МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П.КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)»

С. Б. Попов

Методические указания к лабораторному практикуму

по параллельному программированию на базе высокопроизводительной кластерной системы HP BladeSystem c3000 Enclosure
для студентов направления 010400.62
Прикладная математика и информатика, бакалавриат

Самара
2011

Автор: ПОПОВ Сергей Борисович

Методические указания к лабораторному практикуму по курсу «Параллельное программирование» предназначены для бакалавров четвертого курса факультета информатики направления 010400.62 «Прикладная математика и информатика».

Содержание

1 Технология параллельного программирования MPI

1.1 Общие сведения и основные функции

Наиболее распространенный подход к параллельному программированию на MPP-системах (massively parallel processing systems – системах с укрупнено распараллеленной обработкой) заключается в следующем. На каждом компьютере параллельной вычислительной системы (например, кластерной архитектуры) запускается программа, написанная на стандартном языке последовательного программирования. Каждая такая программа либо обрабатывает свою порцию данных, либо выполняет свою операцию обработки. Взаимодействие этих программ обеспечивается путем вызова функций синхронизации, посылки и получения сообщений, которые объединены в специальную библиотеку.

Фактическим стандартом параллельного программирования с использованием такого подхода сейчас является библиотека MPI (Message Passing Interface – Интерфейс Передачи Сообщений).

MPI-программа представляет собой программу, которая запускается одновременно на нескольких процессорах. Каждая копия программы выполняется как отдельный процесс. С помощью служебных функций MPI процесс может получить информацию о количестве одновременно запущенных процессов и свой номер. Таким образом, в одной и той же программе в зависимости от номера процесса может выполняться различный код, или же код программы параметризуется номером процесса. Коммуникационные функции MPI предоставляют процессам MPI-программы различные способы взаимодействия. Возможны как индивидуальные, так и групповые операции обмена данными.

Для запуска MPI-программы используется команда `mpirun` (специальный командный файл – скрипт), при этом указывается количество запускаемых процессов и имя файла исполняемого модуля (программы). Дополнительно можно задать конфигурацию (список компьютеров) на которой запускается программа. Количество процессов не обязательно равно количеству процессоров. Возможен запуск MPI-программы с количеством процессов большим, чем число доступных процессоров, при этом несколько процессов должны одновременно выполняться на одном процессоре. Выполнение программы с указанием запуска нескольких процессов на однопроцессорном компьютере позволяет производить первоначальную отладку параллельного приложения.

При инициализации MPI создается предопределенная область связи, содержащая все процессы MPI-программы, с которой связывается предопределенный коммуникатор `MPI_COMM_WORLD`. В большинстве случаев именно он используется в качестве коммуникатора при задании параметров функций MPI. Используемые здесь и далее понятия *область*

связи и *коммуникатор области связи*, подробно рассматривались в [5]. При начальном знакомстве с библиотекой MPI достаточно знать, что они автоматически создаются при запуске любой программы.

Предварительно рассмотрим основные соглашения по использованию библиотеки MPI в языках программирования Си и Фортран.

Регистр символов при указании имен функций (процедур) и именованных констант существенен в Си, и не играет роли в Фортране. Все идентификаторы начинаются с префикса `MPI_`. Не рекомендуется использовать собственные идентификаторы, начинающиеся с этого префикса, а также с префиксов `MPID_`, `MPIR_` и `PMPI_`, которые применяются в служебных целях. Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами: `MPI_COMM_WORLD`, `MPI_FLOAT`. В именах функций первая за префиксом буква – заглавная, остальные маленькие: `MPI_Send`, `MPI_Comm_size`. Определение всех именованных констант, прототипов функций и определение типов выполняется в языке Си подключением файла `mpi.h`, а в Фортране – `mpif.h`.

В операциях пересылки и преобразовании данных необходимо указывать собственные типы MPI. Это связано как с обеспечением переносимости программ между различными компиляторами и платформами, так и с возможностью запуска параллельного приложения на гетерогенных вычислительных системах. В последнем случае MPI обеспечивает автоматическое преобразование типов данных при пересылках.

Список предопределенных типов для Си-версии библиотеки MPI приводится в таблице 1, а для языка Фортран – в таблице 2.

Таблица 1

Соответствие между MPI-типами и типами языка Си

тип MPI	*тип языка Си*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
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
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Таблица 2

Соответствие между MPI-типами и типами языка Фортран

тип MPI	*тип языка Фортран*
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Тип `MPI_BYTE` используется для передачи двоичной информации без какого-либо преобразования. Тип `MPI_PACKED` используется для передачи в одном сообщении разнотипных данных, предварительно упакованных специальными функциями. Кроме того, программисту предоставляются средства создания собственных типов на базе стандартных.

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 <stdio.h>
#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 2 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 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
```
INTEGER IERROR
MPI_INIT(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`**. Функция закрывает все MPI-процессы и ликвидирует все области связи.

C
```
int MPI_Finalize(void)
```
FORTRAN
```
INTEGER IERROR
MPI_FINALIZE(IERROR)
```

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

C
```
int MPI_Comm_size(MPI_Comm comm, int *size)
```
FORTRAN
```
INTEGER COMM, SIZE, IERROR
MPI_COMM_SIZE(COMM, SIZE, IERROR)
```

Параметры функции `MPI_Comm_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`**.

C
```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```
FORTRAN
```
INTEGER COMM, RANK, IERROR
MPI_COMM_RANK(COMM, RANK, IERROR)
```

Параметры функции `MPI_Comm_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).

Теперь необходимо откомпилировать, скомпоновать и запустить программу **`helloworld`**, например, на 4 процессорах.

```
% mpicc -o helloworld helloworld.c
% mpirun -np 4 helloworld
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
%
```

Заметим, что порядок появления сообщений от различных процессов не определен. Если появляется какое-либо сообщение об ошибке, например:

```
mpicc: Command not found.
```

то, скорее всего, необходимо в переменной окружения PATH указать каталог, где находятся исполняемые файлы библиотеки MPI. Например,

```
setenv PATH /usr/local/mpi/bin:$PATH
rehash
```

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 <stdio.h>
#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 ранее. Поэтому подробно рассмотрим лишь две из них (`MPI_Send`, `MPI_Recv`), которые введены в настоящем примере впервые.

Функция передачи сообщения **`MPI_Send`**.

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

FORTRAN
```
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,
IERROR)
```

Параметры функции `MPI_Send`:

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`.

Функция приема сообщения **`MPI_Recv`**.

C
```
int MPI_Recv(void* buf, int count, MPI_Datatype
```

```
                    datatype, int source, int tag, MPI_Comm
                    comm, MPI_Status *status)
```

FORTRAN

```
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
        STATUS(MPI_STATUS_SIZE), IERROR
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
        STATUS, IERROR)
```

Параметры функции `MPI_Recv`:

OUT `buf` – адрес начала расположения принимаемого сообщения;

IN `count` – максимальное число принимаемых элементов;

IN `datatype` – тип элементов принимаемого сообщения;

IN `source` – номер процесса-отправителя в группе, связанной с коммуникатором `comm`;

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.

