpi.h, поэтому в начале MPI-программы должна стоять директива `#include <mpi.h>`.

MPI-программа — это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается.
Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в MPI нет. Основным способом взаимодействия между процессами является явная посылка сообщений.
MPI является стандартом интерфейса обмена сообщениями между параллельно работающими процессами, существует много реализации этого стандарта (как открытых, так и коммерческих). Официальная реализация представляет собой библиотеку подпрограмм для C (C++) и Fortran, но есть и реализации для других языков программирования (Java, Python, MATLAB).
При запуске MPI-программы запускается указанное в параметрах число копий программы на указанных ресурсах. Сама по себе MPI-программа и является множеством параллельно работающих процессов со своими кодом, данными и возможностью передавать друг другу сообщения в произвольные моменты времени.



Для локализации взаимодействия параллельных процессов программы можно создавать *группы процессов*, предоставляя им отдельную среду для общения — *коммуникатор*. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя `MPI_COMM_WORLD`. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут *номер процесса*, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда *два основных атрибута процесса: коммуникатор и номер в коммуникаторе.*

Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до $n - 1$. Подобная линейная нумерация не всегда адекватно отражает логическую взаимосвязь процессов приложения. Например, по смыслу задачи процессы могут располагаться в узлах прямоугольной решетки и взаимодействовать только со своими непосредственными соседями. Такую ситуацию пользователь может легко отразить в своей программе, описав соответствующую *виртуальную топологию процессов*. Эта информация может оказаться полезной при отображении процессов программы на физические процессоры вычислительной системы. Сам процесс отображения в MPI никак не специфицируется, однако система поддержки MPI в ряде случаев может значительно уменьшить коммуникационные накладные расходы, если воспользуется знанием виртуальной топологии.

Основным способом общения процессов между собой является посылка сообщений. *Сообщение* — это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32 767. Для работы с атрибутами сообщений введена структура `MPI_Status`, поля которой дают доступ к значениям атрибутов.

На практике сообщение чаще всего является набором однотипных данных, расположенных подряд друг за другом в некотором буфере. Такое сообщение может состоять, например, из двухсот целых чисел, которые пользователь разместил в соответствующем целочисленном векторе. Это типичная ситуация, на нее ориентировано большинство функций MPI, однако такая ситуация имеет, по крайней мере, два ограничения. Во-первых, иногда необходимо составить сообщение из разнотипных данных. Конечно же, можно отдельным сообщением послать количество вещественных чисел, содержащихся в последующем сообщении, но это может быть и неудобно программисту, и не столь эффективно. Во-вторых, не всегда посылаемые данные занимают непрерывную область в памяти. Если в Fortran элементы столбцов матрицы расположены в памяти друг за другом, то элементы строк уже идут с некоторым шагом. Чтобы послать строку, данные нужно сначала упаковать, передать, а затем вновь распаковать.

Чтобы снять указанные ограничения, в MPI предусмотрен *механизм для введения производных типов данных* (derived datatypes). Описав состав и схему размещения в памяти посылаемых данных, пользователь в дальнейшем работает с такими типами так же, как и со стандартными типами данных MPI.

На практике типы данных и виртуальные топологии встречаются не очень часто.

Стандарт MPI включает в себя коммуникации точка-точка, коллективные и односторонние коммуникации, типы данных, группы процессов (процессы могут объединяться в группы и в каждой группе своя нумерация), топологии (для каждой группы можно задать виртуальную топологию сети связи для наилучшего отображения на топологию реальной сети), коммуникаторы (описанный MPI_COMM_WORLD – включает в себя все запущенные MPI-процессы с топологией полного графа; в операциях передачи сообщений используются именно коммуникаторы), динамическое управление процессами и средства профилирования.

Общие функции MPI

Прежде чем переходить к описанию конкретных функций, сделаем несколько общих замечаний. При описании функций мы всегда будем пользоваться словом `OUT` для обозначения выходных параметров, через которые функция возвращает результаты. Даже если результатом работы функции является одно число, оно будет возвращено через один из параметров. Связано это с тем, что практически все функции MPI возвращают в качестве своего значения информацию об успешности завершения. В случае успешного выполнения функция вернет значение `MPI_SUCCESS`, иначе — код ошибки. Вид ошибки, которая произошла при выполнении функции, можно будет понять из описания каждой функции. Предопределенные возвращаемые значения, соответствующие различным ошибочным ситуациям, определены в файле mpi.h. В дальнейшем при описании конкретных функций, если ничего специально не сказано, то возвращаемое функцией значение будет подчиняться именно этому правилу.

Далее мы рассмотрим общие функции MPI, необходимые практически в каждой программе.

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

Инициализация параллельной части программы. Все другие функции MPI могут быть вызваны только после вызова `MPI_Init`. Необычный тип аргументов `MPI_Init` предусмотрен для того, чтобы иметь возможность передать всем процессам аргументы функции `main`.

Инициализация параллельной части для каждого приложения должна выполняться только один раз. Поскольку для сложных приложений, которые состоят из многих модулей и пишутся разными людьми, это отследить трудно, введена дополнительная функция `MPI_Initialized`:

```
MPI_Initialized (int *flag)
```

Здесь `OUT flag` — признак инициализации параллельной части программы.

