ЛАБОРАТОРНАЯ РАБОТА №2

Основы использования MPICH, программной реализации интерфейса MPI. Двухточечный блокирующий обмен сообщениями.

1 ЦЕЛЬ РАБОТЫ

1.1 Изучить принципы организации двухточечного обмена сообщениями MPI.

1.2 Освоить использование типовых функций MPICH для блокирующего обмена сообщениями.

1.3 Получить практический опыт написания MPICH-приложений.

2 КЛЮЧЕВЫЕ ПОЛОЖЕНИЯ

В данной лабораторной работе мы начнем осваивать один из базовых для MPI способов обмена сообщениями — двухточечный обмен. Рассмотрим особенности различных режимов обмена, разберем правила использования основных функций передачи и приема сообщений. Кроме двухточечного существует еще коллективный обмен сообщениями, но о нем мы будем говорить в более поздних работах.

Двухточечный обмен, с точки зрения программиста, выполняется следующим образом: для пересылки сообщения процесс-источник вызывает функцию передачи, при обращении к которой указывается ранг процесса-получателя в соответствующей области взаимодействия. Последняя задается своим коммуникатором, обычно это MPI_COMM_WORLD. Процесс-получатель, для того чтобы получить направленное ему сообщение, должен вызвать функцию приема, указав при этом ранг источника.

Необходимо учитывать, что во всех реализациях MPI, в том числе и в MPICH, гарантируется выполнение некоторых свойств двухточечного обмена. Одним из них является сохранение порядка сообщений, которые при двухточечном обмене не могут "обгонять" друг друга. Кроме того, в двухточечном обмене следует соблюдать правило соответствия типов передаваемых и принимаемых данных.

Теоретические положения MPI определяют четыре разновидности двухточечного обмена: синхронный, асинхронный, блокирующий и неблокирующий. В MPI имеются несколько режимов обмена, различающиеся условиями инициализации и завершения передачи сообщения:

- *стандартная* передача считается выполненной и завершается, как только сообщение отправлено, независимо от того, дошло оно до адресата или нет. В стандартном режиме передача сообщения может начинаться, даже если еще не начат его прием;

- *синхронная* передача отличается от стандартной тем, что она не завершается до тех пор, пока не будет завершен прием сообщения. Адресат, получив сообщение, посылает процессу отправившему его уведомление, которое должно быть получено отправителем для того, чтобы обмен считался выполненным. Операцию передачи уведомления иногда называют «рукопожатием»;

- *буферизованная* передача завершается сразу же, сообщение копируется в системный буфер, где и ожидает своей очереди на пересылку. Завершается буферизованная передача независимо от того, выполнен прием сообщения или нет;

- передача «*по готовности*» начинается только в том случае, когда адресат инициализировал прием сообщения, а завершается сразу, независимо от того, принято сообщение или нет.

Каждый из этих четырех режимов имеется как в блокирующей, так и в неблокирующей формах. В MPI приняты следующие соглашения об именах функций двухточечного обмена: `MPI_[I][R, S, B]Send`, где префикс [I] (Immediate) обозначает неблокирующий режим. Один

из префиксов [R, S, B] обозначает режим обмена, соответственно: по готовности, синхронный и буферизованный. Отсутствие префикса обозначает функцию стандартного обмена.

Для функций приема: `MPI_[I]Recv`, т. е. всего две разновидности приема. Функция MPI_Irsend( ), например, выполняет передачу «по готовности» в неблокирующем режиме, MPI_Bsend( ) - буферизованную передачу с блокировкой, а MPI_Recv( ) выполняет блокирующий прием сообщений. Заметим, что функция приема любого типа может принять сообщения от любой функции передачи. В текущей работе мы разберем блокирующий вариант каждого из вышеперечисленных режимов обмена сообщениями.

Блокирующие операции приостанавливают выполнение вызывающего процесса, заставляя процесс ожидать завершения передачи данных. Блокировка гарантирует выполнение действий в заданном порядке, но с другой стороны создает условия для возникновения тупиковых ситуаций, когда оба процесса-участника обмена блокируются одновременно.

