Введение в технологию MPI.

Хохлов Н. И.

МФТИ

1 Введение

Технология MPI является одной из наиболее распространенных технологий для написания параллельных программ в системах с распределенной памятью. MPI расшифровывается как Message Passing Interface (интерфейс передачи сообщений). Исходя из названия, основным способом взаимодействия процессов в MPI является передача сообщений между процессами. MPI это спецификация и определяет интерфейс, которым должны руководствоваться программисты при написании параллельных программ. Основная цель данной технологии - предоставить широко используемый стандарт для написания программ, использующих обмен данными между процессами, в частности параллельных программ в системах с распределенной памятью.

История MPI берет свое начало из 90-х годов прошлого века. В 1980-1990 годах существовало множество технологий для написания параллельных программ в системах с распределенной памятью, все они имели различный интерфейс и были сложны в портировании. В 1992 году начались вести работы по стандартизации интерфейсов передачи сообщений, и в 1993 году появилась первая версия стандарта, а в 1994 - финальная версия, она имеет название MPI-1.1. Затем появилась вторая версия стандарта, MPI-2, это было в 1996 году. Она включила в себя весь функционал MPI-1.1 и несколько расширила его. На данный момент ведется разработка интерфейса MPI-3. Стандартизацией интерфейса занимается открытое сообщество MPI Forum [1]. Далее по тексту, если не оговорено другое, речь будет идти о стандарте MPI-1.1.

MPI предоставляет интерфейсы для языков C и Fortran и содержит более 115 функций. Однако распараллелить большинство программ, которые параллелятся в системах с распределенной памятью, можно при помощи всего лишь 5-6 функций MPI. При написании программ используя данную технологию, не возникает проблем при переносе с одной платформы на другую, поскольку везде интерфейс библиотек одинаковый, то все решается перекомпиляцией. Существуют различные реализации стандарта MPI, как открытые, так и коммерческие. Из наиболее известных это OpenMPI [2] - наиболее распространенная версия из открытых реализаций, MPICH [3] - реализация MPI для сети Myrinet, MVAPICH [4] - реализация MPI для сети InfiniBand.

MPI предоставляет абстракцию над любой архитектурой с разделяемой памятью [5], при этом выполнятся MPI-программа может как на вычислительных системах с распределенной памятью, так на системах с общей памятью и гибридных.

2 Компиляция и запуск

При компиляции программ, использующих библиотеки MPI, необходимо указать пути до

заголовочных файлов, пути до библиотек и прилинковать соответствующие библиотечные модули. Спецификация MPI не стандартизирует путей в системе, по которым должны находится модули MPI, и в зависимости от реализации они могут находится в совершенно различных системных папках, помимо этого названия библиотек также меняются в различных реализациях. Чтобы избежать проблем при компиляции и переносимости кода на другие платформы, реализации MPI предоставляют обертки поверх системных компиляторов в системе. Так для языка C компилятор называется **mpicc**, для языка C++ — **mpiCC, mpicxx, mpic++** (в зависимости от реализации присутствуют те или иные команды), для языков Фортран 77 и 90 — **mpif77** и **mpif90** соответственно. Следует учитывать, что это не отдельный компилятор, а вызов системного компилятора (gcc, icc и др.) с некоторым набором опций. Соответственно все опции, которые будут указаны компилятору MPI, будут переданы системному компилятору.

Все описание интерфейса MPI собраны в одном заголовочном файле, он называется **mpi.h** для языков C/C++ либо **mpif.h** для языка Фортран, поэтому в начале MPI–программ должна стоять директива **#include <mpi.h>** для языка C/C++ и **include 'mpif.h'** для языка Фортран.

Запуск MPI–программ также несколько отличается от запуска обычных программ. Для запуска программы в несколько потоков обычно предоставляются команды запуска MPI–приложений **mpirun** или **mpiexec**. Опции данных команд могут несколько различаться в различных версиях библиотек, однако в большинстве случаев программу можно запустить следующим образом:

```
mpirun -np N [programm_with_arguments],
```

где N — число процессов, каждый из которых будет представлять собой экземпляр запущенной программы, работающих одновременно. Вывод работы всех программ MPI–окружение собирает и выводит на экран, при этом MPI гарантирует, что внутри одной строки вывода будет вывод только одного процесса.

