Федеральное агентство по образованию
Государственное образовательное учреждение высшего профессионального образования
Самарский государственный аэрокосмический университет имени академика С.П. Королева

Изучение технологии удаленного доступа к ресурсам и реализация простейших вычислительных схем на многопроцессорной системе

Методические указания к лабораторной работе №13

САМАРА 2009

Составители: Никоноров А.В., Фурсов В.А.

УДК

Изучение технологии удаленного доступа к ресурсам и реализация простейших вычислительных схем на многопроцессорной системе / Сост. Никоноров А.В. - Самара: Изд-во Самарского государственного аэрокосмического университета, 2009. 22 с.

В настоящих методических указаниях приведен материал, необходимый для выполнения лабораторных работ по дисциплине «Информатика» учебных планов машиностроительных факультетов.

Целью лабораторных работ является изучение суперкомпьютерных технологий, в частности, технологий удаленного доступа к ресурсам и реализации простейших вычислительных схем на многопроцессорной системе.

Методические указания предназначены для студентов дневного и вечернего отделений всех специальностей машиностроительных факультетов.

Печатается по решению редакционно-издательского совета Самарского государственного аэрокосмического университета

Рецензент: Попов С.Б.

СОДЕРЖАНИЕ

1. Цель лабораторной работы 4
2. Необходимое оборудование....4
3. Сведения, необходимые для проведения работы 4
3.1. Доступ к кластеру. 4
3.2 Этапы разработки на кластере. 7
3.3 Жизненный цикл процессов MPI и простейший обмен данными между ними, тупиковые ситуации 10
4. Порядок выполнения работы 19
5. Задание к лабораторной работе 22
5. Контрольные вопросы 22
6. Список литературы 22

1. Цель лабораторной работы

– Изучение технологии удаленного доступа к ресурсам многопроцессорной вычислительной системы;
– приобретение практических знаний и навыков в компиляции и запуске простейших MPI-программ;
– получение навыков составления и реализации параллельных программ для решения простейших задач на высокопроизводительной многопроцессорной системе;
– получение практических навыков работы с базовыми функциями библиотеки MPI.

2. Необходимое оборудование

Вычислительный кластер под управлением UNIX-совместимой ОС, предустановленная поддержка MPI, рабочая консоль программиста для управления прохождением пользовательских задач.

3. Сведения, необходимые для проведения работы

3.1. Доступ к кластеру.

Стандарт MPI – это стандарт, допускающий различные прикладные реализации, существует, например, проприетарная реализация Intel MPI, оптимизированная под архитектуру Intel. Open source реализация MPI разрабатывается в университете Чикаго и носит название mpich. В настоящей методичке рассматривается mpich версии 2.

Несмотря на то, что mpich 2 может работать под управлением различных ОС, и, в частности, windows, стандартом кластерной ОС является Linux. В нашем

конкретном случае мы рассмотрим rpm-based систему CentOS.

Удаленный доступ к головной машине кластера осуществляется посредством SSH. Бесплатный SSH клиент для windows – putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html). Для соединения с кластером нужно в строке Host Name указать имя хоста (или IP адрес) кластера. В нашем случае – cluster410.ssau.ru, далее свой логин и пароль (рис. 1). Для корректного отображения unicode кодировки в сеансе SSH необходимо в настройках putty в разделе Window->Translation указать кодировку UTF-8, вместо принятой по-умолчанию кодировки KOI8-R.

