Лабораторная работа 3

Знакомство с процедурами буферизованного и неблокирующего двухточечного обмена MPI

Цель работы:

Приобрести навыки составления MPI-программ. Изучить работу процедур буферизованного и неблокирующего двухточечного обмена MPI.

Задачи работы:

1) Изучить теоретические основы работы процедур буферизованного и неблокирующего двухточечного обмена MPI.
2) Выполнить задания лабораторной работы.
3) Ответить на вопросы.

Теория

Двухточечный буферизованный обмен

При передаче сообщения в буферизованном режиме источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме. Выделение буфера и его размер контролируются программистом, который должен заранее создать буфер достаточного размера. Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи.

Создание буфера:

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

Выходной параметр:

- `buf` - буфер размером size байтов.

После завершения работы с буфером его необходимо отключить. После отключения буфера можно вновь использовать занимаемую им память, однако следует помнить, что в языке С данный вызов не освобождает автоматически память, отведенную для буфера.

Отключение буфера:

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

Выходные параметры:

- `buf` - адрес;
- `size` - размер отключаемого буфера.

Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти.

Двухточечные неблокирующие обмены

Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Передача данных из буфера или их считывание происходит одновременно с выполнением других операций. Завершается обмен вызовом дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи. До завершения обмена запись в буфер или считывание из него производить нельзя, так как сообщение может быть еще не отправлено или не получено. Неблокирующая передача может быть принята подпрограммой блокирующего приема и наоборот.

Неблокирующий обмен выполняется в два этапа:

1) Инициализация обмена.
2) Проверка завершения обмена.

Стандартная блокирующая передача:

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

Входные параметры:

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

Стандартный блокирующий прием:

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

Входные параметры:

- `count` - максимальное количество элементов в буфере приема (фактическое их количество можно определить с помощью подпрограммы `MPI_Get_count`);
- `datatype` - тип принимаемых данных. Напомним о необходимости соблюдения соответствия типов аргументов подпрограмм приема и передачи;
- `source` - ранг источника. Можно использовать специальное значение `MPI_ANY_SOURCE`, соответствующее произвольному значению ранга. В программировании идентификатор, отвечающий произвольному значению параметра, часто называют "джокером". Этот термин будем использовать и мы;
- `tag` - тег сообщения или "джокер" `MPI_ANY_TAG`, соответствующий произвольному значению тега;
- `comm` - коммуникатор. При указании коммуникатора "джокеры" использовать нельзя.

Выходные параметры:

- `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.

Синхронная передача:

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

Параметры совпадают с параметрами функции `MPI_Send`.

Буферизованный обмен:

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

Параметры совпадают с параметрами функции `MPI_Send`.

Передача по готовности:

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

Параметры совпадают с параметрами подпрограммы `MPI_Send`.

Определение размера полученного сообщения:

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

Аргумент `datatype` должен соответствовать типу данных, указанному в операции передачи сообщения.

Подпрограммы-пробники

Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова подпрограмм ожидания, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение "истина", если операция выполнена.

Получить информацию о сообщении до его помещения в буфер приема можно с помощью подпрограмм-пробников **`MPI_Probe`** и **`MPI_IProbe`**. На основании полученной информации принимается решение о дальнейших действиях. С помощью вызова подпрограммы **`MPI_Probe`** фиксируется поступление (но не прием) сообщения. Затем определяется источник сообщения, его длина, выделяется буфер подходящего размера и выполняется прием сообщения.

Блокирующая проверка доставки сообщения:

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

Входные параметры:

- `source` - ранг источника или "джокер";
- `tag` - значение тега или "джокер";
- `comm` - коммуникатор.

Выходной параметр:

- `status` - статус.

Неблокирующая проверка сообщения:

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

Входные параметры этой функции те же, что и у функции `MPI_Probe`.

Выходные параметры:

- `flag` - флаг;
- `status` - статус.

Если сообщение уже поступило и может быть принято, возвращается значение флага "истина".

Подпрограмма **`MPI_Wait`** блокирует работу процесса до завершения приема или передачи сообщения. Функции **`MPI_Wait`** и **`MPI_Test`** можно использовать для завершения операций приема и передачи.

Блокировка работы процесса до завершения приема или передачи сообщения:

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

Входной параметр:

- `request` - идентификатор операции обмена.

Выходной параметр:

- `status` - статус выполненной операции.

Значение статуса для операции передачи сообщения можно получить вызовом подпрограммы `MPI_Test_cancelled`. Можно вызвать `MPI_Wait` с пустым или неактивным аргументом `request`. В этом случае операция завершается сразу же с пустым статусом.

Успешное выполнение подпрограммы `MPI_Wait` после вызова `MPI_Ibsend` подразумевает, что буфер передачи можно использовать вновь, то есть пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы `MPI_Buffer_attach`. В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму `MPI_Cancel`, которая освобождает память, выделенную подсистеме коммуникаций.

Неблокирующая проверка завершения приема или передачи сообщения:

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

Входной параметр:

- `request` - идентификатор операции обмена.