3 Процессы и коммуникаторы

Работающая MPI–программа представляет собой набор одновременно исполняемых процессов, при этом процессы порождаются один раз при запуске. Во время работы нельзя ни порождать новые MPI–процессы, ни уничтожать. Как уже говорилось ранее, стандарт MPI предназначен для систем с распределенной памятью, потому каждый процесс имеет свое адресное пространство, и нету никаких общих данных либо переменных между ними, все взаимодействие осуществляется путем передачи сообщений.

Для организации процессов в MPI существуют специальные объекты — *группы процессов* и *коммуникаторы*. Для локализации набора процессов их объединяют в отдельные группы, каждой группе приписывается среда для обмена сообщениями — коммуникатор. Коммуникатор непосредственно связан с группой, поэтому далее по тексту, где это возможно, будем оперировать понятиями коммуникаторов, подразумевая не только среду для общения, но также и группу процессов. Процесс может входить в любое число коммуникаторов, однако должен входить хотя бы в один. Группы процессов могут пересекаться, полностью совпадать, не пересекаться или быть одна частью другой. Любое общение между процессами может проходить только в рамках одного коммуникатора, сообщения отправленные в разных коммуникаторах отсылаются независимо и никак не взаимодействуют друг с другом.

Большинство функций MPI принимает в качестве одного из аргументов коммуникатор, в рамках которого идет обмен сообщениями. В языке С для него существует специальный тип **MPI_Comm**.

При запуске программы все процессы включены в один общий коммуникатор, он представлен в виде глобальной переменной **MPI_COMM_WORLD**. Размер этого коммуникатора (число процессов в нем) равен числу запущенных процессов. Кроме того при старте программы существуют коммуникаторы **MPI_COMM_SELF** и **MPI_COMM_NULL,** содержащие только один текущий процесс и не содержащий ни одного процесса соответственно.

Для адресации процессов в MPI внутри коммуникатора каждому процессу в нем присвоен уникальный номер, он называется *идентификатор процессора*, *номер процессора* или *ранк (rank)* (далее по тексту возможно использование любых из этих понятий, подразумевая одно и тоже). Внутри коммуникатора номер процесса уникален, причем номера процессов всегда идут последовательно от **0** до **N-1**, где **N** — размер коммуникатора. Если процесс входит в более чем один коммуникатор, то его номера в разных коммуникаторах могут не совпадать.

4 Структура библиотеки MPI

Все процедуры MPI можно выделить в несколько групп, прежде всего это процедуры инициализации MPI, порождения и завершения параллельных процессов. Затем идут процедуры взаимодействия процессов, они делятся на две группы — процедуры типа точка-точка, в данных типах процедур участвуют только два процесса, и коллективные процедуры, в которых участвуют все процессы коммуникатора. Также в MPI присутствует набор функционала для создания собственных типов данных, организации групп процессов, коммуникаторов и создания виртуальных топологий. В данной работе мы коснемся только основных процедур MPI, однако при их помощи уже можно распараллелить большинство программ в распределенной памяти. Рассмотрим далее подробнее некоторые процедуры MPI.

5 Общие процедуры MPI

В данном разделе рассмотрим основные процедуры библиотеки MPI не связанные с взаимодействиями процессов, а также рассмотрим общую структуру MPI–программ.

Все объекты MPI, как то функции, типы данных, предопределенные константы и т. д. начинаются с префикса **MPI_**. Это сделано для избежания конфликтов имен функций MPI с функциями других библиотек и именами используемыми в программе.

Рассмотрим простейшую MPI–программу на языке С:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rc;
  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
    printf ("Error starting MPI program. Terminating.\n");
```