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 запуска программы `ring.c` приводится ниже.

```
% mpicc -o ring ring.c
% mpirun -np 4 ring
10
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.

1.2 Коллективные операции

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

- Синхронизацию всех процессов с помощью барьеров (MPI_Barrier).
- Коллективные коммуникационные операции, в число которых входят:
    - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
    - сборка (gather) распределенного по процессам массива в один

массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
    - сборка (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
    - разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
    - совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами, в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).
- Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
    - с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
    - с рассылкой результата всем процессам (MPI_Allreduce);
    - совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
    - префиксная редукция (MPI_Scan).

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант, который предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

Отличительные особенности коллективных операций:

- Коллективные коммуникации не взаимодействуют с коммуникациями типа точка-точка.
- Коллективные коммуникации выполняются в режиме с блокировкой.
- Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию.
- Количество получаемых данных должно быть равно количеству посланных данных.
- Типы элементов посылаемых и получаемых сообщений должны совпадать.
- Сообщения не имеют идентификаторов.

Краткая сводка по функциям коллективных операций:

```
int MPI_Barrier(MPI_Comm comm)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm)

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
*recvcounts,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Таблица 3. Предопределенные операции в функциях редукции MPI.

Название	Операция	Разрешенные типы
MPI_MAX MPI_MIN	Максимум Минимум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сумма Произведение	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логическое AND Логическое OR Логическое исключающее OR	C integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Поразрядное AND Поразрядное OR Поразрядное исключающее OR	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальное значение и его индекс Минимальное значение и его индекс	Специальные типы для этих функций

1.3. Производные типы данных и передача упакованных данных

При разработке параллельных программ иногда возникает потребность передавать данные разных типов (например, структуры) или данные, расположенные в несмежных областях памяти (части массивов, не образующих непрерывную последовательность элементов). MPI предоставляет два механизма эффективной пересылки данных в этих случаях:

- создание производных типов для использования в коммуникационных операциях вместо предопределенных типов MPI;
- пересылка упакованных данных (процесс-отправитель упаковывает пересылаемые данные перед их отправкой, а процесс-получатель распаковывает их после получения).

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме коммуникационных. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (opaque) объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как отображение (карта) типа:

Typemap = {(type0, disp0), ..., (typen-1, dispn-1)

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию. Отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена. Этот буфер будет содержать n элементов, а i-й элемент будет иметь адрес buf+desp и иметь базовый тип type. Стандартные типы MPI имеют предопределенные отображения типов. Например, MPI_INT имеет отображение {(int,0)}.

Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.
- Новый производный тип регистрируется вызовом функции MPI_Type_commit. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при

конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.

- Когда производный тип становится ненужным, он уничтожается функцией MPI_Type_free.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

- Протяженность типа определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой MPI_Type_extent).
- Размер типа определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой MPI_Type_size).

Для простых типов протяженность и размер совпадают.

Краткая сводка по функциям, необходимым для работы с производными типами данных:

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)

int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_hvector(int count, int blocklength,
MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_indexed(int count, int *array_of_blocklengths,
int *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)

int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype
*array_of_types, MPI_Datatype *newtype)

int MPI_Type_commit(MPI_Datatype *datatype)

int MPI_Type_free(MPI_Datatype *datatype)
```

Для посылки элементов разного типа из нескольких областей памяти их следует предварительно запаковать в один массив, последовательно обращаясь к функции упаковки MPI_Pack. При первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера. Для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова.

Упакованный буфер пересылается любыми коммуникационными операциями с указанием типа MPI_PACKED и коммуникатора comm, который использовался при обращениях к функции MPI_Pack.

Полученное упакованное сообщение распаковывается в различные массивы или переменные. Это реализуется последовательными вызовами функции распаковки MPI_Unpack с указанием числа элементов, которое следует извлечь при каждом вызове, и с передачей значения position, возвращенного предыдущим вызовом. При первом вызове функции параметр position следует установить в 0. В общем случае, при первом обращении должно быть установлено то значение параметра position, которое было использовано при первом обращении к функции упаковки данных. Очевидно, что для правильной распаковки данных очередность извлечения данных должна быть той же самой, как и упаковки.

Функция MPI_Pack_size помогает определить размер буфера, необходимый для упаковки некоторого количества данных типа datatype.

Краткая сводка по функциям передачи упакованных данных:

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outsize, int *position, MPI_Comm comm)

int MPI_Unpack(void* inbuf, int insize, int *position,
void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm
comm)

int MPI_Pack_size(int incount, MPI_Datatype datatype,
MPI_Comm comm, int *size)
```

1.4 Работа с группами и коммуникаторами

Часто в приложениях возникает потребность ограничить область коммуникаций некоторым набором процессов, которые составляют подмножество исходного набора. Для выполнения каких-либо коллективных операций внутри этого подмножества из них должна быть сформирована своя область связи, описываемая своим коммуникатором. Для решения таких задач MPI поддерживает два взаимосвязанных механизма. Во-первых, имеется набор функций для работы с группами

процессов как упорядоченными множествами, и, во-вторых, набор функций для работы с коммуникаторами для создания новых коммуникаторов как описателей новых областей связи.

Группа представляет собой упорядоченное множество процессов. Каждый процесс идентифицируется переменной целого типа. Идентификаторы процессов образуют непрерывный ряд, начинающийся с 0. В MPI вводится специальный тип данных MPI_Group и набор функций для работы с переменными и константами этого типа. Существует две предопределенных группы:

- MPI_GROUP_EMPTY – группа, не содержащая ни одного процесса;
- MPI_GROUP_NULL – значение, возвращаемое в случае, когда группа не может быть создана.

Созданная группа не может быть модифицирована (расширена или усечена), может быть только создана новая группа. Интересно отметить, что при инициализации MPI не создается группы, соответствующей коммуникатору MPI_COMM_WORLD. Она должна создаваться специальной функцией явным образом.

Коммуникатор представляет собой скрытый объект с некоторым набором атрибутов, а также правилами его создания, использования и уничтожения. Коммуникатор описывает некоторую область связи. Одной и той же области связи может соответствовать несколько коммуникаторов, но даже в этом случае они не являются тождественными и не могут участвовать во взаимном обмене сообщениями. Если данные посылаются через один коммуникатор, процесс-получатель может получить их только через тот же самый коммуникатор.

При инициализации MPI создается два предопределенных коммуникатора:

- MPI_COMM_WORLD – описывает область связи, содержащую все процессы;
- MPI_COMM_SELF – описывает область связи, состоящую из одного процесса.