Если функция `MPI_Init` уже была вызвана, то через параметр `flag` возвращается значение 1, в противном случае 0.

```
int MPI_Finalize(void)
```

Завершение параллельной части приложения. Все последующие обращения к любым MPI-функциям, в том числе к `MPI_Init`, запрещены. К моменту вызова `MPI_Finalize` каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Общая схема MPI-программы выглядит так:

```
main(int argc, char **argv)
{
...
MPI_Init(&argc, &argv);
...
MPI_Finalize();
...
}
```

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

- `comm` — идентификатор коммуникатора;
- `OUT size` — число процессов в коммуникаторе `comm`.

Определение общего числа параллельных процессов в коммуникаторе `comm`. Результат возвращается через параметр `size`, для чего функции передается адрес этой переменной. Поскольку коммуникатор является сложной структурой, перед ним стоит имя предопределенного типа `MPI_Comm`, определенного в файле mpi.h.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- `comm` — идентификатор коммуникатора;
- `OUT rank` — номер процесса в коммуникаторе `comm`.

Определение номера процесса в коммуникаторе `comm`. Если функция `MPI_Comm_size` для того же коммуникатора `comm` вернула значение `size`, то значение, возвращаемое функцией `MPI_Comm_rank` через переменную `rank`, лежит в диапазоне от 0 до `size-1`.

`double MPI_Wtime(void)`

Эта функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса. Заметим, что эта функция возвращает результат своей работы не через параметры, а явным образом.

Простейший пример программы, в которой использованы описанные выше функции, выглядит так:

```
main(int argc, char **argv)
{
    int me, size;
...
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Process %d size %d \n", me, size);
...
MPI_Finalize();
...
}
```

Строка, соответствующая функции `printf`, будет выведена столько раз, сколько процессов было порождено при вызове `MPI_Init`. Порядок появления строк заранее не определен и может быть, вообще говоря, любым. Гарантируется только то, что содержимое отдельных строк не будет перемешано друг с другом.

Прием/передача сообщений между отдельными процессами

Все функции передачи сообщений в MPI делятся на две группы. В одну группу входят функции, которые предназначены для взаимодействия двух процессов программы. Такие операции называются индивидуальным или операциями типа "точка-точка". Функции другой группы предполагают, что в операцию должны быть вовлечены все процессы некоторого коммуникатора. Такие операции называются коллективными.

Начнем описание функций обмена сообщениями с обсуждения операций типа "точка-точка". Все функции данной группы, в свою очередь, также делятся на два класса: функции с блокировкой (с синхронизацией) и функции без блокировки (асинхронные).

Прием/передача сообщений с блокировкой задаются конструкциями следующего вида.

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

- `buf` — адрес начала буфера с посылаемым сообщением;
- `count` — число передаваемых элементов в сообщении;
- `datatype` — тип передаваемых элементов;
- `dest` — номер процесса-получателя;
- `msgtag` — идентификатор сообщения;
- `comm` — идентификатор коммуникатора.

Блокирующая посылка сообщения с идентификатором `msgtag`, состоящего из `count` элементов типа `datatype`, процессу с номером `dest`. Все элементы посылаемого сообщения расположены подряд в буфере `buf`. Значение `count` может быть нулем. Разрешается передавать сообщение самому себе. Тип передаваемых элементов `datatype` должен указываться с помощью предопределенных констант типа, например, `MPI_INT`, `MPI_LONG`, `MPI_SHORT`, `MPI_LONG_DOUBLE`, `MPI_CHAR`, `MPI_UNSIGNED_CHAR`, `MPI_FLOAT` или с помощью введенных производных типов. Для каждого типа данных языков Fortran и C есть своя константа. Полный список предопределенных имен типов можно найти в файле mpi.h.

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Это означает, что после возврата из данной функции можно использовать любые присутствующие в вызове функции переменные без опасения испортить передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу `dest`, остается за разработчиками конкретной реализации MPI.

Следует специально отметить, что возврат из функции `MPI_Send` не означает ни того, что сообщение получено процессом `dest`, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, запустивший `MPI_Send`. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове данной функции.

Подобная неопределенность далеко не всегда устраивает пользователя. Чтобы расширить возможности передачи сообщений, в MPI введены дополнительные три функции. Все параметры у этих функций такие же, как и у функции `MPI_Send`, однако у каждой из них есть своя особенность.

MPI_Bsend — передача сообщения с буферизацией. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в буфер и произойдет немедленный возврат из функции. Выполнение данной функции никак не зависит от соответствующего вызова функции приема сообщения. Тем не менее, функция может вернуть код ошибки, если места под буфер недостаточно.

MPI_Ssend — передача сообщения с синхронизацией. Выход из данной функции произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Если прием сообщения также выполняется с блокировкой, то функция `MPI_Ssend` сохраняет семантику блокирующих вызовов.

MPI_Rsend — передача сообщения по готовности. Данной функцией можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов функции, вообще говоря, является ошибочным и результат ее выполнения не определен. Во многих реализациях функция `MPI_Rsend` сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи.

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

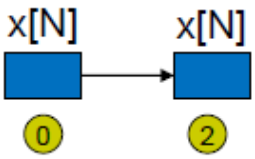
- `OUT buf` — адрес начала буфера для приема сообщения;
- `count` — максимальное число элементов в принимаемом сообщении;
- `datatype` — тип элементов принимаемого сообщения;
- `source` — номер процесса-отправителя;
- `msgtag` — идентификатор принимаемого сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT status` — параметры принятого сообщения.