Начнем рассмотрение режимов обмена сообщениями со стандартной передачи. Сообщение, отправленное в стандартном режиме, может в течение некоторого времени «перемещаться» по коммуникационной сети параллельного компьютера, увеличивая тем самым ее загрузку. В MPI-программах рекомендуется соблюдать следующие правила:

- блокирующие операции обмена следует использовать осторожно, поскольку в этом случае возрастает вероятность тупиковых ситуаций;

- если для правильной работы процесса имеет значение последовательность приема сообщений, источник должен передавать новое сообщение, только убедившись, что предыдущее уже принято. В противном случае может нарушиться детерминизм выполнения программы;

- сообщения должны гарантированно и с достаточно большой частотой приниматься процессами, которым они посылаются, иначе коммуникационная сеть может переполниться, что приведет к резкому падению скорости ее работы. Это, в свою очередь, снизит производительность всей системы.

Библиотечной функцией стандартной блокирующей передачи является MPI_Send( ). Она блокирует выполнение процесса до завершения передачи сообщения (что, однако, не гарантирует завершения его приема):

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

где buf — адрес первого элемента в буфере передачи; count — количество элементов в буфере передачи; datatype — тип MPI каждого пересылаемого элемента; dest — ранг процесса-получателя сообщения. Ранг здесь — целое число от 0 до n — 1, где n — число процессов в области взаимодействия; tag — тег сообщения; comm — коммуникатор.

Стандартная блокирующая передача начинается независимо от того, был ли зарегистрирован соответствующий прием, а завершается только после того, как сообщение принято системой и процесс-источник может вновь использовать буфер передачи. Сообщение может быть скопировано прямо в буфер приема, а может быть помещено во временный системный буфер, где и будет дожидаться вызова адресатом функции приема. В этом случае говорят о буферизации сообщения. Она требует выполнения дополнительных операций копирования из памяти в память, а также выделения памяти для промежуточного буфера. В стандартном режиме MPI самостоятельно решает, надо ли буферизовать исходящие сообщения. Передача может завершиться еще до вызова соответствующей операции приема. С другой стороны, буфер может быть недоступен или MPI может решить не буферизовать исходящие сообщения по соображениям сохранения высокой производительности. В этом случае передача завершится только после того, как будет зарегистрирован соответствующий прием и данные будут переданы адресату.

Стандартный блокирующий прием выполняется функцией:

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

где входные параметры: count — максимальное количество элементов в буфере приема; datatype — тип принимаемых данных; source — ранг источника. Можно использовать специальное значение MPI_ANY_SOURCE, соответствующее произвольному значению ранга. В программировании идентификатор, отвечающий произвольному значению параметра, часто называют "джокером"; tag — тег сообщения или "джокер" MPI_ANY_TAG, соответствующий произвольному значению тега; comm — коммуникатор. При указании коммуникатора "джокеры" использовать нельзя. Выходными параметрами являются: buf — начальный адрес буфера приема. Его размер должен быть достаточным, чтобы разместить принимаемое сообщение, иначе при выполнении приема произойдет сбой — возникнет ошибка переполнения; status — статус обмена.

Если сообщение меньше, чем буфер приема, изменяется содержимое лишь тех ячеек памяти буфера, которые относятся к сообщению. Информация о длине принятого сообщения содержится в одном из полей статуса, но к этой информации у программиста нет прямого доступа (как к полю структуры или элементу массива). Размер полученного сообщения (count) можно определить с помощью вызова функции MPI_Get_count( ):

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

где count содержит количество элементов данных. Аргумент datatype должен соответствовать типу данных, указанному в операции обмена.

Функция MPI_Recv( ) может принимать сообщения, отправленные в любом режиме. Имеется некоторая асимметрия между операциями приема и передачи. Она состоит в том, что прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес. Приемник может использовать "джокеры" для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что в этом случае блокирующие операции могут привести к «тупику».