Краткая сводка по функциям работы с группами:

```
MPI_Group_size(MPI_Group group, int *size)

MPI_Group_rank(MPI_Group group, int *rank)

MPI_Group_translate_ranks(MPI_Group group1, int n, int
*ranks1, MPI_Group group2, int *ranks2)

MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_Group_union(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
```

```
MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)

MPI_Group_incl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup)
MPI_Group_excl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup)

MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)
MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
MPI_Group *newgroup)

MPI_Group_free(MPI_Group *group)
```

Функции работы с коммуникаторами разделяются на функции доступа к коммуникаторам и функции создания коммуникаторов. Функции доступа являются локальными и не требуют коммуникаций, в отличие от функций создания, которые являются коллективными и могут потребовать межпроцессорных коммуникаций.

Краткая сводка по функциям работы с коммуникаторами:

```
MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2, int *result)

MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_Comm_create(MPI_Comm comm, MPI_Group group,
MPI_Comm *newcomm)

MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *newcomm)

MPI_Comm_free(MPI_Comm *comm)
```

1.5 Виртуальные топологии процессов

Топология процессов является одним из необязательных атрибутов коммуникатора. Такой атрибут может быть присвоен только intra-коммуникатору. По умолчанию предполагается линейная топология, в которой процессы пронумерованы в диапазоне от 0 до n-1, где n – число процессов в группе. Однако для многих задач линейная топология неадекватно отражает логику коммуникационных связей между процессами. MPI предоставляет средства для создания достаточно сложных “виртуальных” топологий в виде графов, где узлы являются процессами, а грани – каналами связи между процессами. Конечно же, следует различать логическую топологию процессов, которую позволяет формировать MPI, и физическую топологию процессоров. В идеале логическая топология процессов должна учитывать как алгоритм решения

задачи, так и физическую топологию процессоров. Для очень широкого круга задач наиболее адекватной топологией процессов является двумерная или трехмерная сетка. Такие структуры полностью определяются числом измерений и количеством процессов вдоль каждого координатного направления, а также способом раскладки процессов на координатную сетку. В MPI, как правило, используется row-major нумерация процессов, т.е. используется нумерация вдоль строки.

Обобщением линейной и матричной топологий на произвольное число измерений является декартова топология. Для создания коммуникатора с декартовой топологией используется функция MPI_Cart_create. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в отдельности можно накладывать периодические граничные условия. Таким образом, для одномерной топологии мы можем получить или линейную структуру, или кольцо в зависимости от того, какие граничные условия будут наложены. Для двумерной топологии, соответственно, либо прямоугольник, либо цилиндр, либо тор. Заметим, что не требуется специальной поддержки гиперкубовой структуры, поскольку она представляет собой n-мерный тор с двумя процессами вдоль каждого координатного направления.

Краткая сводка по функциям работы с коммуникаторами с декартовой топологией:

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)

MPI_Dims_create(int nnodes, int ndims, int *dims)

MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_Cart_get(MPI_Comm comm, int ndims, int *dims, int
*periods,
int *coords)

MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
int *coords)
MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int *rank_source, int *rank_dest)

MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
*newcomm)
```

Краткая сводка по функциям работы с коммуникаторами с топологией графов:

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

```
int MPI_Graph_create( MPI_Comm comm, int nnodes,
int *index, int *edges, int reorder, MPI_Comm *comm_graph)

MPI_Graph_neighbors_count( MPI_Comm comm, int rank,
int *neighbors_count)

MPI_Graph_neighbors( MPI_Comm comm, int rank, int max,
int *neighbors)

MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_Graph_get(MPI_Comm comm, int nnodes, int nedges,
int *index, int * edges)
```

2. Способы доступа пользователя на кластер

2.1 Удаленный доступ на кластер для компиляции и запуска расчетных программ пользователя

Пользователи операционной систем Linux могут воспользоваться стандартным ssh-клиентом. Пользователям Microsoft Windows рекомендуется использовать программу PuTTy (http://www.chiark.greenend.org.uk/~sgtatham/putty/ download.html)

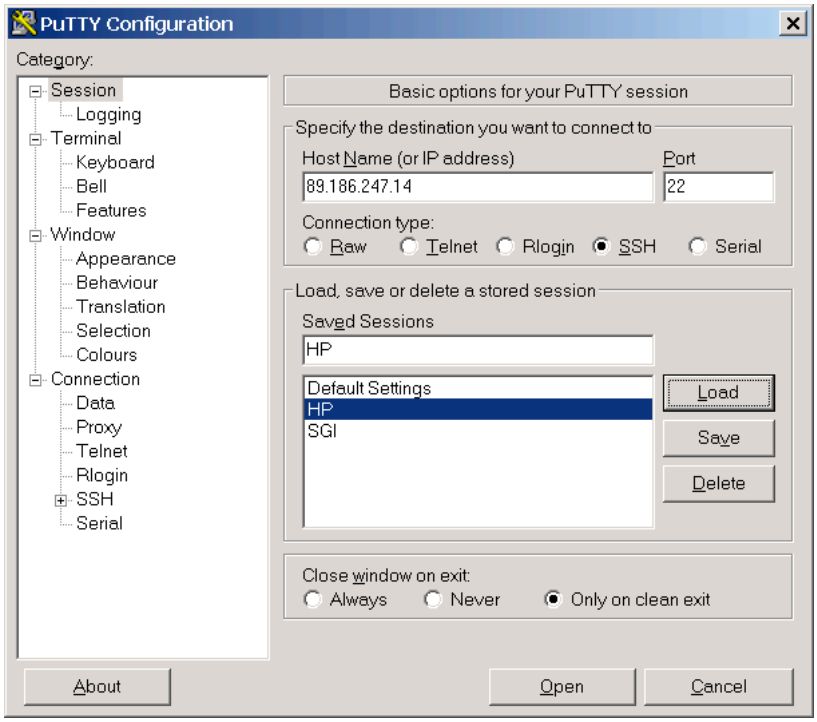

Рис. 1. Конфигурация программы PuTTy

При первом запуске программы в окне HostName набрать IP адрес кластера – 89.186.247.14. В поле Saved Sessions ввести любое удобное название (на рис.1 выбрано название «HP»).

В поле Category — Window — Translation — Character set translation on received data выбрать UTF-8 (см. рис. 2). Вернуться в окно Category — Session. Нажать кнопку «Save». Под именем HP будут сохранены настройки.

При последующих запусках PuTTy в окне конфигурации выбрать «HP» и щелкнуть кнопку “Load”. В окне Host Name появится IP адрес кластера – 89.186.247.14.

Нажать кнопку “Open”. Произойдет соединение с кластером, откроется окно терминала кластера. В этом окне сначала ввести имя пользователя, затем пароль.

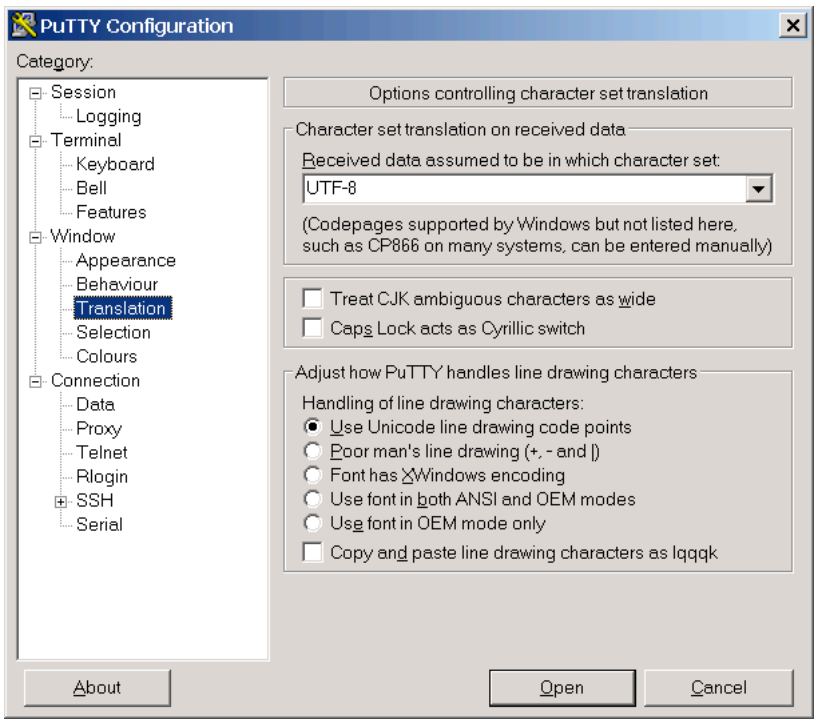

Рис. 2. Выбор таблицы кодировки символов

2.2 Удаленный доступ на кластер для копирования файлов между персональным компьютером пользователя и кластером

Сетевой файловый менеджер WinSCP может быть использован для файловых операций с удаленными и локальными файлами. Скачать программу WinSCP можно по адресу http://winscp.net.

При первом запуске программы WinSCP необходимо ввести в поле Host name IP адрес кластера – 89.186.247.14, а в поле User name имя пользователя (см. рис. 3). Поле Password можно не заполнять, его вы будете вводить при каждом следующем соединении. Введенные данные сохраняются кнопкой “Save”.

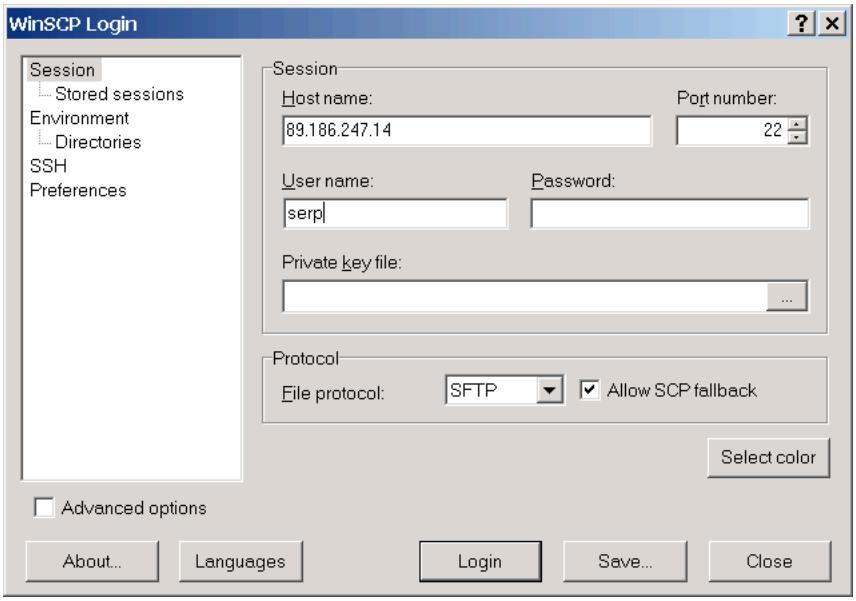

Рис. 3. Стартовое окно программы WinSCP при первом запуске

При последующих запусках программы окно WinSCP Login будет с заполненными личными данными (рис. 4).

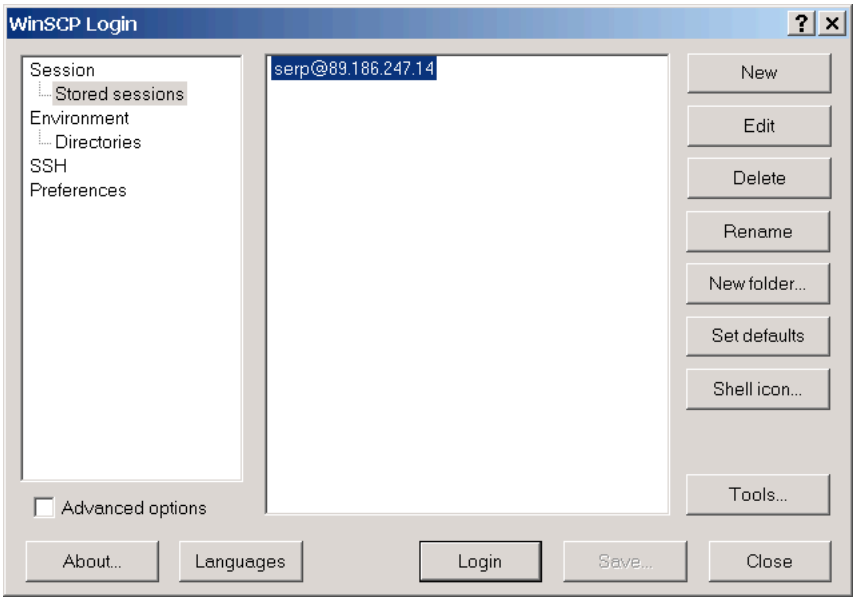

Рис. 4. Окно соединения

Соединение начинается выбором пункта “Login”. В окне Server prompt надо ввести свой пароль (рис. 5). Для соединения с удаленной ЭВМ необходимо время. Поэтому окно программы WinSCP появится с некоторой задержкой. В зависимости от варианта, выбранного при установке программы откроется окно, по внешнему виду сходное с Windows Explorer либо двухпанельное окно в стиле Total Commander. В этом окне будет отображена файловая система кластера и вы можете

копировать файлы на кластер и обратно также, как это делается в Windows. Поддерживаются операции Drag and Drop.

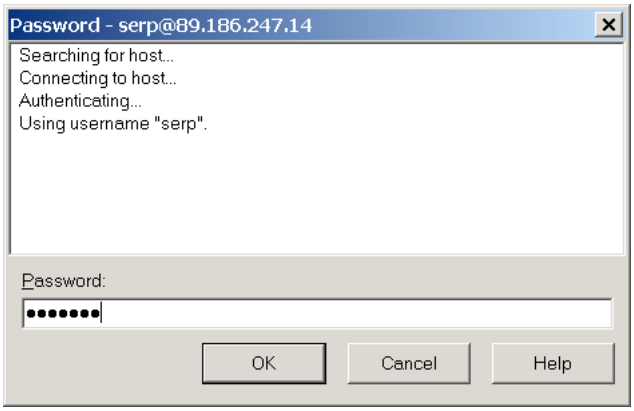

Рис. 5. Окно ввода пароля

Кроме того, программа WinSCP позволяет проводить другие операции с файлами:редактирование, переименование, удаление, изменение прав доступа и прочее.

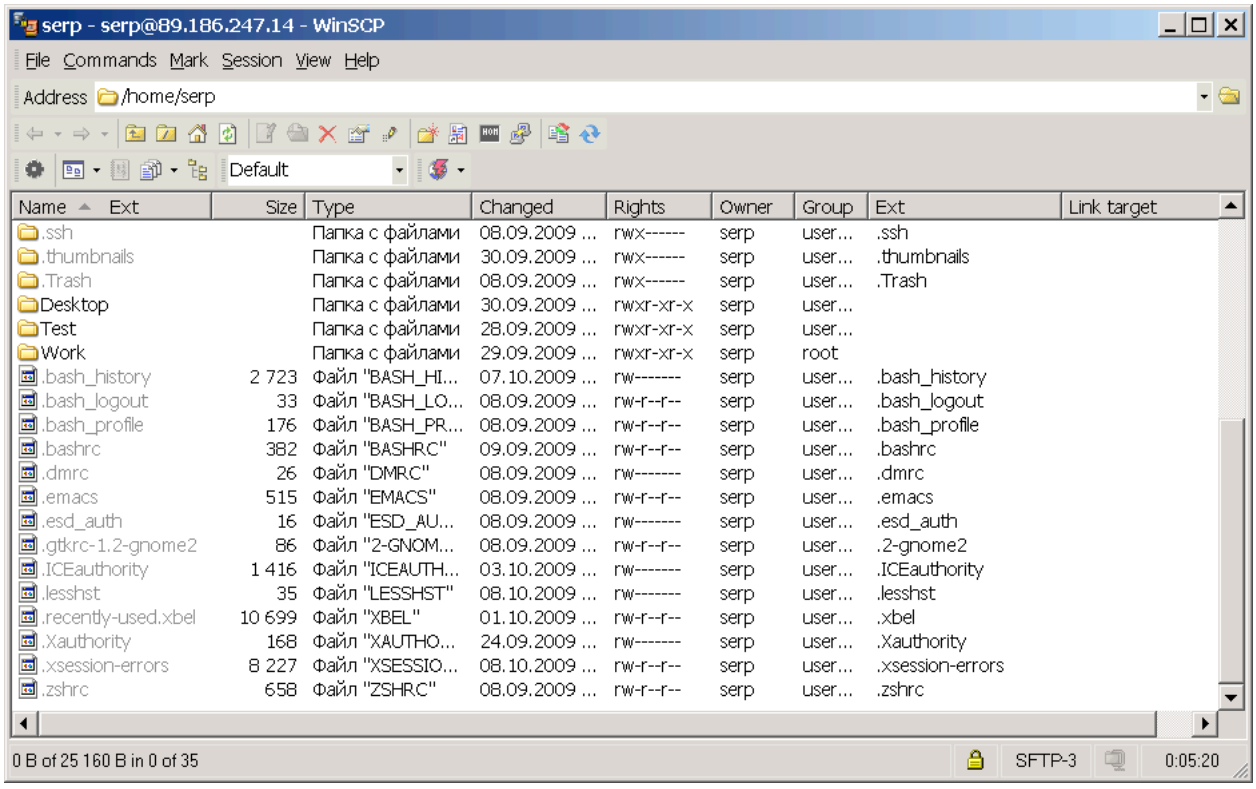