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 принимаемом сообщении, то можно воспользоваться функциями `MPI_Probe` или `MPI_Get_count`.

Блокировка гарантирует, что после возврата из функции все элементы сообщения уже будут приняты и расположены в буфере `buf`.

Пример: передача данных от 0-го процесса 2-му

```
double x[N];
if (size>=3)
{ if (rank==0)
    MPI_Send(x,N,MPI_DOUBLE,2,123,comm);
  if (rank==2)
    MPI_Recv(x,N,MPI_DOUBLE,0,123,comm,MPI_STATUS_IGNORE);
}
```



Ниже приведен пример программы, в которой нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа. Если программа будет запущена с бóльшим числом процессов, то реально выполнять пересылки все равно станут только нулевой и первый процессы. Остальные процессы после их порождения функцией `MPI_Init` сразу завершатся, выполнив функцию `MPI_Finalize`.

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, src, rc, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

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

if (rank == 0) {
  dest = 1;
  src = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD,
      &Stat);
}
else
  if (rank == 1) {
    dest = 0;
    src = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, src, tag, MPI_COMM_WORLD,
         &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
MPI_Finalize();
}
```

В следующем примере каждый процесс с четным номером посылает сообщение своему соседу с номером, на единицу большим. Дополнительно поставлена проверка для процесса с максимальным номером, чтобы он не послал сообщение несуществующему процессу. Показана только схема программы.

```
main(int argc, char **argv)
{
int me, size;
int SOME_TAG=0;
MPI_Status status;
...
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &me);
MPI_Comm_size (MPI_COMM_WORLD, &size);
if ((me % 2) == 0)  {
  if ((me+1) < size)   /* посылают все процессы, кроме последнего */
     MPI_Send (..., me+1, SOME_TAG, MPI_COMM_WORLD);
}
else
  MPI_Recv (..., me-1, SOME_TAG, MPI_COMM_WORLD, &status);
...
MPI_Finalize();
}
```

При приеме сообщения 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` — это признак того, что подходит сообщение с любым идентификатором. При одновременном использовании этих двух констант будет принято любое сообщение от любого процесса.

Параметры принятого сообщения всегда можно определить по соответствующим полям структуры `status`. Предопределенный тип `MPI_Status` описан в файле mpi.h. В языке C параметр `status` является структурой, содержащей поля с именами `MPI_SOURCE`, `MPI_TAG` и `MPI_ERROR`. Реальные значения номера процесса-отправителя, идентификатора сообщения и кода ошибки доступны через `status.MPI_SOURCE`, `status.MPI_TAG` и `status.MPI_ERROR`. В языке Fortran параметр `status` является целочисленным массивом размера `MPI_STATUS_SIZE`. Константы `MPI_SOURCE`, `MPI_TAG` и `MPI_ERROR` являются индексами по данному массиву для доступа к значениям соответствующих полей, например, `status(MPI_SOURCE)`.

Обратим внимание на некоторую несимметричность операций посылки и приема сообщений. С помощью константы MPI_ANY_SOURCE можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

В стандарте оговорено, что если один процесс последовательно посылает два сообщения другому процессу, и оба эти сообщения соответствуют одному и тому же вызову MPI_Recv, то первым будет принято сообщение, которое было отправлено раньше. Вместе с тем, если два сообщения были одновременно отправлены разными процессами и оба сообщения соответствуют одному и тому же вызову MPI_Recv, то порядок их получения принимающим процессом заранее не определен.

MPI не гарантирует выполнения свойства "справедливости" при распределении приходящих сообщений. Предположим, что процесс P_1 послал сообщение, которое может быть принято процессом P_2. Однако прием может не произойти, в принципе, сколь угодно долгое время. Такое возможно, например, если процесс P_3 постоянно посылает сообщения процессу P_2, также подходящие под шаблон MPI_Recv.

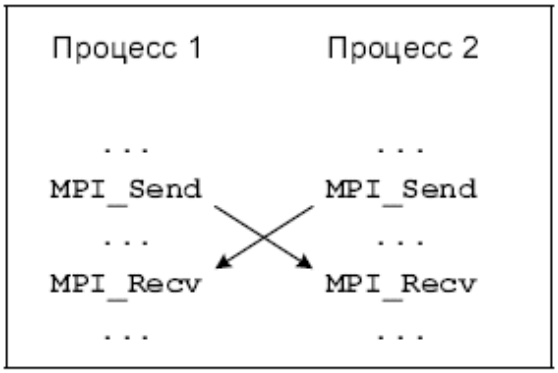


Рис. 5.5. Тупиковая ситуация при использовании блокирующих функций

Последнее замечание относительно использования блокирующих функций приема и посылки связано с возможным возникновением тупиковой ситуации. Предположим, что работают два параллельных процесса и они хотят обменяться данными. Было бы вполне естественно в каждом процессе сначала воспользоваться функцией MPI_Send, а затем MPI_Recv (схематично эта ситуация изображена на рис. 5.5). Но именно этого и не стоит делать. Дело в том, что мы заранее не знаем, как реализована функция MPI_Send. Если разработчики для гарантии корректного повторного использования буфера посылки заложили схему, при которой посылающий процесс ждет начала приема, то возникнет классический тупик. Первый процесс не может вернуться из функции посылки, поскольку второй не начинает прием сообщения. А второй процесс не может начать прием сообщения, поскольку сам по похожей причине застрял на посылке. Выход из этой ситуации прост. Нужно использовать либо неблокирующие функции приема/передачи, либо функцию совмещенной передачи и приема. Оба варианта мы обсудим ниже.

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

- `status` — параметры принятого сообщения;
- `datatype` — тип элементов принятого сообщения;
- `OUT count` — число элементов сообщения.

По значению параметра `status` данная функция определяет число уже принятых (после обращения к `MPI_Recv`) или принимаемых (после обращения к `MPI_Probe` или `MPI_Iprobe`) элементов сообщения типа `datatype`. Данная функция, в частности, необходима для определения размера области памяти, выделяемой для хранения принимаемого сообщения.

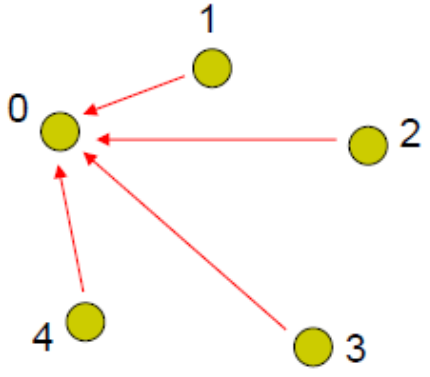
```
int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status)
```

- `source` — номер процесса-отправителя или `MPI_ANY_SOURCE`;
- `msgtag` — идентификатор ожидаемого сообщения или `MPI_ANY_TAG`;
- `comm` — идентификатор коммуникатора;
- `OUT status` — параметры найденного подходящего сообщения.

Получение информации о структуре ожидаемого сообщения с блокировкой. Возврата из функции 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 особо обратить внимание на то, что функция определяет только факт прихода сообщения, но реально его не принимает.

Пример: приём произвольного сообщения 0-м процессом