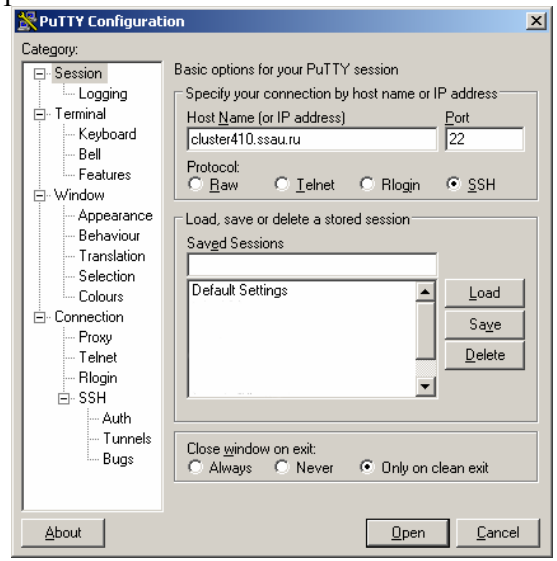

Рисунок 1. Окно ssh клиента.

SSH не только обеспечивает удаленный доступ к системной консоли Linux машины, реализуя так называемый shell-доступ, но позволяет выполнять копирование файлов по шифрованному каналу. Для файлового обмена можно использовать консольный аналог ftp – sftp, а можно воспользоваться WinSCP, программой, которая позволяет

работать с файлами удаленной Linux системой как с сетевым диском. Программа WinSCP бесплатная и может быть получена с сайта http://winscp.net/eng/docs/lang:ru.

Таким образом, разработка программы под mpi может быть выполнена в привычном для пользователя программном окружении, а потом скопирована на кластер посредством SSH. Однако с текстом программы можно работать и непосредственно в сессии удаленного доступа. В Linux существует множество текстовых редакторов, которые позволяют это сделать, например, nano или vim. На наш взгляд, наиболее удобным является редактор mcedit, редактор, встроенный в файловый менеджер Midnight Commander или mc (рис). Работа с mc очень похожа на работу с Far Manager в Windows. Оба файловых менеджера имеют одного предка – Norton Commander for DOS. Так что знакомым с NC освоится в mc не составит труда.

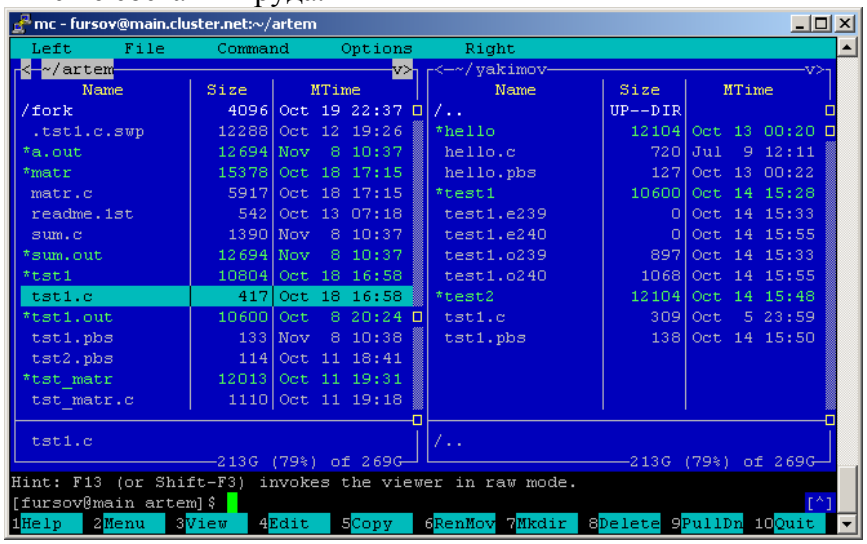

Рисунок 2. Окно Midnight Commander.

Перечислим некоторые, наиболее часто используемые операции. Переключение между отображением окна mc и командной строкой производится одновременным нажатием клавиш Ctrl+O. Для создания нового файла текста программы

нажмите Shift+F4. В открывшемся окне редактирования можно набирать текст программы, а затем при сохранении файла будет выдан запрос на ввод имени нового файла. При редактировании можно вставлять текст из буфера обмена ОС MS Windows.

Оболочка mc позволяет провести все необходимые процедуры по написанию программы, запуску ее на кластере и анализ результатов исполнения. Рассмотрим эти процедуры подробнее.