Рис. 6. Рабочее окно программы WinSCP

Завершение работы с программой и прекращение связи проводится клавишей F10 или выбором в меню пункта Commands/Quit. Требуется подтвердить окончание работы в окне завершения.

3. Основные операции выполняемые пользователем на кластере

3.1 Редактирование текста программ пользователя

Процесс разработки программного обеспечения, независимо от размеров и предназначения программы содержит этап редактирования исходных текстов программы. Конечно, вы можете изменять свои программы на рабочем компьютере, а потом копировать их на кластер. Также вы можете редактировать файлы прямо на кластере. Для этого на кластере имеется несколько текстовых редакторов разной степени удобства: vim, ed, emacs, joe. Наиболее простым для освоения, на наш взгляд, является редактор, встроенный в оболочку MidnightCommander.

Для запуска этой программы используется команда mc.

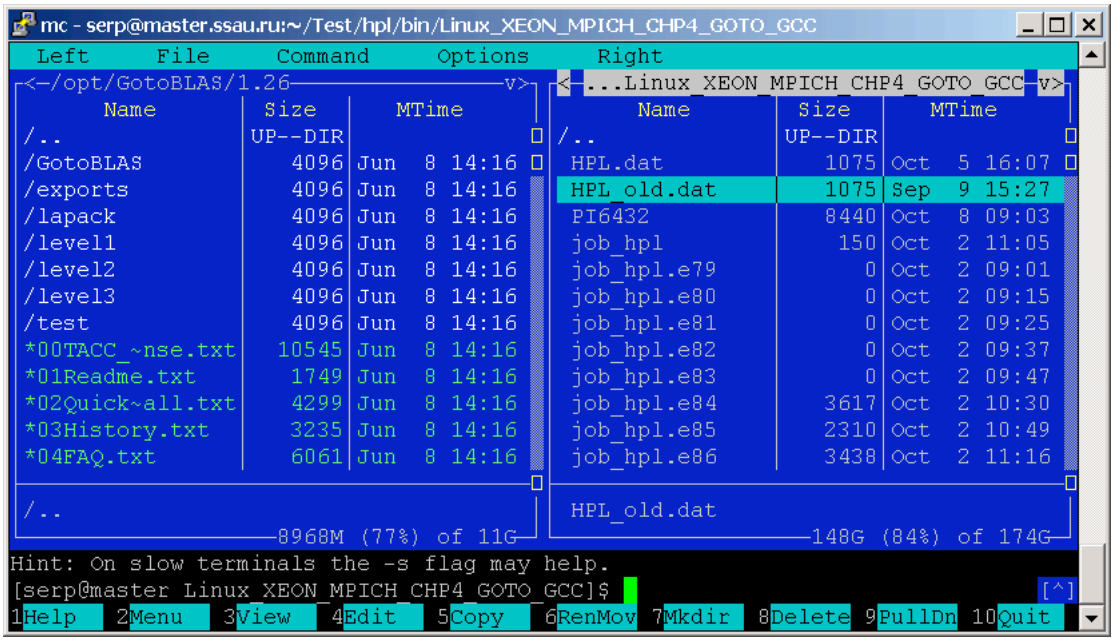

Рис. 7 Окно оболочки MidnightCommander

Midnight Commander — это файловый менеджер с текстовым интерфейсом. Его предназначение — упростить основные действия пользователя, связанные с управлением файлами. Принцип работы Midnight Commander такой же, как и у Far Manager, TotalCommander, или Norton Commander. Экран состоит из двух панелей, в которых отображается список файлов и каталогов в выбранных каталогах, и пользователь может выполнять некоторый набор действий над этими файлами. В нижней части экрана расположена командная строка и панель горячих клавиш F1-F10. Можно вызвать верхнее меню, нажав клавишу F9. Одна из панелей всегда является активной, а в ней курсор установлен на

активный файл. Пользователь может выполнять действия либо с активным файлом или каталогом, либо групповые операции со всеми объектами активной панели. Также доступны некоторые общие операции: поиск файлов, помощь по работе с MC, выполнение команд операционной системы и т.д.

Для редактирования файла необходимо клавишами управления курсором выбрать нужный файл, и нажать клавишу F4. Запустится редактор текстовых файлов, выйти из которогоможно, нажав клавишу F10 либо Esc. Чтобы сохранить изменения, необходимо нажать клавишу F2, либо выбрать нужный вариант при выходе из редактора.

Для создания нового файла нужно нажать Shift+F4 (при нажатой клавише Shift нажмите клавишу F4). Откроется окно редактора, и при сохранении изменений программа предложит ввести имя сохраняемого файла.

3.2 Компиляция программ пользователя

Для компиляции и сборки параллельных программ используются следующие утилиты:

- *mpicc* — для программ написанных на языке программирования C;
- *mpixx (mpiCC)* — для программ написанных на языке программирования C++;
- *mpif77 и mpif90* — для программ написанных на языке программирования Fortran.

Синтаксис данных утилит во многом похож на синтаксис компилятора ***gcc***, более полная информация о синтаксисе доступна по команде **`имя_утилиты --help`** (напр., **`mpif77 –help`**).

Например, для обработки программы myprog.c можно выполнить команду

