Лабораторная работа №1

Передача и прием сообщений в MPI

Цель: изучить основные принципы приема и передачи сообщений в технологии MPI на примере использования в рамках языка C++.

Для получения **теоретических сведений** настоятельно рекомендуется при домашней подготовке изучить материалы, представленные в списке литературы в конце разработки, а также прочие материалы по тематике лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект материала, приведенного в данных источниках, в конце включающий короткие примеры фрагментов программ.

1. Понятие параллельной программы

Под **параллельной программой** в рамках MPI понимается множество одновременно выполняемых **процессов**. Процессы могут выполняться как на разных процессорах, так и на одном. Каждый процесс параллельной программы порождается на основе копии одного и того же программного кода (**модель SPMD**). Все процессы программы последовательно перенумерованы от 0 до **p-1**, где **p** есть общее количество процессов. Номер процесса именуется **рангом** процесса.

2. Операции передачи данных

Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются **парные** (**point-to-point**) операции между двумя процессами и **коллективные** (**collective**) коммуникационные действия для одновременного взаимодействия нескольких процессов.

3. Понятие коммуникаторов

Процессы параллельной программы объединяются в **группы**. Под **коммуникатором** в MPI понимается служебный объект, объединяющий в своем составе группу процессов и ряд дополнительных параметров (**контекст**), используемых при выполнении операций передачи данных. Один и тот же процесс может принадлежать разным группам и коммуникаторам. Все имеющиеся в программе процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (**intercommunicator**).

4. Типы данных

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать **тип** пересылаемых данных. MPI содержит большой набор **базовых типов** данных, во многом совпадающих с типами данных в языках C и Fortran. Кроме того, в MPI имеются возможности для создания новых **производных типов**.

5. Инициализация и завершение MPI программ

Первой вызываемой функцией MPI должна быть функция:

```
int MPI_Init(int *agrc, char ***argv);
```

для инициализации среды выполнения MPI-программы. Параметрами функции являются количество аргументов в командной строке и текст самой командной строки.

Последней вызываемой функцией MPI обязательно должна являться функция:

```
int MPI_Finalize(void);
```

Приведем **пример структуры параллельной программы**, разработанная с применением MPI:

```
#include "mpi.h"

int main(int argc, char *argv[])
{
 // <программный код без использования MPI функций>
 MPI_Init(&agrc, &argv);
 // <программный код с использованием MPI функций>
 MPI_Finalize();
 // <программный код без использования MPI функций>
 return 0;
}
```

- файл **mpi.h** содержит определения именованных констант, прототипов функций и типов данных библиотеки MPI;
- функции **MPI_Init** и **MPI_Finalize** являются обязательными и должны быть выполнены (и только один раз) каждым процессом параллельной программы;
- перед вызовом **MPI_Init** может быть использована функция **MPI_Initialized** для определения того, был ли ранее выполнен вызов **MPI_Init**.

6. Определение количества и ранга процессов

Определение **количества процессов** в выполняемой параллельной программе осуществляется при помощи функции:

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

Для определения **ранга процесса** используется функция:

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

Как правило, вызов функций **MPI_Comm_size** и **MPI_Comm_rank** выполняется сразу после **MPI_Init**:

```
#include "mpi.h"

int main(int argc, char *argv[])
{
 int ProcNum, ProcRank;
 // <программный код без использования MPI функций>
 MPI_Init(&agrc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
 // <программный код с использованием MPI функций>
 MPI_Finalize();
 // <программный код без использования MPI функций>
 return 0;
}
```

- коммуникатор **MPI_COMM_WORLD** создается по умолчанию и представляет все процессы выполняемой параллельной программы;
- ранг, получаемый при помощи функции **MPI_Comm_rank**, является рангом процесса, выполнившего вызов этой функции, переменная **ProcRank** будет принимать различные значения в разных процессах.