```
MPI_Status st;
char x[1000];
if (size>1)
{ if (rank==0)
  { int i,idx = 0;
    for (i=1;i<size;i++)
    { MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &st);
      MPI_Recv(x+idx, st.count, MPI_BYTE,
               st.MPI_SOURCE, st.MPI_TAG, comm, MPI_STATUS_IGNORE);
      idx += st.count;
    }
  }
  else
    MPI_Send(x,rank,MPI_CHAR,0,rank,comm);
}
```



Прием/передача сообщений без блокировки. В отличие от функций с блокировкой, возврат из функций данной группы происходит сразу без какой-либо блокировки процессов. На фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции. В принципе, данная возможность исключительно полезна для создания эффективных программ. В самом деле, программист знает, что в некоторый момент ему потребуется массив, который вычисляет другой процесс. Он заранее выставляет в программе асинхронный запрос на получение данного массива, а до того момента, когда массив реально потребуется, он может выполнять любую другую полезную работу. Опять же, во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения для выполнения последующих вычислений.

Если есть возможность операции приема/передачи сообщений скрыть на фоне вычислений, то этим, вроде бы, надо безоговорочно пользоваться. Однако на практике не все согласуется с теорией. Многое зависит от конкретной реализации. К сожалению, далеко не всегда асинхронные операции эффективно поддерживаются аппаратурой и системным окружением. Поэтому не стоит удивляться, если эффект от выполнения вычислений на фоне пересылок окажется нулевым. Стандарт стандартом, но вы работаете в конкретной среде, на конкретной вычислительной системе. А справедливости ради заметим, что стандарт эффективности и не обещал...

Сделанные замечания касаются только вопросов эффективности. В отношении предоставляемой функциональности асинхронные операции исключительно полезны, поэтому они присутствуют практически в каждой реальной программе.

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

- `buf` — адрес начала буфера с посылаемым сообщением;
- `count` — число передаваемых элементов в сообщении;
- `datatype` — тип передаваемых элементов;
- `dest` — номер процесса-получателя;
- `msgtag` — идентификатор сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT request` — идентификатор асинхронной операции.

Передача сообщения аналогична вызову `MPI_Send`, однако возврат из функции `MPI_Isend` происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере `buf`. Это означает, что нельзя что-то записывать в данный буфер без получения дополнительной информации, подтверждающей завершение посылки. Определить тот момент времени, когда можно повторно использовать буфер `buf` без опасения испортить передаваемое сообщение, можно с помощью параметра `request` и функций `MPI_Wait` и `MPI_Test`.

Аналогично трем модификациям функции `MPI_Send`, предусмотрены три дополнительных варианта функций `MPI_Ibsend`, `MPI_Issend`, `MPI_Irsend`. К изложенной выше семантике работы этих функций добавляется асинхронность.

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

- `OUT buf` — адрес начала буфера для приема сообщения;
- `count` — максимальное число элементов в принимаемом сообщении;
- `datatype` — тип элементов принимаемого сообщения;
- `source` — номер процесса-отправителя;
- `msgtag` — идентификатор принимаемого сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT request` — идентификатор операции асинхронного приема сообщения.

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

Сообщение, отправленное любой из функций MPI_Send, MPI_Isend и любой из трех их модификаций, может быть принято любой из процедур MPI_Recv и MPI_Irecv.

С помощью данной функции легко обойти возможную тупиковую ситуацию, показанную на рис. 5.5. Заменим вызов функции приема сообщения с блокировкой на вызов функции MPI_Irecv. Расположим его перед вызовом функции MPI_Send, т. е. преобразуем фрагмент:

```
...
MPI_Send (...)
MPI_Recv(...)
...
```

следующим образом

```
...
MPI_Irecv (...)
MPI_Send (...)
...
```

В такой ситуации тупик гарантированно не возникнет, поскольку к моменту вызова функции MPI_Send запрос на прием сообщения уже будет выставлен.

int MPI_Wait(MPI_Request *request, MPI_Status *status)

- request — идентификатор операции асинхронного приема или передачи;
- OUT status — параметры сообщения.

Ожидание завершения асинхронной операции, ассоциированной с идентификатором request и запущенной функциями MPI_Isend или MPI_Irecv. Пока асинхронная операция не будет завершена, процесс, выполнивший функцию MPI_Wait, будет заблокирован. Если речь идет о приеме, атрибуты и длину принятого сообщения можно определить обычным образом с помощью параметра status.

int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses)

- count — число идентификаторов асинхронных операций;
- requests — идентификаторы операций асинхронного приема или передачи;
- OUT statuses — параметры сообщений.

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.

Ниже показан пример программы, в которой все процессы обмениваются сообщениями с ближайшими соседями в соответствии с топологией кольца.

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

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

prev = rank - 1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD,
          &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD,
           &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

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

- `count` — число идентификаторов асинхронных операций;
- `requests` — идентификаторы операций асинхронного приема или передачи;
- `OUT index` — номер завершенной операции обмена;
- `OUT status` — параметры сообщения.

- OUT flag — признак завершенности операций обмена;
- OUT statuses — параметры сообщений.

В параметре flag функция возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены. В этом случае параметры сообщений будут указаны в массиве statuses. Если какая-либо из операций не завершилась, то возвращается 0, и определенность элементов массива statuses не гарантируется.
Выполнение процесса блокируется до тех пор, пока какая-либо асинхронная операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если завершились несколько операций, то случайным образом будет выбрана 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)
```

- incount — число идентификаторов асинхронных операций;
- requests — идентификаторы операций асинхронного приема или передачи;
- OUT outcount — число идентификаторов завершившихся операций обмена;
- OUT indexes — массив номеров завершившихся операций обмена;
- OUT statuses — параметры завершившихся операций приема сообщений.

Выполнение процесса блокируется до тех пор, пока 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)
```

- request — идентификатор операции асинхронного приема или передачи;
- OUT flag — признак завершенности операции обмена;
- OUT status — параметры сообщения.

Проверка завершенности асинхронных функций MPI_Isend или MPI_Irecv, ассоциированных с идентификатором request. В параметре flag функция MPI_Test 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)
```

- count — число идентификаторов асинхронных операций;
- requests — идентификаторы операций асинхронного приема или передачи;

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

- `count` — число идентификаторов асинхронных операций;
- `requests` — идентификаторы операций асинхронного приема или передачи;
- `OUT index` — номер завершенной операции обмена;
- `OUT flag` — признак завершенности операции обмена;
- `OUT status` — параметры сообщения.

Если к моменту вызова функции `MPI_Testany` хотя бы одна из операций асинхронного обмена завершилась, то в параметре `flag` возвращается значение 1, `index` содержит номер соответствующего элемента в массиве `requests`, а `status` — параметры сообщения. В противном случае в параметре `flag` будет возвращено значение 0.

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

- `incount` — число идентификаторов асинхронных операций;
- `requests` — идентификаторы операций асинхронного приема или передачи;
- `OUT outcount` — число идентификаторов завершившихся операций обмена;
- `OUT indexes` — массив номеров завершившихся операций обмена;
- `OUT statuses` — параметры завершившихся операций приема сообщений.

Данная функция работает так же, как и `MPI_Waitsome`, 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)
```

- `source` — номер процесса-отправителя или `MPI_ANY_SOURCE`;
- `msgtag` — идентификатор ожидаемого сообщения или `MPI_ANY_TAG`;
- `comm` — идентификатор коммуникатора;
- `OUT flag` — признак завершенности операции обмена;
- `OUT status` — параметры подходящего сообщения.

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре `flag` возвращается 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.

Пример: передача по кольцу на фоне счёта

```
int x[N],y[N],done;
MPI_Request reqs,reqr;
MPI_Irecv(x,N,MPI_INT, (rank+size-1)%size,123, comm, &reqr);
MPI_Isend(y,N,MPI_INT, (rank+1)%size,123, comm, &reqs);
do
{ do_some_work();
  MPI_Test(reqr, &done, MPI_STATUS_IGNORE);
} while (!done);
MPI_Wait(reqs, MPI_STATUS_IGNORE);
```

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)
```

- `buf` — адрес начала буфера с посылаемым сообщением;
- `count` — число передаваемых элементов в сообщении;
- `datatype` — тип передаваемых элементов;
- `dest` — номер процесса-получателя;
- `msgtag` — идентификатор сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT request` — идентификатор асинхронной передачи.

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`. Как и прежде, дополнительно предусмотрены варианты и для трех модификаций посылки сообщений: `MPI_Bsend_init`, `MPI_Ssend_init`, `MPI_Rsend_init`.

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

- `OUT buf` — адрес начала буфера приема сообщения;
- `count` — число принимаемых элементов в сообщении;
- `datatype` — тип принимаемых элементов;
- `source` — номер процесса-отправителя;
- `msgtag` — идентификатор сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT request` — идентификатор асинхронного приема.