Пример блокирующего стандартного двухточечного обмена приведен в листинге 1. Приведенный код демонстрирует обмен данными между процессами с четными и нечетными рангами. Предполагается, что значение size четно.

Листинг 1 — Использование стандартного блокирующего обмена

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int myrank, size, message;
int TAG = 0;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
message = myrank;
if((myrank % 2) == 0) {
if((myrank + 1) != size)
MPI_Send(&message, 1, MPI_INT, myrank + 1, TAG, MPI_COMM_WORLD);
}
else {
if(myrank != 0)
MPI_Recv(&message, 1, MPI_INT, myrank—1, TAG, MPI_COMM_WORLD,\
&status);
printf("received :%i\n", message);
}
MPI_Finalize();
return 0;
}
```

Результат выполнения этой программы в случае запуска 10 процессов выглядит так:

```
#mpiexec –f machinefile –n 10 example_prog
received :0 received :2 received :6 received :4 received :8
```

Порядок вывода сообщений может быть другим, он меняется от запуска к запуску.

Если стандартная передача не может быть выполнена из-за недостаточного объема приемного буфера, осуществление процесса блокируется до тех пор, пока не будет доступен буфер достаточного размера. Иногда это может оказаться удобным. Например, пусть источник циклически посылает новые значения адресату и пусть они вырабатываются быстрее, чем потребитель может их принять. При использовании буферизованной передачи может произойти переполнение буфера приема. Для того чтобы его избежать, в программу придется включить дополнительную синхронизацию, а при использовании стандартной передачи такая синхронизация выполняется автоматически.

Следующим в списке режимов обмена сообщениями является синхронный блокирующий обмен. При синхронном обмене адресат посылает источнику «квитанцию» — уведомление о завершении приема. Только после получения этого уведомления обмен считается завершенным, и источник «знает», что его сообщение получено. Синхронная передача выполняется с помощью функции MPI_Ssend( ):

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

Ее параметры совпадают с параметрами функции MPI_Send( ).

Поговорим про особенности блокирующего буферизованного обмена. Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме так же, как в стандартном режиме. Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема. Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.

Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи. В отличие от стандартного обмена, в этом случае работа источника и адресата не синхронизирована. Вызов функции буферизованного обмена MPI_Bsend( ):

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

Ее параметры совпадают с параметрами функции MPI_Send( ).

При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера. Это делается с помощью вызова функции MPI_Buffer_attach( ):

```
int MPI_Buffer_attach(void *buffer, int size)
```

В результате вызова создается буфер buffer размером size байтов.

После завершения работы с буфером его необходимо отключить. Делается это с помощью вызова функции MPI_Buffer_detach( ):

```
int MPI_Buffer_detach(void *buffer_addr, int *size)
```

В результате выполнения вызова возвращается адрес (buffer) и размер (size) отключаемого буфера. Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны. Благодаря этому, вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал буфер, однако следует помнить, что в языке С данный вызов не освобождает автоматически память, отведенную для буфера.

Приведем простой пример использования функций буферизованного обмена, а также функций подключения и отключения буфера (см. листинг 2).

Листинг 2 — Пример использования блокирующего буферизованного обмена

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int *buffer;
int myrank;
MPI_Status status;
int buffsize = 1;
int TAG = 0;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
buffer = (int *) malloc(buffsize + MPI_BSEND_OVERHEAD);
MPI_Buffer_attach(buffer, buffsize + MPI_BSEND_OVERHEAD);
buffer = (int *) 10;
MPI_Bsend(&buffer, buffsize, MPI_INT, 1, TAG, MPI_COMM_WORLD);
MPI_Buffer_detach(&buffer, &buffsize);
}
else
{
MPI_Recv(&buffer,buffsize, MPI_INT, 0, TAG, MPI_COMM_WORLD, &status);
printf("received: %i\n", buffer);
}
MPI_Finalize() ;
return 0;
}
```

Обратите внимание на то, что размер буфера должен превосходить размер сообщения на величину MPI_BSEND_OVERHEAD. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.