7. Передача сообщений

Для **передачи сообщения** процесс-отправитель должен выполнить функцию:

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

где
- buf – адрес буфера памяти, в котором располагаются данные отправляемого сообщения,

- count – количество элементов данных в сообщении,
- type - тип элементов данных пересылаемого сообщения,
- dest - ранг процесса, которому отправляется сообщение,
- tag - значение-тег, используемое для идентификации сообщений,
- comm - коммуникатор, в рамках которого выполняется передача данных.

Таблица 1. Базовые (предопределенные) типы данных MPI для языка C

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

- отправляемое сообщение определяется через указание буфера памяти, в котором это сообщение располагается;
- используемая для указания буфера триада (buf, count, type) входит в состав параметров практически всех функций передачи данных;
- процессы, между которыми выполняется передача данных, должны принадлежать коммуникатору, указываемому в функции **MPI_Send**.

Сразу же после завершения функции **MPI_Send** процесс-отправитель может начать повторно использовать буфер памяти, в котором располагалось отправляемое сообщение. Вместе с этим, следует понимать, что в момент завершения функции **MPI_Send** состояние самого пересылаемого сообщения может быть совершенно различным - сообщение может:
- располагаться в процессе-отправителе,
- находиться в процессе передачи,
- храниться в процессе-получателе
- или же может быть принято процессом-получателем при помощи функции **MPI_Recv**.

Завершение функции **MPI_Send** означает лишь, что операция передачи начала выполняться.

8 Прием сообщений

Для **приема сообщения** процесс-получатель должен выполнить функцию:

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

где
- buf, count, type – буфер памяти для приема сообщения, назначение каждого отдельного параметра соответствует описанию в MPI_Send,
- source - ранг процесса, от которого должен быть выполнен прием сообщения,
- tag - тег сообщения, которое должно быть принято для процесса,
- comm - коммуникатор, в рамках которого выполняется передача данных,
- status – указатель на структуру данных с информацией о результате выполнения операции приема данных.

Следует отметить:

- буфер памяти должен быть достаточным для приема сообщения, а тип элементов передаваемого и принимаемого сообщения должны совпадать; при нехватке памяти часть сообщения будет потеряна и в коде завершения функции будет зафиксирована ошибка переполнения;
- при необходимости приема сообщения от любого процесса-отправителя для параметра **source** может быть указано значение **MPI_ANY_SOURCE**;
- при необходимости приема сообщения с любым тегом для параметра **tag** может быть указано значение **MPI_ANY_TAG**;
- параметр **status** позволяет определить ряд характеристик принятого сообщения: - status.MPI_SOURCE – ранг процесса-отправителя принятого сообщения, status.MPI_TAG - тег принятого сообщения.

- функция:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count);
```

возвращает в переменной **count** количество элементов типа **type** в принятом сообщении.

Вызов функции **MPI_Recv** не должен согласовываться со временем вызова соответствующей функции передачи сообщения **MPI_Send** – прием сообщения может быть инициирован до момента, в момент или после момента начала отправки сообщения.

По завершении функции **MPI_Recv** в заданном буфере памяти будет располагаться принятое сообщение. Принципиальный момент здесь состоит в том, что функция **MPI_Recv** является **блокирующей** для процесса-получателя, т.е. его выполнение приостанавливается до завершения работы функции. Таким образом, если ожидаемое для приема сообщение будет отсутствовать, выполнение параллельной программы будет блокировано.

Пример 1.

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

int main(int argc, char* argv[])
{
 int ProcNum, ProcRank, RecvRank;
 MPI_Status Status;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
 if (ProcRank == 0)
 {
  // Действия, выполняемые только процессом с рангом 0
  printf ("\n Hello from process %3d", ProcRank);
  for ( int i=1; i < ProcNum; i++ )
  {
   MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
   MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
   printf("\n Hello from process %3d", RecvRank);
  }
 } else // Сообщение, отправляемое всеми процессами,
        // кроме процесса с рангом 0
  MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
 MPI_Finalize();
 return 0;
}
```