3.2 Этапы разработки на кластере.

0. Написание текста программы, например,
Hello_mpi.c

1. Компиляция программы:
mpicc -o Hello_mpi Hello_mpi.c

Утилита mpicc это скрипт, выполняющий вызов стандартного компилятора gcc с передачей путей к заголовочным файлам и библиотекам пакета mpi.

2. Запуск программы на кластере:

qsub Hello_mpi.pbs,

На самом деле, выполняется не непосредственно запуск, а регистрация программы в очереди на исполнение. Регистрация выполняется при помощи планировщика MPI-PBS, на основании файла описания задания – Hello_mpi.pbs. Формат файла описан ниже.

3.Проверить состояния задачи в очереди можно утилитой qstat.

```
[fursov@main mpi]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
236.main                  Hello_mpi        fursov                 0 R dque
```


В колонках таблицы выводимой утилитой присутствуют следующие поля. Job id – номер задачи, Name – название задачи, как оно задано в файле Hello_mpi.pbs, Time – используемое процессорное время, S – состояние задачи. В поле S может быть: R означает, что задача выполняется, E – задача в состоянии ошибки, C – исполнение задачи завершено.

4. После завершения выполнения задачи планировщиком в каталоге появятся два файла – Идентификатор_задачи.oНомер_задачи и Идентификатор_задачи.eНомер_задачи, в этих файлах сохранен поток стандартного вывода (STDOUT) и поток ошибок (STDERR).

Работа с файлом .pbs

Рассмотрим пример .pbs файла.

```
#PBS -N Hello_mpi
#PBS -l nodes=2:ppn=4
#PBS -l walltime=00:01:00
cd $PBS_O_WORKDIR

mpiexec-pbs -np 8 /home/fursov/mpi/Hello_mpi
```

Определение идентификатора задачи:
#PBS -N Hello_mpi

Указание количества узлов и потоков для выполнения задачи, в данном случае, использовано 2 узла (nodes) и по 4 потока на каждом узле (ppn). Итого, наша задача будет выполняться в 8 потоках.
#PBS -l nodes=2:ppn=4

Определение максимального времени для выполнения задачи (walltime) в формате ЧЧ:ММ:СС, по истечению указанного интервала задача будет принудительно завершена планировщиком.

```
#PBS -l walltime=00:01:00
```

Механизм принудительное завершения или как его еще называют, сторожевой таймер, предохраняет систему от зависших задач. На этапе отладки, когда есть вероятность ошибок в программе приводящих к блокировке или бесконечному циклу не рекомендуется ставить walltime больше нескольких минут. Если значение walltime не указано то оно принимается равным значению по-умалчанию, как правило это 60 секунд.

Установка рабочего каталога. Кроме установки каталога, могут быть выполнены дополнительные команды инициализации, например, копирование необходимых для вычисления файлов.

```
cd $PBS_O_WORKDIR
```

Строка, указывающая путь к исполняемому файлу.

```
mpiexec-pbs -np 8 /home/fursov/mpi/Hello_mpi
```

Дополнительно можно указать опцию, которая позволит объединить файлы стандартного вывода и ошибок в один.

```
#PBS -j oe
```

Задача, описанная pbs файлом отправляется в очередь командой qsub, задача может быть удалена из очереди при помощи:

qdel Идентификатор_задачи.

3.3 Жизненный цикл процессов MPI и простейший обмен данными между ними, тупиковые ситуации

Общие сведения. При использовании компактно расположенных кластеров исполняемый код приложения компилируется главной машиной, рассылается по вычислительным узлам (ВУ) и запускается на каждом средствами ОС. Момент старта каждой ветви программы может отличаться от такового на других ВУ, также нельзя априори точно определить момент выполнения любого оператора внутри конкретной ветви, см. рис.3 (именно поэтому применяются различного типа приемы синхронизации при обмене сообщениями); Для практического осознания необходимости синхронизации процессов при параллельном программировании служит первая часть данной работы. Во второй части практически рассматривается формальный обмен данными между процессами (использование простейших функций передачи MPI_Send и приема данных MPI_Recv).