Выходные параметры:

- `flag` - "истина", если операция, заданная идентификатором `request`, выполнена;
- `status` - статус выполненной операции.

Если при вызове `MPI_Test` используется пустой или неактивный аргумент `request`, операция возвращает значение флага "истина" и пустой статус.

Проверка завершения всех обменов:

```
int MPI_Waitall(int count, MPI_Request requests[], MPI_Status
            statuses[])
```

Выполнение процесса блокируется до тех пор, пока все операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` – количество запросов на обмен (размер массивов `requests` и `statuses`).

В результате выполнения подпрограммы `MPI_Waitall` запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение `MPI_REQUEST_NULL`. Список может содержать пустые или неактивные запросы. Для каждого из них устанавливается пустое значение статуса.

В случае неуспешного выполнения одной или более операций обмена подпрограмма `MPI_Waitall` возвращает код ошибки `MPI_ERR_IN_STATUS` и присваивает полю ошибки статуса значение кода ошибки соответствующей операции. Если операция выполнена успешно, полю присваивается значение `MPI_SUCCESS`, а если не выполнена, но и не было ошибки – значение `MPI_ERR_PENDING`. Последний случай соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Неблокирующая проверка завершения обменов:

```
int MPI_Testall(int count, MPI_Request requests[], int *flag,
            MPI_Status statuses[])
```

При вызове возвращается значение флага (`flag`) "истина", если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение "ложь", а массив `statuses` не определен. `count` – количество запросов.

Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена. Если запрос был сформирован операцией неблокирующего обмена, он аннулируется, а соответствующему элементу массива присваивается значение `MPI_REQUEST_NULL`. Каждому статусу, соответствующему пустому или неактивному запросу, присваивается пустое значение.

Блокирующая проверка завершения любого числа обменов:

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

Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (requests) не будет завершен.

Входные параметры:

- requests - запрос;
- count - количество элементов в массиве requests, а выходные: status и index.

Выходные параметры:

- index - индекс запроса (в языке С это целое число от 0 до count-1) в массиве requests;
- status - статус.

Если запрос на выполнение операции был сформирован неблокирующей операцией обмена, он аннулируется и ему присваивается значение MPI_REQUEST_NULL. Массив запросов может содержать пустые или неактивные запросы. Если в списке вообще нет активных запросов или он пуст, вызовы завершаются сразу со значением индекса MPI_UNDEFINED и пустым статусом.

Проверка выполнения любого ранее инициализированного обмена:

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

Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы MPI_Waitany. Дополнительный аргумент flag, который принимает значение "истина", если одна из операций завершена. Блокирующая подпрограмма MPI_Waitany и неблокирующая MPI_Testany взаимозаменяемы, впрочем, как и другие аналогичные пары.

Подпрограммы MPI_Waitsome и MPI_Testsome действуют аналогично подпрограммам MPI_Waitany и MPI_Testany, кроме случая, когда завершается более одного обмена. В подпрограммах MPI_Waitany и MPI_Testany обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для MPI_Waitsome и MPI_Testsome статус возвращается для всех завершенных обменов. Эти подпрограммы можно использовать для определения, сколько обменов завершено:

```
int MPI_Waitsome(int incount, MPI_Request requests[], int
            *outcount, int indices[], MPI_Status statuses[])
```

Здесь incount – количество запросов. В outcount возвращается количество выполненных запросов из массива requests, а в первых outcount элементах массива indices возвращаются индексы этих операций. В первых outcount элементах массива statuses возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру outcount присваивается значение MPI_UNDEFINED.

Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request requests[], int
            *outcount, int indices[], MPI_Status statuses[])
```

Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Трансляция и выполнение MPI-программ

Для трансляции и компоновки программ на языке C++ используется команда **`mpiCC`**, для трансляции программ на языке C – команда **`mpicc`**. Пример применения команды:

```
mpicc lab.c –o lab
```

Для выполнения MPI-программ используется загрузчик приложений **`mpirun`**. Он запускает указанное количество копий программы. Команда запуска:

```
mpirun –np X [ключи MPI] программа [ключи и аргументы программы]
```

где **`X`** – число запускаемых процессов.

Например, после развёртывания кластера PelicanHPC, строка запуска программы будет выглядеть так:

```
mpirun –np 10 --hostfile /home/user/tmp/bhosts Lab2
```

т.е. будет запущено `10` процессов программы `Lab3`, IP-адреса вычислительных узлов кластера будут получены из файла `bhosts`.

Лабораторная работа

В заданиях лабораторной работы предлагается дописать пропущенные фрагменты программ на языке C (программы написаны с использованием процедур MPICH 1.2.7). Пропущенные фрагменты обозначены многоточием.

Задание 1

В исходном тексте программы на языке C пропущены вызовы процедур буферизованного обмена. Добавить эти вызовы, откомпилировать и запустить программу.

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

int main(int argc, char *argv[])
{
  int *buffer;
  int myrank;
  MPI_Status status;
  int bufsize = 1;
  int tag = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
  buffer = (int*)malloc(bufsize + MPI_BSEND_OVERHEAD);

  if (myrank == 0)
  {
    MPI_Buffer_....

    *buffer = 10;

    MPI_....
  }
  else
  {
    MPI_Recv(buffer, sizeof(buffer), MPI_INT, 0, tag, MPI_COMM_WORLD,
&status);
    printf("%i received: %i\n", myrank, *buffer);
  }
  MPI_Finalize();
  return 0;
}
```

Задание 2

В исходном тексте программы на языке С пропущены вызовы подпрограмм-пробников. Добавить эти вызовы, откомпилировать и запустить программу несколько раз.