```
mpicc –o myprog –lm myprog.c
```

Ключ **`-o`** указывает, что результат компилирования должен быть помещен в файл **`myprog`**. Если мы просто выполним команду

```
mpicc  –lm myprog.c
```

то в текущем каталоге появится исполняемый файл **`a.out`**.

Если программа представлена не одним файлом, а несколькими исходными и заголовочными файлами, то для сборки можно использовать Makefile или shell-скрипт.

Т.к. на кластере установлено несколько реализаций MPI, то пользователь должен контролировать контекст выполнения параллельных программ. Это можно делать с помощью команды ***switcher***.

Чтобы посмотреть установленные контексты выполнения MPI программ наберите команду.

```
switcher mpi --list
```

Используя команду

```
which mpicc
```

можно контекст выполнения MPI на управляющем узле и на вычислительных узлах.

```
cexec which mpicc
```

Команда ***switcher*** позволяет переключать контекст выполнения MPI программ, задавать контекст по умолчанию и т.д. (см. **`man switcher`**).

3.3 Запуск расчетных задач пользователя

Запуск задачи на кластере выполняется с использованием системы пакетного запуска ***PBS Torque***. Это гибкий и удобный способ работы на кластере, особенно в многопользовательском режиме.

Вообще, использованию Torque посвящены подробные руководства, здесь же мы перечислим только самые простые варианты использования.

Запуск задачи осуществляется следующим образом:

```
qsub <имя_программы>
```

или

```
qsub <имя_скрипта>
```

Сценарий, или скрипт не обязательно должен быть исполняемым, это может быть простой текстовый файл, содержащий размещенные построчно команды в формате обычной командной строки (см. пп. 4.1 и 4.2). Кроме исполняемых команд скрипт может содержать специальные директивы ***PBS Torque***, которые позволяют описать необходимые для выполнения задачи ресурсы: кол-во процессоров, память, время, политику многозадачного режима – и многие другие параметры с различной степенью детализации.

Torque ставит запущенную задачу в очередь, отыскивает свободные процессоры (вычислительные узлы) и запускает задачу. При этом пользователь может закрыть свою терминальную сессию – задача будет выполняться в фоновом режиме. Пользователь может просмотреть очередь при помощи команды

```
qstat
```

Команда отобразит состояние очереди задач. При этом будут отображаться задачи запущенные только этим пользователем (рис. 8).

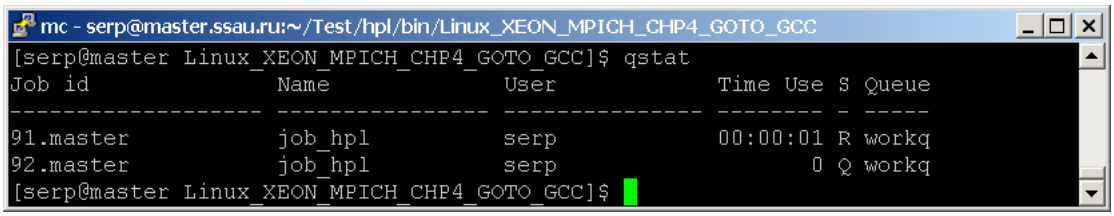

Рис. 8 Очередь задач пользователя

Здесь показана очередь из двух заданий с номерами 91 и 92, которые запущены пользователем `serp`, имя очереди `workq`, имя задачи `job_hpl`.

С помощью команды

```
qstat -n
```

можно получить более полную информацию о количестве заказанных вычислительных узлов (колонка NDS), максимальном времени выполнения (Time), количество задач (TSK) и состоянии (S):

- Q – в очереди,
- R – выполняется.

Удалить задание из очереди можно командой

```
qdel <номер_задания>
```

4. Лабораторные работы пользователя на кластере

4.1 Определение латентности и пропускной способности MPI коммуникаций кластера (запуск теста PMB)

Тест PMB (Pallas MPI Benchmark) позволяет исследовать характеристики вычислительной сети кластера для различных коммуникаций MPI установленных на кластерной системе: коммуникаций точка-точка, коллективных операций.

Необходимые файлы находятся в каталоге `/opt/benchmarks`.

Используя команду