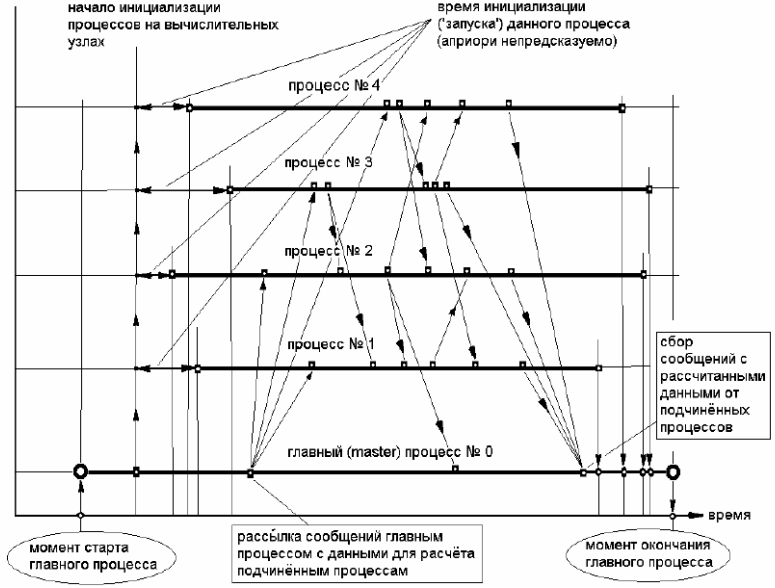

Рисунок 3. Схема жизненного цикла MPI процесса.

В MPI не используются привычные для C типы данных (int, char и др.), вместо них удобно применять определенные для данной платформы константы MPI_INT, MPI_CHAR и т.д. (см. табл.1).

Таблица 1.— Предопределенные в MPI константы типов данных.

Константы MPI	*Соответствующий тип в C*
MPI_INT	signed int
MPI_UNSIGNED	Unsigned int
MPI_SHORT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_SHORT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	long double
MPI_UNSIGNED_CHAR	unsigned char
MPI_CHAR	signed char

Пользователь может зарегистрировать в MPI свои собственные типы данных (например, структуры), после чего MPI сможет обрабатывать их наравне с базовыми.

Практически в каждой MPI-функции одним из параметров является коммуникатор (идентификатор группы процессов); в момент инициализации библиотеки MPI создается коммуникатор MPI_COMM_WORLD и в его пределах процессы нумеруются линейно от 0 до size.

Изучение MPI традиционно начнем с создания простейшей программы, которая сообщает о себе окружающему миру, т.е. каждый запускаемый процесс выдает следующее сообщение:

```
Hello world from process i of n
```


Здесь i – номер процесса, а n – количество процессов.

Используя редактор, набираем текст программы **helloworld.c**:

:#include

```
#include "mpi.h"

int main( argc, argv )
int  argc;
char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n",
rank, size );
    MPI_Finalize();
    return 0;
}
```

В этой простой программе используется 4 функции MPI. Далее будет введено еще только 3 функции. В настоящем пособии, в соответствии с поставленной задачей, мы намеренно ограничимся изучением лишь 6 функций MPI, достаточных для написания простейших параллельных программ. Для более полного знакомства с библиотекой MPI можно рекомендовать учебное пособие [5].

Любая прикладная MPI-программа (приложение) должна начинаться с вызова функции инициализации MPI (функция MPI_Init). В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. В нашем случае величина groupsize равна числу процессоров, выделенных задаче.

