Лабораторная работа 2

Знакомство со структурой MPI-программы и процедурами блокирующего двухточечного обмена

Цель работы:

Приобрести навыки составления MPI-программ. Изучить работу процедур блокирующего двухточечного обмена.

Задачи работы:

1) Изучить теоретические основы составления MPI-программ.
2) Изучить общую организацию MPI-программы, MPI-функции обмена сообщениями между процессами и коммуникаторы.
3) Выполнить задания лабораторной работы.
4) Ответить на вопросы

Теория

Системы программирования на основе передачи сообщений

Модель программирования на основе передачи сообщений подразумевает наличие параллельных процессов: каждый процесс оперирует своим набором данных, который не пересекается с наборами данных других процессов (отсутствует общая память). Взаимодействие и обмен данными между процессами происходит с помощью передачи сообщений.

Главным достоинством данной модели является возможность достаточно просто наращивать вычислительные мощности: масштабирование системы выполняется простым добавлением процессов. Главным недостатком данной модели является необходимость передач данных. Это усложняет программные модели и снижает производительность.

Стандарт MPI-1, описание интерфейса

При программировании на основе передачи сообщений чаще всего используется коммуникационная библиотека MPI (*Message Passing Interface*), фактически ставшая международным стандартом программирования модели с передачей сообщений. Реализации MPI существуют практически для всех многопроцессорных вычислительных систем. MPI включает большой набор средств, ключевыми являются подпрограммы передачи данных от одного процесса к другому (другим). MPI реализован для языков *C* и *Fortran* и поддерживает все типы данных, имеющиеся в этих языках.

MPI не является формальным стандартом, подобным тем, что выпускают организации стандартизации. Вместо этого он является стандартом, спроектированным на открытом форуме, который включал крупных поставщиков компьютеров, исследователей, разработчиков библиотек программ и пользователей.

Такое широкое участие в его развитии гарантировало быстрое превращение MPI в широко используемый стандарт для написания параллельных программ передачи сообщений.

«Стандарт» MPI был введен MPI-форумом в мае 1994 г. и обновлен в июне 1995 г. Документ, который определяет его, озаглавлен «*MPI: A Message Passing Standard*», и опубликован университетом Тэннеси.

В 1997-1998 гг. появился стандарт MPI-2, расширивший функциональные возможности первой версии.

Стандарт MPI имеет намного более одной реализации. Эти реализации обеспечивают асинхронную коммуникацию, эффективное управление буфером сообщений, эффективные группы и богатые функциональные возможности. MPI включает большой набор коллективных операций коммуникации, виртуальных топологий и различных способов коммуникации. В настоящее время имеются следующие реализации стандарта MPI:

1) *MPICH* – реализация Argonne National Lab.
2) *LAM* – реализация Ohio Supercomputer Center.
3) *MPI/Pro* – реализация MPI Software Technology.
4) *IBM MPI* – реализация IBM для кластерных рабочих станций SP и RS/6000.
5) *CHIMP* – реализация Edinburgh Parallel Computing Centre.
6) *UNIFY* – реализация Mississippi State University.

Общая организация MPI-программы

Основная схема программы MPI подчиняется следующим общим шагам:

1) Инициализация окружения MPI:
   - **`MPI_Init(…)`**
     инициализирует окружение MPI, эта функция вызывается один раз в начале программы, до первого вызова других MPI-функций.
2) Коммуникации между распределенными процессами:
   - **`MPI_Comm_size(…)`**
     возвращает число процессов внутри коммуникатора; по умолчанию существует предопределенный коммуникатор **`MPI_COMM_WORLD`**, который включает все процессы;
   - **`MPI_Comm_rank(…)`**
     возвращает ранг (номер) текущего процесса в коммуникаторе;
   - **`MPI_Send(…)`**
     отправляет сообщение;
   - **`MPI_Recv(…)`**
     получает сообщение.
3) Выход из системы передачи сообщений:
   - **`MPI_Finalize()`**
     выход из системы MPI; после вызова этой функции код программы может продолжить свое выполнение, но выполнять вызов MPI-функций уже нельзя.

Стандарт библиотеки MPI содержит описание более 120 функций. Все объекты (имена функций, константы, предопределенные типы данных) имеют префикс **`MPI_`**, следующая за префиксом буква должна быть заглавной. Если программист не будет описывать в своей программе имен с таким префиксом, то заведомо конфликтов имен не будет.

Программа, написанная на *C*, должна включать заголовочный файл – **`mpi.h`**, он содержит определения для констант и функций MPI. Поэтому в начале программы должна быть директива:

```
#include "mpi.h"
```

Если уже есть последовательная версия программы, то перед тем как сделать ее параллельной (используя MPI), лучше убедиться, что последовательная версия отлажена и корректно работает. После этого добавить вызовы функций MPI в соответствующие места программы.

Если MPI-программа создается с чистого листа и написать сначала последовательную программу (без вызовов MPI) не составит большого труда, то лучше сделать это (поиск и удаление непараллельных ошибок вначале намного облегчит процесс отладки параллельной программы).