```
tar xvf PMB2.2.tar
```

распакуйте и установите в свою домашнюю директорию дерево исходных кодов теста PMB.

В файле `make_i86` укажите следующее (Рис. 9)

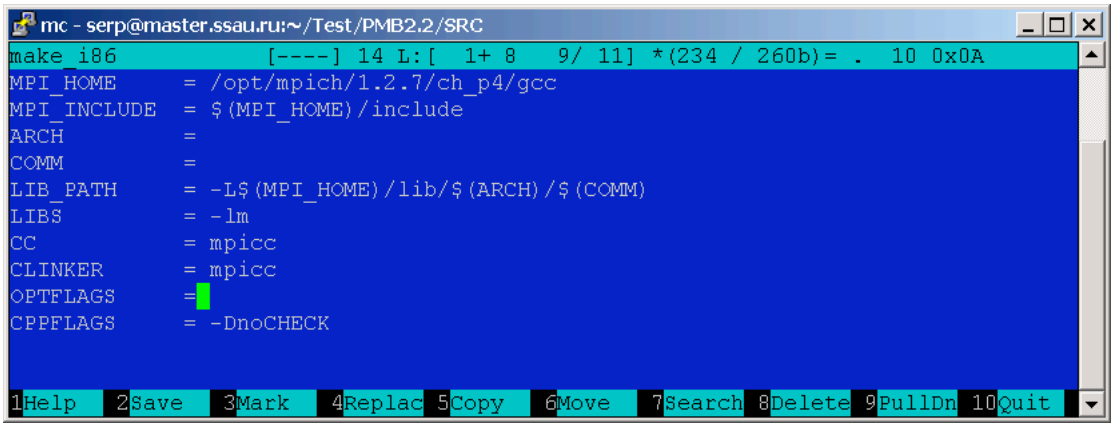

Рис. 9 Содержимое файла make_i86

Здесь:

- MPI_HOME - домашняя директория mpich
- CC - команда вызова компиляции C-программы с использованием mpich
- CLINKER - команда вызова сборщика C-программы с использованием mpich

Затем внесите в файл Makefile следующие изменения (Рис. 10):

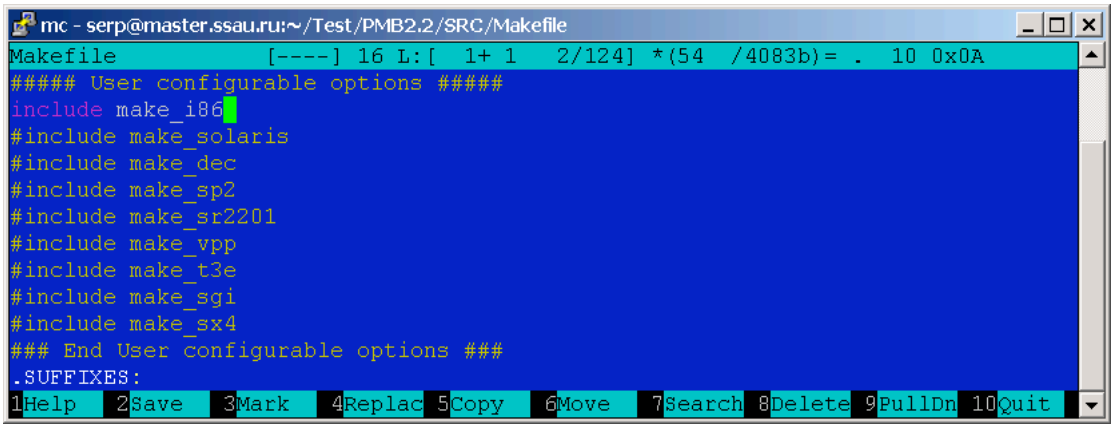

Рис. 10 Содержимое файла Makefile

Используя команду make соберите тест.

Создайте PBS задание для запуска теста PMB с требованием к ресурсам,, кластера: время работы программы = 30 минут, число узлов = 2, число вычислительных ядер = 8.

Для запуска теста используйте команду

`/opt/mpiexec/mpiexec`

с опцией позволяющей запускать на каждом выделенном системой пакетной обработки вычислительном узле не более одного процесса (Рис. 11).

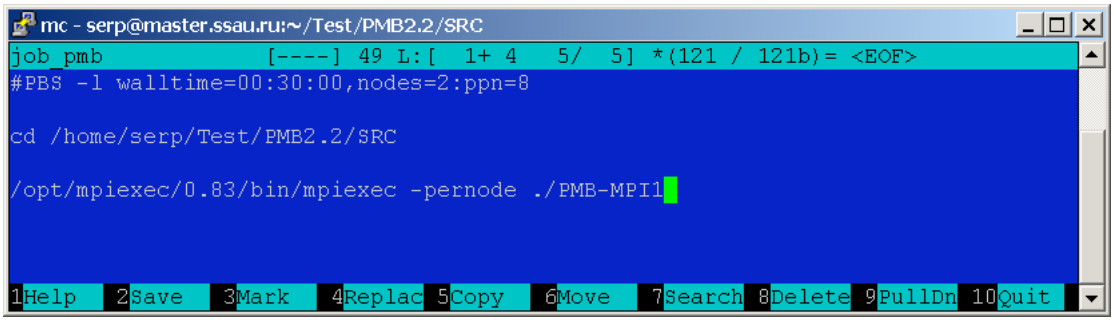

Рис. 11 Содержимое скрипта job_pmb

Поставьте задание в очередь на выполнение командой

`qsub job_pmb`

После окончания задания сравните результаты по параметрам системные задержки на передачу данных (латентность), пропускная способность, для разного типа MP1 коммуникаций.

Например, латентность для «точечных» операций можно оценить на тесте PingPong для пакета 0-ой длины, пропускную способность - на пакете максимальной длины (Рис. 12).

4.2 Определение производительности и эффективности кластера (запуск теста HPL)

Тест HPL (High Performance Linpack) позволяет исследовать характеристики вычислительного кластера для теста HPL производительность, эффективность.

Для выполнения задания вам необходимо установить переменную окружения библиотеки *GotoBLAS*. Для этого отредактируйте файл `~/.bashrc` добавив переменную окружения

```
export OMP_NUM_THREADS=1
```

Для активизации установки откройте новый терминал или выполните команду bash.

Необходимые для теста файлы находятся в `/opt/benchmarks`.

Используя команду

```
tar xvf hpl.tar
```

распакуйте и установите в свою домашнюю директорию дерево исходных кодов теста HPL.

В конфигурационный файл `Make.Linux_XEON_MPICH_CHP4` внесите следующие изменения:

```
APRH = Linux_XEON_MPICH_CHP4
TOPdir = $(HOME)/hpl
CC = mpicc
MPdir = путь к MPI (см. which mpicc)
LAdir = /opt/GotoBLAS/1.26
```

Используя команду