Формирование запроса на выполнение приема сообщения. Все параметры точно такие же, как и у функции `MPI_Irecv`, однако, в отличие от нее, реальный прием не начинается до вызова функции `MPI_Startall`.

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

- `count` — число запросов на взаимодействие;
- `OUT requests` — массив идентификаторов приема/передачи.

Запуск всех отложенных операций передачи и приема, ассоциированных с элементами массива запросов `requests` и инициированных функциями `MPI_Recv_init`, `MPI_Send_init` или ее тремя модификациями. Все отложенные взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью функций семейств `MPI_Wait` и `MPI_Test`.

Совмещенные прием и передача сообщений. Совмещение приема и передачи сообщений между процессами позволяет легко обходить множество подводных камней, связанных с возможными тупиковыми ситуациями. Предположим, что в линейке процессов необходимо организовать обмен данными между i-м и $i + 1$-м процессами. Если воспользоваться стандартными блокирующими функциями посылки сообщений, то возможен тупик, обсуждавшийся ранее. Один из способов обхода такой ситуации состоит в использовании функции совмещенного приема и передачи.

```
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)
```

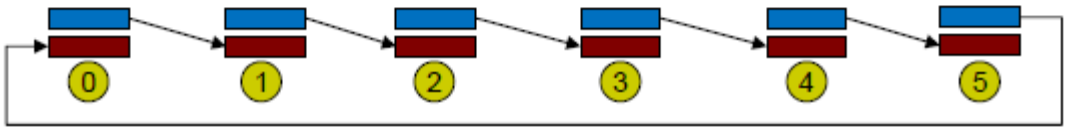
- `sbuf` — адрес начала буфера с посылаемым сообщением;
- `scount` — число передаваемых элементов в сообщении;
- `stype` — тип передаваемых элементов;
- `dest` — номер процесса-получателя;
- `stag` — идентификатор посылаемого сообщения;
- `OUT rbuf` — адрес начала буфера приема сообщения;
- `rcount` — число принимаемых элементов сообщения;
- `rtype` — тип принимаемых элементов;
- `source` — номер процесса-отправителя;
- `rtag` — идентификатор принимаемого сообщения;
- `comm` — идентификатор коммуникатора;
- `OUT status` — параметры принятого сообщения.

Данная операция объединяет в едином запросе посылку и прием сообщений. Естественно, что реализация этой функции гарантирует отсутствие тупиков, которые могли бы возникнуть между процессами при использовании обычных блокирующих операций `MPI_Send` и `MPI_Recv`.

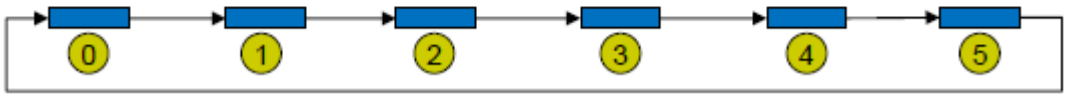
Принимающий и отправляющий процессы могут являться одним и тем же процессом. Буфера приема и посылки обязательно должны быть различными. Сообщение, отправленное операцией `MPI_Sendrecv`, может быть принято обычным образом, и точно так же операция `MPI_Sendrecv` может принять сообщение, отправленное обычной операцией `MPI_Send`.

Пример: сдвиг данных по кольцу

```
double x[N],y[N];
MPI_Sendrecv(x,N,MPI_DOUBLE,(rank+1)%size,123,
             y,N,MPI_DOUBLE,(rank+size-1)%size,123,
             MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```




```
MPI_Sendrecv_replace(x,N,MPI_DOUBLE,
            (rank+1)%size,123, (rank+size-1)%size,123,
            MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```



Многократно повторяющиеся операции. Пример:

```
MPI_Request req[2];
int x[N],y[N],i;

MPI_Send_init(x,N,MPI_INT,(rank+1)%size,123,comm,&req[0]);
MPI_Recv_init(y,N,MPI_INT,(rank+size-1)%size,
                                        123,comm,&req[1]);
for (i=0;i<100;i++)
{
  do_some_work();
  MPI_Startall(2, req);
  do_more_work();
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
}
```

Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов *участвуют все процессы коммуникатора*. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. 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. Асинхронных коллективных операций в MPI нет.

В коллективных операциях можно использовать те же коммуникаторы, что и были использованы для операций типа "точка-точка". MPI гарантирует, что сообщения, вызванные коллективными операциями, никак не повлияют и не пересекутся с сообщениями, появившимися в результате индивидуального взаимодействия процессов.

Вообще говоря, нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций. Если какой-то процесс уже завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (конечно же, если это возможно по смыслу операции).

В коллективных операциях не используются идентификаторы сообщений.

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

- `OUT buf` — адрес начала буфера посылки сообщения;
- `count` — число передаваемых элементов в сообщении;
- `datatype` — тип передаваемых элементов;
- `source` — номер рассылающего процесса;
- `comm` — идентификатор коммуникатора.

Рассылка сообщения от процесса `source` всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера `buf` процесса `source` будет скопировано в локальный буфер каждого процесса коммуникатора `comm`. Значения параметров `count`, `datatype`, `source` и `comm` должны быть одинаковыми у всех процессов. В результате выполнения следующего оператора всеми процессами коммуникатора `comm`:

```
MPI_Bcast(array, 100, MPI_INT, 0, comm);
```

первые сто целых чисел из массива `array` нулевого процесса будут скопированы в локальные буфера `array` каждого процесса.