Отлаживать параллельную версию лучше в следующей последовательности: убедиться, что запуски программы успешны на нескольких узлах, а затем увеличивать число узлов.

Обмен сообщениями между процессами

Основной способ обмена данными между процессами – это отправка и получение сообщений. Сообщения по количеству адресатов различают как:

- *попарные* – в обмене участвуют только два процесса: отправитель и получатель;
- *коллективные* – сообщение посылается одним процессом всем процессам из его группы.

Передача сообщений может выполняться:

- *синхронно* – отправитель сообщения ожидает пока получатель не будет готов принять сообщение, а получатель ожидает пока отправитель не будет готов отправить сообщение;
- *асинхронно* – отправитель не ждет готовности получателя принять сообщение, вместо этого сообщение передается в буфер и отправитель продолжает работать дальше. Получатель забирает сообщение из буфера.

Сообщение – это набор данных определенного типа, состоящий из двух частей:

1) Данные – информация, которая будет отослана или получена (начальный адрес, число элементов, тип данных).
2) Оболочка – используется для маршрутизации сообщения к получателю и связывает вызовы отправки с вызовами получения (номер получателя, тег, коммуникатор).

Прототип функции отправки сообщения:

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

где **buf** – начальный адрес данных; **count** – число элементов в буфере; **datatype** – тип данных каждого из элементов буфера; **dest** – ранг получателя в коммуникаторе comm; **tag** – тег сообщения; **comm** – коммуникатор.

Прототип функции получения сообщения:

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

где **buf** – адрес начала буфера принимаемых данных; **count** – число принимаемых элементов; **datatype** – тип данных каждого из элементов буфера; **source** – ранг отправителя в коммуникаторе comm; **tag** – тег сообщения; **comm** – коммуникатор, **status** – структура статуса принимаемого сообщения.

Итак, рассмотрим более подробно передаваемые в функции `MPI_Send()` и `MPI_Recv()` параметры.

Начальный адрес – адрес, с которого начинаются данные в памяти.

Число элементов – целое число данных в сообщении, это количество элементов (не байтов). Это реализовано для переносимости кода, чтобы не было необходимости беспокоиться о различных представлениях типов данных на различных компьютерах. Реализация MPI определяет число байт автоматически. Число, определенное при получении, должно быть больше или равно числу, определенному при отправке. Происходит ошибка, если посылается данных больше, чем имеется места в буфере принимающего процесса.

Тип данных – тип данных, которые будут передаваться, например целое с плавающей точкой (таблица 1). Тип данных при отправке и получении сообщения должен совпадать. Исключением из этого правила является тип данных **MPI_PACKED**, который является одним из способов обработки сообщений со смешанным типом данных (предпочтительным методом является метод с производными типами данных). И ещё одним исключением является тип **MPI_BYTE**, который используется, если система не должна выполнять преобразование между различными представлениями данных. Проверка типов в этих случаях не нужна.

Таблица 1 – Основные типы данных MPI в сравнении с типами данных языка Си

Типы данных MPI	Типы данных Си
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed longint
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned longint
MPI_FLOAT	Float

Типы данных MPI	Типы данных Си
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	—
MPI_PACKED	—

Могут быть также определены дополнительные типы данных, названные производными типами данных, что позволяет в некоторых случаях сделать программирование легче и обеспечить более быстрое выполнение кода.

Номер получателя – целое число, равное рангу (номеру) процесса в коммуникаторе (сами коммуникаторы будут рассмотрены далее). Ранг меняется от *0* до *(P–1)*, где *P* – это число процессов в коммуникаторе. Номер получателя используется, чтобы определить маршрутизацию сообщения к соответствующему процессу. *Номер процесса-источника* определяется в процессе-получателе, только сообщения, идущие от этого номера источника, будут приняты при вызове получения. Также процесс-получатель может установить источник в **MPI_ANY_SOURCE**, чтобы принимать сообщения от любого источника.

Тег – метка, произвольное число, которое помогает различать сообщения, например пришедшие от одного процесса. Теги, определяемые отправителем и получателем, должны совпадать, но получатель может принимать сообщения только с определенным значением тега или определить его как **MPI_ANY_TAG**, означающее, что допустим любой тег.

Номер коммуникатора, определенный при отправке должен равняться номеру коммуникатора, определенному при получении.

Коммуникаторы

В MPI-программе параллельные процессы можно объединять в группы коммуникаций – коммуникаторы. Процесс может входить в несколько коммуникаторов, у каждого процесса свой номер, отличающийся от коммуникатора к коммуникатору. Коммуникаторы могут пересекаться (по процессам), входить друг в друга.

При старте программы автоматически создается коммуникатор **MPI_COMM_WORLD**, в который входят все процессы. У каждого процесса два основных атрибута – коммуникатор и номер процесса в коммуникаторе.