Каждый процесс определяет свой ранг, после чего действия в программе разделяются. Все процессы, кроме процесса с рангом 0, передают значение своего ранга нулевому процессу. Процесс с рангом 0 сначала печатает значение своего ранга, а далее последовательно принимает сообщения с рангами процессов и также печатает их значения. При этом, порядок приема сообщений заранее не определен. Для рассматриваемого простого примера можно восстановить постоянство получаемых результатов при помощи задания ранга процесса-отправителя в операции приема сообщения:

```
MPI_Recv(&RecvRank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
```

Указание ранга процесса-отправителя регламентирует порядок приема сообщений, и, как результат, строки печати будут появляться строго в порядке возрастания рангов процессов (однако такая регламентация может приводить к замедлению выполняемых параллельных вычислений).

Для разделения фрагментов кода между процессами обычно используется подход, примененный в только что рассмотренной программе - при помощи функции **MPI_Comm_rank** определяется ранг процесса, а затем в соответствии с рангом выделяются необходимые для процесса участки программного кода. Наличие в одной и той же программе фрагментов кода разных процессов усложняет понимание и, в целом, разработку MPI-программы.

Можно рекомендовать при увеличении объема разрабатываемых программ выносить программный код разных процессов в отдельные программные модули (функции). Общая схема MPI программы в этом случае будет иметь вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

Во многих случаях, как и в рассмотренном примере, выполняемые действия являются отличающимися только для процесса с рангом 0. В этом случае общая схема MPI программы принимает более простой вид:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoManagerProcess();
else DoWorkerProcesses();
```

В завершение обсуждения примера поясним использованный в MPI подход для контроля правильности выполнения функций - все функции MPI возвращают в качестве своего значения **код завершения**. При успешном выполнении функции возвращаемый код равен **MPI_SUCCESS**. Другие значения кода завершения свидетельствуют об обнаружении тех или иных ошибочных ситуаций в ходе выполнения функций. Для выяснения типа обнаруженной ошибки используются предопределенные именованные константы, среди которых:

- `MPI_ERR_BUFFER` – неправильный указатель на буфер,
- `MPI_ERR_COMM` – неправильный коммуникатор,
- `MPI_ERR_RANK` – неправильный ранг процесса,

и др. – полный список констант для проверки кода завершения содержится в файле **mpi.h**.

9. Определение времени выполнения MPI-программы

Получение времени текущего момента выполнения программы обеспечивается при помощи функции:

```
double MPI_Wtime(void);
```

результат вызова которой есть количество секунд, прошедшее от некоторого определенного момента времени в прошлом. Возможная схема применения функции **MPI_Wtime** может состоять в следующем:

```
double t1, t2, dt;
t1 = MPI_Wtime();
…
t2 = MPI_Wtime();
dt = t2 – t1;
```

OpenMP в MS Visual Studio 2010

В MS Visual Studio 2010 для использования возможностей технологии MPI, к примеру, в рамках консольного приложения, нужно предварительно проделать следующие действия:

1. Создать консольное приложение Win32 на C++ с именем, к примеру, ParallelPI. При этом, используйте проект без предкомпилированных заголовков.

2. В меню **File** выберите пункт Создать, а затем — Проект.

3. В диалоговом окне **New -> Project...** щелкните элемент **Установленные шаблоны** и выберите пункт **Visual C++**. (В зависимости от настроек **Visual Studio** пункт **Visual C++** может находиться под узлом **Other Languages**.)

4. В списке шаблонов щелкните элемент **Win32 Console Application**.

5. Введите имя проекта -> **ОК** -> **Next>** -> Снимите флажок **Precompiled header** -> **Finish**.

6. В обозревателе решений щелкните правой кнопкой мыши **на элемент с названием проекта** и выберите пункт Properties, и выберите пункт VC++Directories.

7. В области Include Directories поместите курсор в начало списка, который отображается в текстовом окне, и укажите местоположение файлов заголовков MPI CH, после чего введите точку с запятой (;).

Внимание! На терминале 4100 заголовки MPI находятся в поставке пакета SPB (версия 16.3), поэтому в случае работы на 4100, задайте:

C:\Program Files %28x86%29\SPB_16.3\share\pcb\pvt\ven_inc;

в иных случаях использования других реализаций MPI, место расположения данного каталога будет другим!

В области **Library Directories** поместите курсор в начало списка, который отображается в текстовом окне, и укажите местоположение файла библиотеки MPI CH, после чего введите точку с запятой (;).

Внимание! На терминале 4100 библиотеки MPI находятся в поставке пакета SPB (версия 16.3), поэтому в случае работы на 4100, задайте:

C:\Program Files %28x86%29\SPB_16.3\tools\pcb\pvt\lib;

в иных случаях использования других реализаций MPI, место расположения данного каталога будет другим!

Далее, в разделе **Linker** выберите элемент **Input**, в разделе Additional Dependencies поместите курсор в начало списка, который отображается в текстовом окне, и введите следующее:

mpich.lib;

После чего, можете приступать к созданию проекта, не забывая к коде подключать:

#include "mpi.h"

Внимание! Так как для работы проекта MPI необходимо создание приложением сокетов, а на терминале 4100 данная возможность **заблокирована**, скомпилированный проект необходимо перенести на (виртуальную) машину, где такая возможность есть!

Лабораторные задания (№ варианта = № компьютера % 3)

Задание. В соответствии с вариантом задания, написать на C++ программу, реализующую многопоточность на основе технологии MPI, работающую на основе программа должна работать на основе простой передачи сообщений.

Вариант	**Задание: написать программы, демонстрирующие работу следующей функции:**

0	Реализуйте функцию ring, которая создаёт N процессов и посылает сообщение первому процессу, который посылает сообщение второму, второй - третьему, и так далее. Наконец, процесс N посылает сообщение обратно процессу 1. После того, как сообщение обежало вокруг кольца M раз, все процессы заканчивают работу.
1	Реализуйте функцию star, которая создаёт N+1 процессов (1 «центральный» и N «крайних») и посылает сообщение центральному процессу, который посылает сообщение всем остальным процессам и дожидается от них ответа, после чего это повторяется (всего M раз). После того, как все события получены, все процессы заканчивают работу.
2	Реализуйте процесс-«счётчик», (который запускается со значением 0 и 1) если получена -1, то он выводит в текущее значение и заканчивает работу; 2) если получено любое другое сообщение, то значение увеличивается на 1 и выводится сообщение об этом.

Контрольные вопросы

1. В чем состоят основы технологии MPI?
2. В чем состоят основные преимущества и недостатки технологии MPI?
3. Что понимается под параллельной программой в рамках технологии MPI?
4. Как происходит инициализация и завершение MPI программ?
5. Как происходит передача и прием сообщений MPI программе?

Требования к сдаче работы

1. При домашней подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчёт результаты выполнения.
2. Продемонстрировать программный код для лабораторного задания.
3. Продемонстрировать выполнение лабораторных заданий (можно в виде скриншотов, если работа была выполнена дома).
4. Ответить на контрольные вопросы.
5. Показать преподавателю отчет.

Литература

1. Спецификации стандарта Open MPI (версия 1.6, на английском языке): http://www.open-mpi.org/doc/v1.6/
2. Материалы, представленные на сайте intuit.ru в рамках курса «Intel Parallel Programming Professional (Introduction)»: http://old.intuit.ru/department/supercomputing/ppinteltt/5/
3. С.А. Лупин, М.А. Посыпкин Технологии параллельного программирования. – М.: ИД «ФОРУМ»: ИНФРА-М, 2011. – С. 12-96. *(Глава, посвященная MPI)*
4. Отладка приложений MPI в кластере HPC http://msdn.microsoft.com/ru-ru/library/dd560808(v=vs.100).aspx
5. http://www.parallel.ru/tech/tech_dev/mpi.html
6. http://msdn.microsoft.com/ru-ru/library/ee441265(v=vs.100).aspx#BKMK_debugMany