```
      MPI_Abort(MPI_COMM_WORLD, rc);
    }
    /* Parallel section */
    MPI_Finalize();
    return 0;
}
```

В приведенной выше программе идет инициализация MPI окружения, после этого возможно выполнение какого-либо параллельного кода, по завершению вычислений необходимо корректно завершить работу с MPI. Рассмотрим подробнее функции MPI, используемые в данной программе.

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

Инициализация MPI окружения и начало параллельной секции программы, должна был вызвана в любой MPI–программе, причем только один раз. Все другие функции MPI (за исключением нескольких утильных функций) должны вызываться только после инициализации. На вход функция принимает аргументы функции **main** — **argc** и **argv**, через них можно передавать некоторые параметры запуска программы, эти параметры не стандартизированы и зависят от реализации библиотеки.

Любая процедура MPI возвращает код выполнения, в случае успешного завершения он равен **MPI_SUCCESS**, как обрабатывать ошибки показано в примере. Далее обработка ошибок будет опускаться с целью не загромождения примеров лишним кодом.

int MPI_Finalize()

Завершение работы MPI окружения. Должна вызываться на всех запущенных процессах, после вызова данной функции нельзя вызывать другие процедуры MPI.

int MPI_Abort(MPI_Comm comm, int errorcode)

Аварийное завершение всех MPI процессов, связанных данным коммуникатором. В некоторых реализациях происходит аварийное завершение всех запущенных MPI процессов. Аргумент **errorcode** определяет код возврата, с которым завершится программа.

Рассмотрим некоторые другие функции MPI.

int MPI_Comm_size(MPI_Comm comm, int *size)

Определить размер группы процессов, связанной с коммуникатором. На вход принимает коммуникатор **comm**, число процессов записывает в переменную **size**. В частности при использовании с коммуникатором **MPI_COMM_WORLD** возвращает число запущенных процессов.

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Возвращает номер процесса **rank** в указанном коммуникаторе **comm**. Как уже указывалось ранее, **rank** может принимать значение от 0 до size-1, где **size** — значение полученное функцией **MPI_Comm_size** для того же коммуникатора.

Рассмотренные процедуры присутствуют практически в каждой MPI программе, далее рассмотрим еще несколько процедур.

int MPI_Initialized(int *flag)

Возвращает **flag**, который содержит 1, если в программе уже был вызов **MPI_Init**, и 0 в противном случае. Если программа имеет несколько модулей, и какие-то из них используют

MPI, то они могут проверить была ли уже произведена инициализация или нет, и по необходимости вызвать **MPI_Init**.

double MPI_Wtime()

Возвращает время в секундах начиная с некоторого момента времени в прошлом на вызывающем процессе. Момент времени, от которого идет отчет, остается постоянным на всем продолжении работы процесса. С помощью данной функции удобно замерять время исполнения отдельных частей кода, для этого необходимо окружить участок программы вызовами данной функции и взять разность возвращаемых значений. Если выставлен параметр **MPI_WTIME_IS_GLOBAL** и его значение равно 1, то момент времени, от которого идет отчет, дополнительно синхронизирован между всеми процессами.

double MPI_Wtick()

Возвращает разрешение таймера **MPI_Wtime** в секундах.

int MPI_Get_processor_name(char *name, int *resultlen)

Возвращает название процессора в переменную **name**, а также длину строки этого название в переменную **resultlen**. Размер входного буфера **name** должен быть как минимум **MPI_MAX_PROCESSOR_NAME**. Название уникальное для каждого из вычислительного узла, на которых ведется расчет, зависит от реализации и может не совпадать с именем хоста.

6 Взаимодействия типа точка-точка

В данном разделе рассмотрим взаимодействия типа *точка–точка*. Данные операции позволяют производить обмен данными между двумя и только двумя процессами, причем один процесс отправляет сообщение, а другой принимает. В MPI существуют несколько типов пересылок, процесс–отправитель сообщения должен явно вызвать одну из процедур отправки сообщения, причем необходимо явно указать номер процесса-получателя сообщения, процесс–получатель также должен явно вызвать одну из процедур приема, однако он может как указывать номер процесса, от которого получает сообщение, так может и получить сообщение от любого процесса. Типы процедур приема–пересылки могут быть различные у процесса–отправителя и процесса–получателя, однако все взаимодействие должно проходить в рамках одного коммуникатора.

Библиотека MPI гарантирует порядок прихода сообщений от одного процесса другому. Например, если процесс–отправитель послал последовательно два сообщения 1 и 2, то процесс–получатель гарантированно получит их в том же порядке — 1 и 2, однако это не гарантируется в случае использования наряду с технологией MPI технологий для распараллеливания в системах с общей памятью. Если же к одному и тому же процессу одновременно были отправлены сообщения от двух разных процессов, то порядок прихода сообщений в этом случае может быть любой.