Коммуникаторы гарантируют уникальные пространства сообщений, их можно использовать, чтобы ограничить коммуникацию на подмножество процессов.

MPI-программа разделяет процессы на коммуникаторы для:

- организации коллективных обменов внутри группы процессов (коллективный обмен можно представить как последовательность попарных обменов, но MPI-функции для коллективных обменов учитывают особенности конкретной вычислительной системы и могут выполняться значительно быстрее соответствующей последовательности попарных обменов);
- изоляции одних обменов сообщениями от других (например, для изоляции от сообщений, поступающих от вызванной из программы функции);

- учета топологии распределенной вычислительной системы (часто узлы распределенного кластера соединены линиями связи с различной скоростью, в этом случае при объединении процессов в коммуникаторы можно учесть скорость обмена между процессами).

Основные функции (представлены прототипы функций) для работы с коммуникаторами:

1) Получить количество процессов в коммуникаторе:

```
int MPI_Comm_size (MPI_Commcomm, int *size);
```

2) Получить номер процесса в коммуникаторе:

```
int MPI_Comm_rank (MPI_Commcomm, int *rank);
```

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 –o program source.c
```

где **program** – желаемое имя исполняемого файла программы, **source.c** – имя файла с исходным кодом программы.

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

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

где **X** – число запускаемых процессов, **program** – имя исполняемого файла программы.

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

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

т.е. будет запущено **10** процессов программы **program**, IP-адреса вычислительных узлов кластера будут получены из файла **bhosts**, до которого указан полный путь.

Лабораторная работа

Лабораторная работа состоит из четырёх заданий (общие задания 1-3 и одно задание по вариантам). В заданиях 1-3 лабораторной работы предлагается дописать фрагменты программ на языке *C*, в том числе, процедуры блокирующего двухточечного обмена. Пропущенные фрагменты обозначены многоточием.

Задание 1

В исходном тексте программы на языке C пропущены вызовы процедур подключения к MPI, определения количества процессов и ранга процесса. Добавить эти вызовы (использовать уже объявленные переменные), откомпилировать и запустить программу, проверить результат при различном числе параллельных процессов.

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

int main(int argc, char* argv[])
{
  int myid, numprocs;

  ....

  fprintf(stdout, "Process %d of %d\n", myid, numprocs);
  MPI_Finalize();
  return 0;
}
```

Задание 2

В исходном тексте программы на языке *C* пропущены вызовы процедур стандартного блокирующего двухточечного обмена. Предполагается, что при запуске двух процессов один из них отправляет сообщение другому, а тот его принимает. Добавить эти вызовы, откомпилировать и запустить программу.

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

int main(int argc, char* argv[])
{
  int numprocs, myrank;
  int myid;
  char message[20];
  MPI_Status status;
  int TAG = 0;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)
  {
    strcpy(message, "Hi, Second Processor!");
    MPI_Send(....);
  }
  else
  {
    MPI_Recv(....);
    printf("Received: %s\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

Задание 3

В исходном тексте программы на языке *C* пропущены вызовы процедур стандартного блокирующего двухточечного обмена. Предполагается, что при запуске чётного числа процессов, те из них, которые имеют чётный ранг, отправляют сообщение следующим по величине ранга процессам. Добавить эти вызовы, откомпилировать и запустить программу, проверить результат при различном числе параллельных процессов.

```
#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(....);
  }
  else
  {
    if(myrank != 0)
      MPI_Recv(....);
    printf("I %i received from %i\n", myrank, message);
  }
  MPI_Finalize();
  return 0;
}
```

Задания по вариантам

1) Создайте и выполните на разном числе процессоров программу, используя блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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 блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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 блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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 блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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 блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), реализующую алгоритм передачи массива псевдослучайных целых чисел от одного процессора другому. Определите максимально допустимую длину передаваемого сообщения (максимальный размер массива).
6) Напишите программу, используя блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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 блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), реализующую алгоритм передачи данных по кольцу: очередной процессор дожидается сообщения от предыдущего и потом посылает следующему процессору.

8) Создайте программу, используя блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), реализующую алгоритм передачи данных по кольцу: все процессора одновременно посылают и принимают сообщения.
9) Напишите программу, используя блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), реализующую алгоритм передачи данных по двум кольцам: нечетные процессора образуют первое кольцо, четные – второе.
10) Создайте программу, используя блокирующие коммуникационные функции (`MPI_Send`, `MPI_Recv`), 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) Что изменится, если команде запуска передать параметр `-loadbalance`?
2) Что означает параметр команды запуска `-bynode`?
3) Что означает параметр команды запуска `-nolocal`?
4) Приведите пример блокирующих операций передачи сообщений типа «точка-точка», который приводит параллельный алгоритм в тупиковую ситуацию.
5) В чем отличие блокирующих и неблокирующих функций?
6) Какого типа и какое значение возвращает функция `MPI_Wtime`?
7) Напишите программу для вывода имен узлов кластера (`MPI_Get_processor_name`).