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

- `sbuf` — адрес начала буфера посылки;
- `scount` — число элементов в посылаемом сообщении;
- `stype` — тип элементов отсылаемого сообщения;
- `OUT rbuf` — адрес начала буфера сборки данных;
- `rcount` — число элементов в принимаемом сообщении;
- `rtype` — тип элементов принимаемого сообщения;
- `dest` — номер процесса, на котором происходит сборка данных;
- `comm` — идентификатор коммуникатора.

Сборка 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. На процессе `dest` существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров `sbuf`, `scount`, `stype`, `dest` и `comm`. Значения параметров `dest` и `comm` должны быть одинаковыми у всех процессов. Параметр `rcount` у процесса `dest` обозначает число элементов типа `rtype`, принимаемых не от всех процессов в сумме, а от каждого процесса. С помощью похожей функции `MPI_Gatherv` можно принимать от процессов массивы данных различной длины.

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void *rbuf,
int rcount, MPI_Datatype rtype, int source, MPI_Comm comm)
```

- `sbuf` — адрес начала буфера посылки;
- `scount` — число элементов в посылаемом сообщении;
- `stype` — тип элементов отсылаемого сообщения;
- `OUT rbuf` — адрес начала буфера сборки данных;
- `rcount` — число элементов в принимаемом сообщении;
- `rtype` — тип элементов принимаемого сообщения;
- `source` — номер процесса, на котором происходит сборка данных;
- `comm` — идентификатор коммуникатора.

Функция `MPI_Scatter` по своему действию является обратной к `MPI_Gather`. Процесс `source` рассылает порции данных из массива `sbuf` всем *n* процессам приложения. Можно считать, что массив `sbuf` делится на *n* равных частей, состоящих из `scount` элементов типа `stype`, после чего *i*-я часть посылается *i*-му процессу. На процессе `source` существенными являются значения всех параметров, а на всех остальных процессах — только значения параметров `rbuf`, `rcount`, `rtype`, `source` и `comm`. Значения параметров `source` и `comm` должны быть одинаковыми у всех процессов.

Также по аналогии существуют функции MPI_Allgather и MPI_Alltoall, осуществляющие соответственно сборку распределенных по всем процессам данных во всех процессах и обмен данными «все со всеми».

Аналогично функции `MPI_Gatherv`, с помощью функции `MPI_Scatterv` процессам можно отослать порции данных различной длины.

В следующем примере показано использование функции `MPI_Scatter` для рассылки строк массива. Напомним, что в языке C, в отличие от Fortran, массивы хранятся в памяти по строкам.

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];
{
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

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

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount,
          MPI_FLOAT, source, MPI_COMM_WORLD);
```

```
  printf("rank= %d  Results: %f %f %f %f\n", rank, recvbuf[0],
         recvbuf[1], recvbuf[2], recvbuf[3]);
  }
else
  printf("Число процессов должно быть равно %d. \n", SIZE);

MPI_Finalize();
}
```

К коллективным операциям относятся и редукционные операции. Такие операции предполагают, что на каждом процессе хранятся некоторые данные, над которыми необходимо выполнить единую операцию, например, операцию сложения чисел или операцию нахождения максимального значения. Операция может быть либо предопределенной операцией MPI, либо операцией, определенной пользователем. Каждая предопределенная операция имеет свое имя, например, `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD` и т. п.

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

- `sbuf` — адрес начала буфера для аргументов операции `op`;
- `OUT rbuf` — адрес начала буфера для результата операции `op`;
- `count` — число аргументов у каждого процесса;
- `datatype` — тип аргументов;
- `op` — идентификатор глобальной операции;
- `comm` — идентификатор коммуникатора.

Данная функция задает выполнение `count` независимых глобальных операций `op`. Предполагается, что в буфере `sbuf` каждого процесса расположено `count` аргументов, имеющих тип `datatype`. Первые элементы массивов `sbuf` участвуют в первой операции `op`, вторые элементы массивов `sbuf` участвуют во второй операции `op` и т. д. Результаты выполнения всех `count` операций записываются в буфер `rbuf` на каждом процессе. Значения параметров `count`, `datatype`, `op` и `comm` 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)`

- `sbuf` — адрес начала буфера для аргументов;
- `OUT rbuf` — адрес начала буфера для результата;
- `count` — число аргументов у каждого процесса;
- `datatype` — тип аргументов;

- `op` — идентификатор глобальной операции;
- `root` — процесс-получатель результата;
- `comm` — идентификатор коммуникатора.

Функция аналогична предыдущей, но результат операции будет записан в буфер `rbuf` не у всех процессов, а только у процесса `root`.

Пример: вычисление скалярного произведения

```
double x[N],y[N], s=0.0, sum;
for (i=0;i<N;i++) s += x[i]*y[i];
MPI_Reduce(&s,&sum,1,MPI_DOUBLE,
           MPI_SUM,0,comm);
if (rank==0) printf("Result: %lf\n",sum);
```

Синхронизация процессов

Синхронизация процессов в MPI осуществляется с помощью единственной функции `MPI_Barrier`.