Синтаксис *функции инициализации* ***MPI_Init*** значительно отличается в языках Си и Фортран:

C:

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

FORTRAN:

```
MPI_INIT(IERROR)
INTEGER IERROR
```

В программах на Си каждому процессу при инициализации передаются аргументы функции main, полученные из командной строки. В программах на языке Фортран параметр IERROR является выходным и возвращает код ошибки.

Функция завершения MPI программ ***MPI_Finalize***.

C:

```
int MPI_Finalize(void)
```

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Функция определения числа процессов в области связи ***MPI_Comm_size***.

C:

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

IN	comm	- коммуникатор;
OUT	size	- число процессов в области связи коммуникатора comm.

(здесь и далее при обсуждении параметров процедур символами **IN** будем указывать входные параметры процедур, символами **OUT** выходные, а **INOUT** - входные параметры, модифицируемые процедурой).

Функция возвращает количество процессов в области связи коммуникатора comm.

До создания явным образом групп и связанных с ними коммуникаторов единственно возможными значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI.

Функция определения номера процесса ***MPI_Comm_rank***.

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


IN	comm	- коммуникатор;
OUT	rank	- номер процесса, вызвавшего функцию.

Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции).

Если откомпилировать, скомпоновать и запустить программу **helloworld,** например, на 4 процессорах, то получиться следующий вывод:

```
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
Hello world from process 2 of 4
```

Далее, для того чтобы завершить рассмотрение всех шести функций MPI, которыми мы решили ограничиться в настоящем пособии, создадим программу, которая рассылает некоторое сообщение (данные) по цепочке запущенных процессов.

Данные состоят из одного целочисленного значения, которое процесс 0 получает от пользователя. Признаком завершения рассылки является ввод отрицательного числа.

Текст программы **ring.c**:

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

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    do {
```

```
        if (rank == 0) {
            scanf( "%d", &value );
            MPI_Send( &value, 1, MPI_INT, rank + 1,
0, MPI_COMM_WORLD );
        }
        else {
            MPI_Recv( &value, 1, MPI_INT, rank - 1,
0, MPI_COMM_WORLD,
                      &status );
            if (rank < size - 1)
                MPI_Send( &value, 1, MPI_INT, rank +
1, 0, MPI_COMM_WORLD );
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);

    MPI_Finalize( );
    return 0;
}
```

В приведенной программе используются все 6 функций MPI, которые изучаются в настоящем пособии. Четыре из них уже использовались нами в примере на странице 9. Поэтому подробно рассмотрим лишь две из них (*MPI_Send, MPI_Recv*), которые введены в настоящем примере впервые.

Функция передачи сообщения ***MPI_Send***.

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

IN	buf	- адрес начала расположения пересылаемых данных;
IN	count	- число пересылаемых элементов;
IN	datatype	- Тип посылаемых элементов;
IN	dest	- номер процесса-получателя в группе, связанной с коммуникатором comm;
IN	tag	- идентификатор сообщения (очень часто является порядковым номером сообщения в последовательности);
IN	comm	- коммуникатор области связи.

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf - это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Существует модификация функции MPI_Send, которая потребуется нам при написании программы суммирования. Данная модификация, **MPI_Bcast**, позволяет разослать сообщение всем процессам некоторого коммуникатора. У этой функции те же параметры, что и у MPI_Send, за исключением четвертого, который отсутствует.

Функция приема сообщения ***MPI_Recv***.

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

OUT	buf	- адрес начала расположения принимаемого сообщения;
IN	count	- максимальное число принимаемых элементов;
IN	datatype	- Тип элементов принимаемого сообщения;
IN	source	- номер процесса-отправителя;
IN	tag	- идентификатор сообщения;
IN	comm	- коммуникатор области связи;
OUT	status	- атрибуты принятого сообщения.

Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

В функции MPI_Recv при указании номера процесса-отправителя возможно использование специального параметра MPI_ANY_SOURCE ("принимай от кого угодно"), а в качестве идентификатора получаемого сообщения – MPI_ANY_TAG ("принимай что угодно"). Это так называемые параметры-джокеры, MPI резервирует для них отрицательные целые числа, в то время как реальные

идентификаторы процессов и сообщений лежат всегда в диапазоне от 0 до 32767. Пользоваться джокерами следует с осторожностью, потому что по ошибке таким вызовом MPI_Recv может быть захвачено сообщение, которое должно приниматься в другой части процесса-получателя.

Если логика программы достаточно сложна, использовать джокеры можно **только** в функциях проверяющих наличие сообщения для процесса (MPI_Probe и MPI_Iprobe), чтобы перед фактическим приемом узнать тип и количество данных в поступившем сообщении. Несмотря на то, что мы хотим получить "что угодно", тип принимаемых данных в функции MPI_Recv должен быть указан явно, а он может быть разным в сообщениях с разными идентификаторами.

Пример вывода программы ring.c приводится ниже.