Все процедуры данной категории делятся на *блокирующие* и *неблокирующие (асинхронные)*. Рассмотрим подробнее процедуры из каждой категории.

6.1 Блокирующие отсылка/прием

Блокирующие операции приостанавливают работу процесса до выполнения некоторого условия, которое зависит от типа операции, обычно до тех пор, пока буфер отсылаемого сообщения не будет безопасен для дальнейшего использования, т. е. никакие изменения

данного буфера не будут видны на принимаемой стороне.

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

Основная операция блокирующей отсылки. Осуществляется отсылка сообщения **buf**, размер которого count типов типа **datatype** (массив), номер процесса-получателя **dest**, сообщение имеет идентификатор **tag** и обмен ведется в рамках коммуникатора **comm**. Идентификатор сообщения должен быть один и тот же на отправляющей и принимающей стороне. Возврат из функции происходит тогда, когда буфер **buf** безопасен для дальнейшего использования на процессе–отправителе. Следует заметить, что в различных реализациях библиотек поведение функции может быть несколько различным, в частности допускается реализация, когда возврат из функции происходит после получения сообщения процессом–получателем. Операция может быть вызвана независимо от того, начат прием или нет. Размер **count** может быть равен 0. Тип данных **datatype** — переменная типа **MPI_Datatype**, специального типа MPI, определяющего тип данных в буфере. Для встроенных типов данных есть набор предопределенных констант, представленных в таблице (для языка C).

Тип данных в C	Тип данных в MPI
int	MPI_INT
float	MPI_FLOAT
double	MPI_DOUBLE
char	MPI_CHAR
long	MPI_LONG
short	MPI_SHORT
unsigned char	MPI_UNSIGNED_CHAR
unsigned int	MPI_UNSIGNED_INT
unsigned short	MPI_UNSIGNED_SHORT
unsigned long	MPI_UNSIGNED_LONG
long double	MPI_LONG_DOUBLE
8 бит	MPI_BYTE
Упакованные данные.	MPI_PACKED

Есть возможность расширять данные типы, создавая на основе них новые. В качестве аргумента **dest** может быть указан номер процесса–отправителя, т. е. можно послать сообщение себе, однако в некоторых случаях это может привести к состоянию блокировки. Также можно использовать специальное значение **MPI_PROC_NULL** для передачи сообщения несуществующему процессу. В таком случае функций вернет значение **MPI_SUCCESS** немедленно.

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

Блокирующий прием сообщения, прием осуществляется в буфер **buf** сообщения размером не более чем count, состоящего из типов **datatype**, от процесса с номером **source**, идентификатором сообщения **tag** и в рамках коммуникатора **comm**. При приеме сообщения заполняется специальная структура типа **MPI_Status** — статус сообщения, хранящая некую информацию о принятых данных. Размер принятого сообщения может быть меньше чем count, однако в случае если оно будет больше, функция вернет ошибку и сообщение не будет

принято. Блокировка гарантирует, что после завершения работы функции, все данные будут приняты в буфер **buf** и с ними можно работать. Вместо аргумента **source** можно использовать предопределенную константу **MPI_ANY_SOURCE**, в таком случае будет принято сообщение от процесса с любым номером. Также вместо аргумента **tag** можно использовать предопределенную константу **MPI_ANY_TAG**, в таком случае будет принято сообщение с любым идентификатором.

Структура **MPI_Status** хранит информацию о размере принятого сообщения, номере процесса–отправителя, идентификатор принятого сообщения и код ошибки. Узнать реальный размер принятого сообщения можно при помощи процедуры **int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)**, она принимает на вход статус сообщения **status**, тип принимаемых данных **datatype**, а возвращает размер принятого сообщения count в указанном типе данных. Для того, чтобы узнать номер процесса–отправителя, идентификатор сообщения и код ошибки, структура **MPI_Status** имеет поля **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR** соответственно. Ниже приведен пример кода, который показывает возможности данных функций:

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int rank, count;
    char buf[100];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        strcpy(buf, "Hello from 0");
        MPI_Send(buf, strlen(buf) + 1,
        MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(buf, 100, MPI_CHAR, MPI_ANY_SOURCE,
                           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_CHAR, &count);
        printf("Message ¥"%s¥", from %d, tag %d, size %d¥n",
                    buf,
                    status.MPI_SOURCE,
                    status.MPI_TAG,
                    count);
    }
    MPI_Finalize();
    return 0;
```

```
}
```