Если перед выполнением операции буферизованного обмена не выделен буфер, MPI ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.

Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

Еще одним режимом является обмен «по готовности». Передача «по готовности» должна начинаться, если уже зарегистрирован соответствующий прием. Завершается она сразу же. Если прием не зарегистрирован, результат выполнения операции не определен. Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь. Сообщение просто отправляется в коммуникационную сеть в предположении, что адресат его получит. Следует иметь в виду, что эта предположение может и не оправдаться. Правила здесь такие же, как и в операциях стандартного или синхронного обмена. В корректной программе передача «по готовности» может быть заменена стандартной передачей, результат выполнения и поведение программы при этом не должны измениться. Измениться может только скорость ее выполнения. Передача «по готовности» выполняется с помощью функции MPI_Rsend( ):

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

Параметры у нее те же, что и у функции MPI_Send( ).

Обмен «по готовности» может увеличить производительность программы, поскольку здесь

не используются этапы установки межпроцессных связей, а также буферизация. Все это — операции, требующие времени. С другой стороны, обмен «по готовности» потенциально опасен, кроме того, он усложняет отладку, поэтому его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой, а выигрыша в быстродействии надо добиться любой ценой.

Иногда необходимо получить информацию о сообщении еще до его помещения в буфер приема. Это возможно реализовать с помощью функций-пробников MPI_Probe() и MPI_Iprobe( ). На основании полученной информации принимается решение о дальнейших действиях. Можно, например, выделить буфер приема достаточного размера, определив длину сообщения с помощью функции MPI_Get_count( ), которая дает доступ к одному из полей статуса. Если после вызова MPI_Probe( ) следует вызов MPI_Recv( ) с такими же значениями аргументов, что и MPI_Probe( ), он поместит в буфер приема то же самое сообщение, информация о котором была получена MPI_Probe( ).

Функция MPI_Probe( ) выполняет блокирующую проверку доставки сообщения:

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

где source — ранг источника или "джокер"; tag — значение тега или "джокер"; comm — коммуникатор. Выходным параметром является статус (status), который и содержит необходимую информацию.

Неблокирующая проверка сообщения выполняется функцией MPI_Iprobe( ):

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm, \
int *flag, MPI_Status *status)
```

Входные параметры те же, что и у подпрограммы MPI_Probe( ), а выходные: flag — флаг и status — статус. Если сообщение уже поступило и может быть принято, возвращается значение флага "истина". Проверка приема может выполняться с помощью MPI_Iprobe( ) неоднократно. При этом не обязательно выполнять прием сообщения сразу же после его поступления.

Рассмотрим пример использования функций-пробников (см. листинг 3). Программа ориентирована на запуск с двумя процессами, которые получат номера 0 и 1. Процесс с рангом 0 посылает сообщение процессу с рангом 1. Сообщение содержит в себе одно целое (200), но процесс с рангом 1 об этом не знает. Поэтому требуется использование функций MPI_Probe( ) и MPI_Get_count( ) соответственно для определения факта поступления сообщения, а также количества элементов данных в сообщении. После этого производится выделение памяти через malloc( ), и собственно прием сообщения через MPI_Recv( ).

Листинг 3 — Пример использования функций-пробников

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
int myid, numprocs;
MPI_Status status;
int mytag, ierr, icount, j, *i;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
printf(" Hello from c process: %d  Numprocs is %d\n",myid, numprocs);

mytag = 123;
if(myid == 0) {
j = 200;
icount = 1;
```

```
ierr = MPI_Send(&j, icount, MPI_INT, 1, mytag, MPI_COMM_WORLD);
}
if(myid == 1){
ierr = MPI_Probe(0, mytag, MPI_COMM_WORLD, &status);
ierr = MPI_Get_count(&status, MPI_INT, &icount);
i = (int*) malloc(icount * sizeof(int));
printf("getting %d\n", icount);
ierr = MPI_Recv(i, icount, MPI_INT, 0, mytag,MPI_COMM_WORLD,&status);
printf("i= ");
for(j=0; j<icount; j++)
printf("%d ",i[j]);
 printf("\n");
}
MPI_Finalize();
}
```