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

int main(int argc,char *argv[])
{
  int *buf, source, i;
  int message[3] = {0, 1, 2};
  int myrank, data = 2014, count, tag = 0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  if(myrank == 0)
  {
    MPI_Send(&data, 1, MPI_INT, 2, tag, MPI_COMM_WORLD);
  }
  else
    if (myrank == 1)
    {
      MPI_Send(&message, 3, MPI_INT, 2, tag, MPI_COMM_WORLD);
    }
    else
    {
      MPI_....

      source = status.MPI_SOURCE;

      MPI_Get_count(....);
```

```
      if (NULL == (buf = (int*)malloc(count)))
      {
  printf("malloc failed!\n");
  return (-1);
      }

      MPI_Recv(&buf[0], count, MPI_INT, MPI_ANY_SOURCE, tag,
MPI_COMM_WORLD, &status);
      for (i = 0; i < count; i++)
      {
  printf("%i receive from %i: %i\n", myrank, source, buf[i]);
      }
    }
    MPI_Finalize();
    return 0;
}
```

Задания по вариантам

1) Создайте и выполните на разном числе процессоров программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую следующий алгоритм:
   - на нулевом процессоре инициализируется переменная (`int A`);
   - нулевой процессор рассылает переменную `A` всем процессорам, кроме самого себя;
   - после получения переменной `A`, все процессоры прибавляют к ней свой индивидуальный номер и передают на нулевой процессор;
   - нулевой процессор получает от всех процессоров сообщения и выводит их на экран.
2) Создайте и выполните на разном числе процессоров программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую следующий алгоритм:
   - на нулевом процессоре инициализируется переменная (`int B`);
   - нулевой процессор рассылает переменную `B` всем чётным процессорам;
   - после получения переменной `B`, чётные процессоры прибавляют к ней свой индивидуальный номер и передают на нулевой процессор;
   - нулевой процессор получает ответные сообщения и выводит их на экран.
3) Создайте и выполните на разном числе процессоров программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую следующий алгоритм:
   - на нулевом процессоре инициализируется переменная (`int C`);
   - нулевой процессор рассылает переменную `C` всем нечётным процессорам;
   - после получения переменной `C`, нечётные процессоры прибавляют к ней свой индивидуальный номер и передают на нулевой процессор;
   - нулевой процессор получает ответные сообщения и выводит их на экран.

4) Создайте и выполните на разном числе процессоров программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую следующий алгоритм:
   - на нулевом процессоре инициализируется переменная (`int D`);
   - нулевой процессор рассылает переменную `D` всем процессорам;
   - после получения переменной `D`, все процессоры прибавляют к ней свой индивидуальный номер, и если он является простым числом, то передают на нулевой процессор;
   - нулевой процессор получает ответные сообщения и выводит их на экран.
5) Напишите программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи массива псевдослучайных целых чисел от одного процессора другому. Определите максимально допустимую длину передаваемого сообщения (максимальный размер массива).
6) Напишите программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи массива псевдослучайных чисел с плавающей точкой от одного процессора другому. Определите максимально допустимую длину передаваемого сообщения (максимальный размер массива).
7) Напишите программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи данных по кольцу: очередной процессор дожидается сообщения от предыдущего и потом посылает следующему процессору.
8) Создайте программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи данных по кольцу: все процессора одновременно посылают и принимают сообщения.
9) Напишите программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи данных по двум кольцам: нечетные процессора образуют первое кольцо, четные – второе.
10) Создайте программу, используя функции неблокирующего (и буферизованного) двухточечного обмена, реализующую алгоритм передачи данных от каждого процессора каждому.
11) * Создайте и выполните программу, используя коммуникационные функции (`MPI_Ssend`, `MPI_Bsend`, `MPI_Rsend`, `MPI_Isend`, `MPI_Irecv`), передающие одномерные и двумерные массивы (вектора и матрицы) между двумя процессорами. Проведите сравнение по скорости передачи данных в зависимости от применяемых функций и размера передаваемых данных.

Вопросы

1) Приведите пример программы, при выполнении которой заметна разница, в зависимости от того, используются ли функции блокирующего двухточечного обмена или неблокирующего.
2) С какой целью используется функция MPI_Get_count?
3) С какой целью используется функция MPI_Probe?
4) С какой целью используется функция MPI_Wait?
5) С какой целью используется функция MPI_Test?