В данном примере процесс с номером 0 отсылает сообщение "Hello from 0" процессу с номером 1, при этом идентификатор сообщения выставлен в 99. Процессор с номером 1 принимает сообщение с максимальным размером в 100 от любого процесса с любым идентификатором. Реальный размер принятого сообщения узнается при помощи функции **MPI_Get_count**, а номер процесса–отправителя и идентификатор сообщения берутся из полей структуры **MPI_Status**. В случае успешной работы программа должна вывести следующую строчку:

```
Message ″Hello from 0″, from 0, tag 99, size 13
```

Блокирующий прием представлен всего лишь одной функцией, однако для отправки существуют несколько процедур, которые подходят для использования в различных случаях. Рассмотрим другие вариации блокирующей отправки подробнее.

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

Передача сообщения с синхронизацией. Выход из данной функции будет осуществлен только в случае, когда буфер отправитель **buf** будет безопасен для дальнейшего использования и процесс–получатель начал прием сообщения. Т. е. данная процедура дает гарантию, что процесс–получает достиг точки в коде, когда он начинает прием. Данная процедура может в каких-то случаях замедлить исполнение программы, но зато она позволяет снизить нагрузку на системный буфер MPI.

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

Блокирующая отсылка по готовности ("ready" отсылка). Данная процедура должна вызваться только в том случае, когда процесс–получатель уже инициировал прием сообщения. В противном случае результат работы данной функции может быть не определен и считается ошибочным. Для корректного использования перед данной функцией должна быть выполнена явная синхронизация процессов. В некоторых реализациях данная функция позволяет сократить накладные расходы при передачи сообщения за счет отсутствия проверки на прием и ускорить передачу данных.

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

Отсылка сообщения с буферизацией. Данная функция позволяет программисту положить данные в специальный буфер и произойдет немедленный возврат из функции. Процесс–получатель может еще не начинать прием сообщения. Для работы с данной функцией необходимо заранее выделить буфер, который она может использовать, это делается процедурой **MPI_Buffer_attach**, в случае если размер выделенного буфера меньше сообщения, функция вернет ошибку.

int MPI_Buffer_attach(void *buffer, int size)

Определить буфер **buf** размера size байт для использования в функциях отсылки с буферизацией. Буфер должен быть выделен заранее и не может использоваться для других целей. В каждом процессе может одновременно использоваться только один буфер, в случае если размер буфера недостаточен его необходимо отключить, используя процедуру **MPI_Buffer_detach**, изменить размер и подключить заново. Размер памяти, выделяемый для

передачи сообщения, должен превосходить размер сообщения на величину как минимум **MPI_BSEND_OVERHEAD**, в противном случае сообщение может не поместиться в буфер и функция пересылки вернет ошибку.

int MPI_Buffer_detach(void *bufferptr, int *size)

Отключение выделенного буфера для передачи сообщений. В переменные **bufferptr** будет записан адрес используемого буфера, а в переменную size размер буфера в байтах. Вызывающий процесс будет блокирован до тех пор, пока все сообщения, использующие буфер не будут отправлены. Данная функция возвращает указатель на буфер и его размер для того, чтобы отельные модули могли выключить буфер, затем использовать свой собственный, а в конце подключить старый буфер обратно. Ниже приведен код показывающий такую возможность.

```
int size, mysize, idummy;
void *ptr, *myptr, *dummy;
MPI_Buffer_detach(&ptr, &size);
MPI_Buffer_attach(myptr, mysize);


// код, использующий новый буфер


MPI_Buffer_detach(&dummy, &idummy);
MPI_Buffer_attach(ptr, size);
```

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

Данная процедура осуществляет совестный блокирующий прием и отсылку сообщения. Параметры данной функции полностью аналогичны параметром функций **MPI_Send** и **MPI_Recv**. Для принимаемого сообщения заполняется структура **status**. Принимающим и отсылающим процессом является один и тот же процесс, буферы приема и передачи не должны пересекаться. Данная функция гарантирует, что не будет возникать блокировки при пересылках. Сообщение отправленное данной функцией может быть принято другой операцией приема, также данная процедура может принять сообщения отправленное другой операцией отсылки.

6.2 Неблокирующие прием/передача

В MPI также предусмотрен набор функций для асинхронной передачи данных. Возврат из таких функций осуществляется сразу после их вызова, однако сам процесс взаимодействия может быть начат позже. Данные функции дают мощный инструмент для оптимизации приложений и избежания блокировок. Программист может не дожидаясь завершения отправки начинать делать какие-то вычисления, при этом одновременно будет осуществляться отправка сообщения. Однако асинхронные операции не всегда в полной мере поддерживаются реализацией библиотеки и аппаратурой. Поэтому в реальных приложениях выигрыш от асинхронных взаимодействий может быть минимальным либо вообще

отсутствовать.