3 КОНТРОЛЬНЫЕ ВОПРОСЫ

3.1 Поясните принципы организации двухточечного обмена сообщениями.
3.2 Опишите имеющиеся в MPI режимы обмена сообщениями, различая их по условиям начала и завершения передачи.
3.3 Поясните на конкретных примерах варианты именования функций двухточечного обмена MPICH, в зависимости от режима обмена, а также способа реализации — блокирующего или неблокирующего.
3.4 Поясните сущность блокирующих операций обмена MPI-сообщениями.
3.5 Опишите способ работы и особенности использования MPI-функции блокирующей стандартной передачи.
3.6 Опишите способ работы и особенности использования MPI-функции блокирующего стандартного приема.
3.7 Опишите способ работы и особенности использования MPI-функции блокирующей синхронной передачи.
3.8 Опишите способ работы и особенности использования MPI-функции блокирующей буферизованной передачи.
3.9 Опишите способ работы и особенности использования MPI-функции блокирующей передачи «по готовности».
3.10 Поясните назначение и особенности использования функций-пробников из состава библиотеки MPI.

4 ДОМАШНЕЕ ЗАДАНИЕ

4.1 Изучить ключевые положения.
4.2 Письменно ответить на контрольные вопросы.
4.3 Подготовиться к выполнению лабораторного задания.

5 ЛАБОРАТОРНОЕ ЗАВДАНИЕ

5.1 Напишите распределенную программу на C, используя функции блокирующего двухточечного обмена библиотеки MPICH, описываемые в ключевых положениях. Предполагается, что в результате запуска такого приложения будет создано четное количество процессов (ключ -n). Процессы с четными и нечетными рангами обмениваются друг с другом сообщениями. Каждая

пара процессов поочередно передает друг другу некоторое сообщение произвольной структуры, количество обменов равно 15. При этом после каждого действия на терминал выводится поясняющее сообщение с указанием передающего и принимающего процессов, номера операции, а также, возможно, содержимое сообщения. Для реализации передачи сообщений использовать функцию блокирующего буферизованного обмена.
5.2 Перепишите программу предыдущего пункта задания, использовав функцию блокирующего синхронного обмена вместо функции блокирующей буферизованной передачи. Кроме того, смоделируйте ситуацию, когда в принимающем процессе заранее не известно количество элементов данных, которые были ему переданы. Чтобы определить размер приемного буфера и далее выделить под него память, воспользуйтесь функциями-пробниками, предоставляемыми библиотекой MPICH.

6. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ ПРОТОКОЛА

6.1 Название лабораторной работы.
6.2 Цель работы.
6.3 Результаты выполнения домашнего задания.
6.4 Краткое описание проделанной работы.
6.5 Выводы о проделанной работе.
6.6 Дата, подпись студента, виза преподавателя.

Составлено по материалам:
1. *Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем. - СПб.:БХВ-Петербург, 2002.*
2. *Мирошников А.С., Гречаный С.В. Методические указания к лабораторным работам по дисциплине «Параллельная обработка данных». - Владикавказ, 2013.*
3. *http://geco.mines.edu/, MPI Workshop.*

Приложение А Обозначения типов данных MPI, которые соответствуют типам данных в языке программирования С. В MPI должны соблюдаться правила совместимости типов. Соответствие типов должно, как правило, иметь место в процедурах отправки и процедурах приема сообщений. Из базовых типов могут быть сконструированы более сложные типы данных.
Таблица 1 — Типы данных MPI для языка С

Тип данных MPI	**Тип данных С**	**Тип данных MPI**	**Тип данных С**
MPI_CHAR	signed char	MPI_LONG_DOUBLE	long double
MPI_SHORT	signed short int	MPI_BYTE	Нет соответствия
MPI_INT	signed int	MPI_PACKED	Нет соответствия
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_FLOAT	float		
MPI_DOUBLE	double		