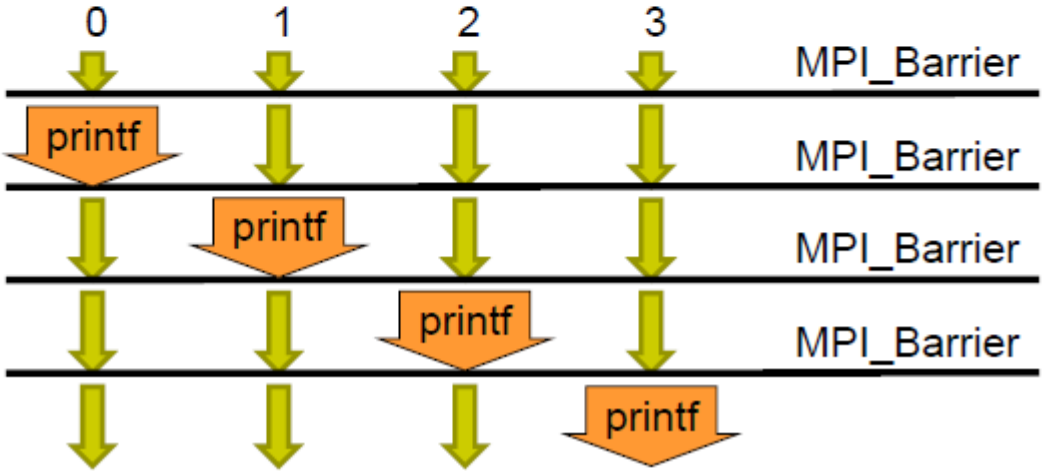
```
int MPI_Barrier(MPI_Comm comm)
```

`comm` — идентификатор коммуникатора.

Функция блокирует работу вызвавших ее процессов до тех пор, пока все оставшиеся процессы коммуникатора `comm` также не выполнят эту процедуру. Только после того, как последний процесс коммуникатора выполнит данную функцию, все процессы будут разблокированы и продолжат выполнение дальше. Данная функция является коллективной. Все процессы должны вызвать `MPI_Barrier`, хотя реально исполненные вызовы различными процессами коммуникатора могут быть расположены в разных местах программы.

Пример: упорядоченный вывод

```
int size,rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
for (i=0;i<size;i++)
{ MPI_Barrier(MPI_COMM_WORLD);
  if (rank==i) printf("%d\n", rank);
}
```



Работа с группами процессов

Несмотря на то, что в MPI есть значительное множество функций, ориентированных на работу с коммуникаторами и группами процессов, мы не будем подробно останавливаться на данном разделе. Представленная функциональность позволяет сравнить состав групп, определить их пересечение, объединение, добавить процессы в группу, удалить группу и т. д. В качестве примера приведем лишь один способ образования новых групп на основе существующих.

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

- `comm` — идентификатор существующего коммуникатора;
- `color` — признак разделения на группы;
- `key` — параметр, определяющий нумерацию в новых группах;
- `OUT newcomm` — идентификатор нового коммуникатора.

Данная процедура разбивает все множество процессов, входящих в группу `comm`, на непересекающиеся подгруппы — одну подгруппу на каждое значение параметра `color`. Значение параметра `color` должно быть неотрицательным целым числом. Каждая новая подгруппа содержит все процессы, у которых параметр `color` имеет одно и то же значение, т. е. в каждой новой подгруппе будут собраны все процессы "одного цвета". Всем процессам подгруппы будет возвращено одно и то же значение `newcomm`. Параметр `key` определяет нумерацию в новой подгруппе.

Предположим, что нужно разделить все процессы программы на две подгруппы в зависимости от того, является ли номер процесса четным или нечетным. В этом случае в поле `color` достаточно поместить выражение `My_Id%2`, где `My_Id` — это номер процесса. Значением данного выражения может быть либо 0, либо 1. Все процессы с четными номерами автоматически попадут в одну группу, а процессы с нечетными в другую.

```
int MPI_Comm_free(MPI_Comm comm)
```

`OUT comm` — идентификатор коммуникатора.

Появление нового коммуникатора всегда вызывает создание новой структуры данных. Если созданный коммуникатор больше не нужен, то соответствующую область памяти необходимо освободить. Данная функция уничтожает коммуникатор, ассоциированный с идентификатором `comm`, который после возвращения из функции будет иметь значение `MPI_COMM_NULL`.

Компиляция MPI-программ:
C: mpicc -o prog prog.c
C++: mpicxx -o prog prog.cpp
Запуск MPI-программ:
mpirun -np 4 ./prog
mpiexec -n 4 ./prog
MPI-типы данных

- Обеспечивают правильное преобразование данных при пересылке с одной архитектуры на другую
  - Стандартные MPI-типы

MPI-тип	C тип	MPI-тип	C тип
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short int	MPI_UNSIGNED_SHORT	unsigned short int
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long int	MPI_UNSIGNED_LONG	unsigned long int
MPI_LONG_LONG	long long int	MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float	MPI_DOUBLE	double
MPI_BYTE	-	MPI_LONG_DOUBLE	long double

  - Пользовательские MPI-типы
    - Конструируются пользователем из базовых типов

MPI-ввод-вывод

- MPI-IO используется для одновременного чтения/записи файлов многими процессами
- Базовые операции:
  - MPI_File_open
  - MPI_File_read
  - MPI_File_write
  - MPI_File_close