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

Основная функция неблокирующей отсылки. Основные параметры аналогичны функции **MPI_Send**, отправляется сообщение **buf** размером count типов **datatype**. Отсылка идет к процессу с номером **dest** и сообщение имеет идентификатор **tag**, общение идет в рамках коммуникатора **comm**. Процедура заполняет специальную структуру типа **MPI_Reauest request**. Данный параметр является для идентификации конкретной неблокирующей процедуры. Буфер отсылки **buf** не безопасен для дальнейшего использования после работы функции, поскольку сообщение может быть еще не доставлено. Для проверки состояния сообщения существуют несколько процедур типа **MPI_Wait** и **MPI_Test.**

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

Неблокирующий прием сообщения. Синтаксис аналогичен функции блокирующего приема **MPI_Recv**, только теперь вместо статуса сообщения функция возвращает идентификатор неблокирующей процедуры **request**. В отличии от блокирующей процедуры возврат из функции происходит сразу, без ожидания заполнения буфера **buf** реальными данными. Дождаться приема данных или проверить дошли они или нет можно воспользоваться процедурами **MPI_Wait** и **MPI_Test.**

int MPI_Wait(MPI_Request *request, MPI_Status *status)

Дождаться выполнения неблокирующей процедуры **MPI_Isend** или **MPI_Irecv**. На вход принимает идентификатор неблокирующего взаимодействия **request**, возвращает статус сообщения **status**. Процесс будет заблокирован до тех пор, пока процедура асинхронного взаимодействия не будет завершена.

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

Проверка выполнения неблокирующего приема или отсылки. Параметр **flag** выставляется в 1 в случае если неблокирующая операция завершена и 0 в противном случае. Если процедура завершена, то также заполняется структура **status.**

Существуют вариации функций **MPI_Test** и **MPI_Wait** для ожидания нескольких процессов из списка, для ожидания только части процессов из списка и для ожидания любого процесса из списка. Соответственно они имеют названия **MPI_Testall**, **MPI_Testsome**, **MPI_Testany**, и **MPI_Waitall**, **MPI_Waitsome**, **MPI_Waitany.**

Для операция неблокирующей отсылки существуют аналоги операций блокирующей отсылки, они полностью аналогичны блокирующим, только для завершения этих операций необходимо вызывать **MPI_Wait**/**MPI_Test:**

- **MPI_Issend** — синхронная неблокирующая отсылка;
- **MPI_Ibsend** — неблокирующая отсылка с буферизацией;
- **MPI_Irsend** — неблокирующая отсылка по готовности.

7 Коллективные взаимодействия

В данном разделе рассмотрим процедуры *коллективного взаимодействия*. Основной

особенностью данных процедур является то, что в них участвуют все процессы коммуникатора. Если необходимо произвести коллективное взаимодействие только между частью процессов коммуникатора, необходимо их сначала выделить в отдельный коммуникатор и вести обмен в рамках нового коммуникатора. Соответствующая процедура должна быть вызвана на всех процессах коммуникатора и может отличатся только аргументами. Выход из коллективной функции осуществляется тогда, когда все буферы используемые в процедуре безопасны для дальнейшего использования. В этом понимании все коллективные процедуры MPI являются блокирующими, при их вызове возможна неявная синхронизация, однако она не гарантированна, что стоит учитывать при написании программ.

Результат коллективной операции может быть принят только такой же коллективной операцией. Операции типа точка–точка никак не мешают коллективным операциям и не взаимодействуют с ними, поэтому все для всех типом обмена можно использовать один коммуникатор.

Также в коллективных операциях отсутствуют идентификаторы сообщений (tag), поэтому стоит строго следить за порядком вызова их в коде.

Следует заметить, что все коллективные процедуры MPI можно заменить набором вызовов типа точка–точка. Однако поскольку большинство процедур довольно общие и используются во многих алгоритмах, то версии библиотек предлагают свою реализацию данного функционала, причем стоит использовать функции стандарта MPI, поскольку они реализованы используя, по возможности, оптимальные алгоритмы их реализации и в некоторых случаях могут использовать аппаратные возможности для ускорения соответствующих процедур.

int MPI_Barrier(MPI_Comm comm)

Процедура барьерной синхронизации. Блокирует вызывающий процесс до тех пор, пока все процессы коммуникатора **comm** не вызовут данную функцию. Процессы будут разблокированы только тогда, когда все процессы коммуникатора вызовут данную процедуру. При этом она может находится у различных процессов в различных частях кода, однако все процессы должны ее вызвать.