```
Process 0 got 10
-1
Process 0 got -1
Process 3 got 10
Process 3 got -1
Process 2 got 10
Process 2 got -1
Process 1 got 10
Process 1 got -1
```

Описанные выше функции реализуют *стандартный режим с блокировкой.* Следует заметить, что возврат из блокирующей функции передачи данных еще не означает, что передача завершена, гарантируется только возможность повторного использования буфера данных для внесения в него изменений, которые уже не повлияют на отправляемые данные.

В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. В дополнение к стандартному режиму возможно использование синхронной, буферизованной или согласованной передачи, как с

блокировкой, так и без блокировки. С помощью только пары операций MPI_Send/MPI_Recv возможно реализовать практически любой параллельный алгоритм.

Пользователи, освоившие работу с описанными в настоящем пособии шестью функциями MPI, могут попытаться построить более эффективный параллельный код с использованием так называемых коллективных функций MPI. Подробное описание коллективных функций и примеры параллельных программ, в которых они используются, можно найти в учебном пособии [4]. В заключение еще раз подчеркнем, что для написания многих параллельных программ, обычно достаточно рассмотренных в настоящем пособии шести функций.

При обмене данными в некоторых случаях возможны *вызванные взаимной блокировкой* т.н. тупиковые ситуации (используются также термины ‘deadlock’, ‘клинч’); в этом случае функции пос**ы**лки и приема данных мешают друг другу и обмен не может состояться. Ниже рассмотрена deadlock-ситуация при использовании для перес**ы**лок разделяемой памяти.

Вариант 1. *Ветвь 1 :*
Recv (из ветви 2)
Send (в ветвь 2)
Ветвь 2 :
Recv (из ветви 1)
Send (в ветвь 1)

Вариант 1 приведет к deadlock’у при любом используемом инструментарии, т.к. функция приема не вернет управления до тех пор, пока не получит данные; из-за этого функция передачи не может приступить к отправке данных, поэтому функция приема не вернет управление... и т.д. до бесконечности.

Вариант 2. *Ветвь 1 :*

Send (в ветвь 2)
Recv (из ветви 2)
Ветвь 2 :
Send (в ветвь 1)
Recv (из ветви 1)

Казалось бы, что (если функция передачи возвращает управление толькопосле того, как данные попали в пользовательский буфер на стороне приема) и здесь deadlock неизбежен. Однако при использовании MPI завис**а**ния во втором варианте не произойдет: функция MPI_Send, если на приемной стороне нет готовности (не вызвана MPI_Recv), не станет дожидаться ее вызова, а скопирует данные во временный буфер и *немедленно вернет управление основной программе*. Вызванный далее MPI_Recv данные получит *не напрямую из пользовательского буфера, а из промежуточного системного* буфера (т.о. используемый в MPI способ буферизации повышает надежность – делает программу более устойчивой к возможным ошибкам программиста). Т.о. наряду с полезными качествами (см. выше) свойство блокировки может служить причиной возникновения (трудно локализуемых и избегаемых для непрофессионалов) тупик**о**вых ситуаций при обмене сообщениями между процессами.

4. Порядок выполнения работы

Работа заключается в подготовке исходных текстовых MPI-программ, их компиляции в исполн**и**мое приложение, запуске на счет и последующем анализе выходных данных программы. Работа выполняется в следующей последовательности.

Часть 1. Первой задачей является компиляция простейшей MPI-программы ring.c, з**а**пуск ее на исполнение на заданном преподавателем числе ВУ и анализ результатов.

Часть 2. Задачей является формальное освоение использования простейших функций передачи и приема данных (MPI_Send и MPI_Recv) а также функции групповой рассылки – MPI_Bcast, а также анализ простейшего параллельного вычислительного алгоритма суммирования числовых значений.

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

//Инициализация массива суммируемых чисел случайными
//значениями
void DataInitialization(double *x, int N){
      int i;

      for(i = 0; i++; i < N){
        x[i] = rand();
      }
}

int main(int argc, char* argv[]){
      int i;
      double x[100], TotalSum = 0.0, ProcSum = 0.0;
      int  ProcRank, ProcNum, N=100, k, i1, i2;
      MPI_Status Status;

      // Инициализация
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
      MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
```

```
// Подготовка данных
if ( ProcRank == 0 ) DataInitialization(x,N);

// Рассылка данных на все процессы
MPI_Bcast(x, N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

// Вычисление частичной суммы на каждом из процессов
// на каждом процессе суммируются элементы вектора x
//от i1 до i2

k = N / ProcNum;
i1 = k *   ProcRank;
i2 = k * ( ProcRank + 1 );
if ( ProcRank == ProcNum-1 ) i2 = N;
for ( i = i1; i < i2; i++ )
ProcSum  = ProcSum + x[i];

// Сборка частичных сумм на процессе с рангом 0
if ( ProcRank == 0 ) {
  TotalSum = ProcSum;
  for ( i = 1; i < ProcNum; i++ ) {

  MPI_Recv(&ProcSum,1,MPI_DOUBLE,MPI_ANY_SOU
RCE,0, MPI_COMM_WORLD, &Status);
          TotalSum = TotalSum + ProcSum;
          }
  }
  else // Все процессы отсылают свои частичные суммы
          MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0,
          MPI_COMM_WORLD);

  // Вывод результата
  if ( ProcRank == 0 )
    printf("\nTotal Sum = %10.2f",TotalSum);
  MPI_Finalize();
```



```
    return 0;
}
```