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

Коллективная рассылка сообщения от процесса с номером **root** всем остальным процессам коммуникатора **comm**. Сообщение состоит из count типов типа **datatype**. При этом значение из **buffer** у процесса с номером **root** будет скопировано в **buffer** у всех остальных процессов, здесь один и тот же аргумент у одних процессов может использоваться на чтение, у других — на запись. Аргументы **count**, **datatype**, **root** и **comm** должны быть одинаковые у всех процессов.

int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

Коллективная рассылка данных от одного процесса с номером **root** ко всем процессам в коммуникаторе, при этом каждый процесс получает свою часть данных, включая процесс с номером **root**. Рассылка осуществляется из массива **sendbuf** блоками размером **sendcnt** типов **sendtype**, данные аргументы используются только на процессе с номером **root**, другие процессы их игнорируют. Прием осуществляется в массив **recvbuf** размером **recvcnt** и типом

recvtype, весь обмен ведется в рамках коммуникатора **comm**. Действие данной функции можно рассматривать как деление массива **sendbuf** на **N** равных частей размером **sendcnt**, где **N** — размер коммуникатора **comm**. Затем эти части в порядке возрастания номеров процессов коммуникатора распределяются между всеми процессами, т. е. *i*-я часть массива **sendbuf** достанется процессу с номером *(i-1)*.

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

Сбор информации от всех процессов коммуникатора одному процессу с номером **root**. Осуществляется сбор массивов данных **sendbuf** размером **sendcnt** и типом **sendtype** в массив на процессе с номером **root** в буфер **recvbuf**, прием осуществляется блоками размером **recvcount** и имеющими тип **recvtype**. Весь обмен ведется в рамках коммуникатора **comm**. Аргументы **recvbuf**, **recvcount** и **recvtype** имеют смысл только на процессоре **root**, на остальных процессах они игнорируются. Данная операция противоположна операции **MPI_Scatter**, также прием данных осуществляется от всех процессов, включая процесс с номером **root**.

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

Операция коллективного сбора данных с одновременной их обработкой. Выполняется **count** независимых глобальных действий оператора **op** над соответствующими элементами массива **sendbuf**, имеющего тип **datatype**. Результат выполнения оператора записывается в массив **recvbuf** у процессора **root**, причем номера элементов в старом и новом массивах совпадают. В MPI существует ряд предопределенных глобальных операций, также программист может самостоятельно создавать новые, используя процедуру **MPI_Op_create**. Ниже приведен список встроенных процедур MPI:

- **MPI_MAX** — глобальный максимум;
- **MPI_MIN** — глобальный минимум;
- **MPI_SUM** — глобальная сумма значений у всех процессов;
- **MPI_PROD** — глобальное произведение;
- **MPI_LAND** — логическое "И";
- **MPI_BAND** — логическое побитовое "И";
- **MPI_LOR** — логическое "ИЛИ";
- **MPI_BOR** — логическое побитовое "ИЛИ";
- **MPI_LXOR** — логическое исключающее "ИЛИ";
- **MPI_BXOR** — логическое побитовое исключающее "ИЛИ";
- **MPI_MAXLOC** — глобальный максимум с номером соответствующего процесса;
- **MPI_MINLOC** — глобальный минимум с номером соответствующего процесса.

8 Заключение

В данной работе описан основной функционал стандарта MPI и основные принципы, лежащие в его основе. Несмотря на то, что дан только краткий обзор процедур MPI, уже этого достаточно для распараллеливания большинства программ MPI. Данный стандарт дает мощный функционал для распараллеливания программ в системах с распределенной памятью, являясь стандартом де-факто он позволяет легко переносить вычислительный код с одной платформы на другую. MPI поддерживается большинством современных производителей высокопроизводительных вычислительных систем и разработчиками программного обеспечения, в тоже время он доступен широкому кругу пользователей в виде бесплатных открыты реализаций. Все это вкупе с его простотой и прозрачной структурой позволяет разрабатывать высокопроизводительный код, работающий на большинстве современных аппаратных платформах.

Список литературы.

1. http://www.mpi-forum.org/
2. http://www.open-mpi.org/
3. http://www.mcs.anl.gov/mpi/mpich
4. http://mvapich.cse.ohio-state.edu/
5. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. Спб.: БХВ-Петербург, 2002.