5. Задание к лабораторной работе

Общее задание №1.

1. Скомпилировать и выполнить программу ring.c.

Варианты задания №2.

1. Скомпилировать и выполнить программу суммирования.
2. Модифицировать программу суммирования для вычисления факториала.
3. Модифицировать программу суммирования для вычисления скалярного произведения векторов.

5. Контрольные вопросы

Вопросы для самопроверки:

1. Из каких стадий состоит жизненный цикл процессов при параллельном программировании?
2. Почему нельзя точно определить момент старта процессов на вычислительных узлах?
3. Что такое коммуникатор?
4. Какие вы знаете MPI функции рассылки и получения сообщений?
5. Какие функции начинают и завершают MPI программу?
6. Как узнать номер параллельного MPI процесса?

6. Список литературы

1. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002 г.
2. Гергель В.П. Теория и практика параллельных вычислений: учебное пособие/ В.П. Гергель.- М.:

Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2007. – 423 с.
3. Головашкин Д.Л. Методы параллельных вычислений. Часть I. Учебное пособие. Изд. СГАУ, Самара. 2002, 92 с.
4. Кравчук В.В. и др. Введение в программирование для параллельных ЭВМ и кластеров: Учебн. пособие/ Авторы-сост.: Кравчук В.В., Попов С.Б., Привалов А.Ю., Фурсов В.А., Шустов В.А.; Самар. научный центр РАН, Самар. гос. аэрокосм. ун-т. Самара. 2000. 87 с.