```
make arch = Linux_XEOM_MP1CH_CHP4
```

соберите тест.

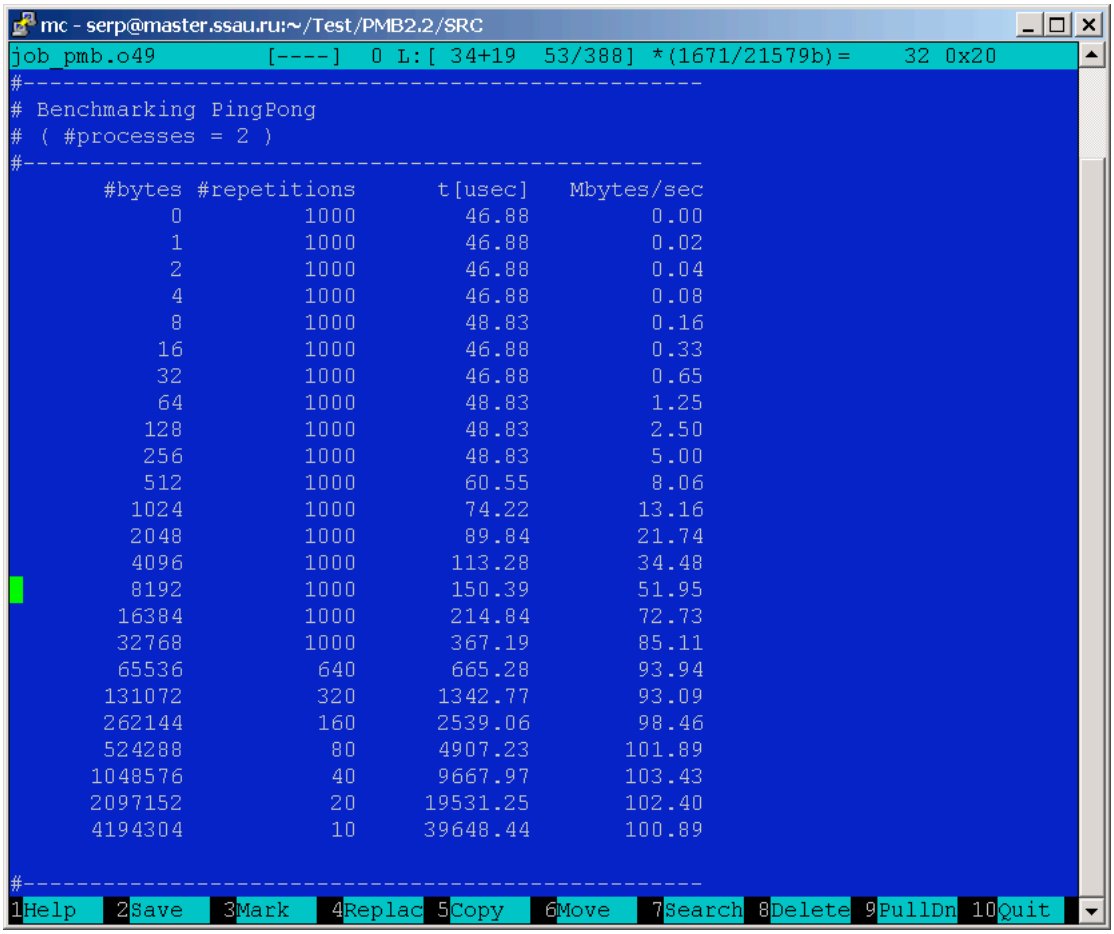

Рис. 12 Фрагмент результатов теста PMB

Подготовьте исходные данные для запуска теста HPL (Рис. 13) исходя из следующего:

- решетка процессоров Ps x Qs для запуска на 14-ти вычислительных узлах (8 процессоров на каждом)
- размер блока матрицы NBs = 220

Рассчитайте размер расчетной матрицы теста HPL исходя из следующего:

Расчетная матрица должна размещаться в оперативной памяти и должна занимать 80% суммарной оперативной памяти. Таким образом, размер расчетной матрицы может быть получен по следующей формуле:

Ns = sqrt[(0,8 x (1024 x 1024 x 1024 x n x m)) / 8]

где

n =14 - число узлов принимающих участие в вычислениях

m = 8 - объем оперативной памяти одного узла (Гб).

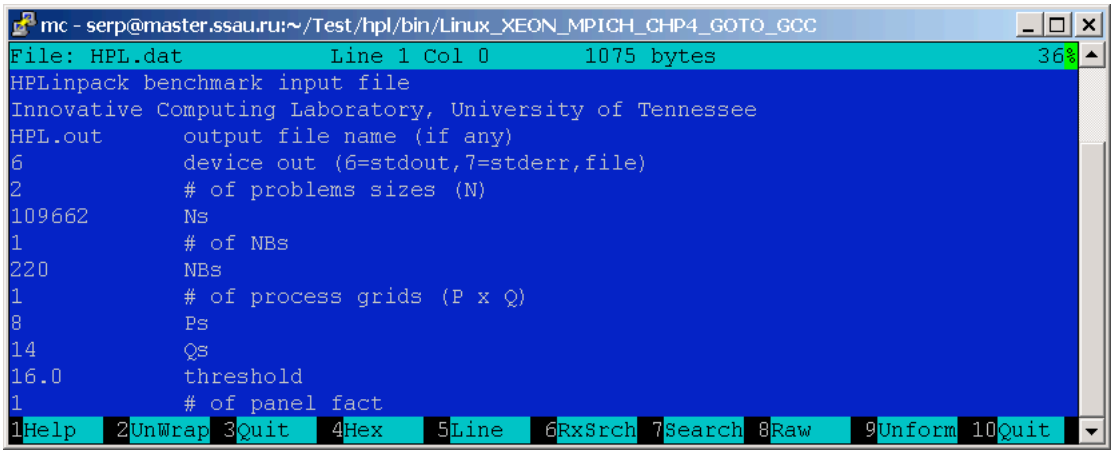

Рис. 13 Фрагмент файла входных данных для теста HPL

Создайте PBS задание для запуска теста PMB с требованием к ресурсам кластера: время работы программы - 50 минут, число вычислительных узлов - 14, число процессорных ядер на каждом узле - 8 (Рис. 14)

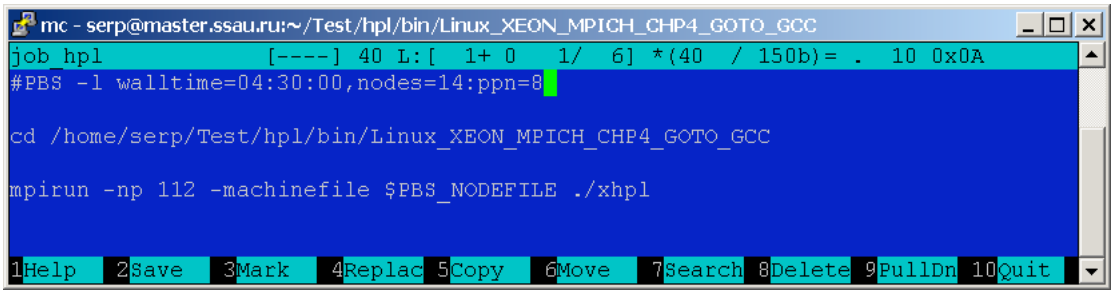

Рис. 14 Содержимое скрипта job_hpl

Для запуска теста используйте команду /opt/mpiexec/mpirun
Поставьте задание в очередь на выполнение командой

qsub job_hpl

С помощью команды

qstat

проверьте статус вашего задания.
С помощью команды

qstat -f

проверьте вычислительные узлы которые система пакетной обработки выделила для выполнения вашего задания.

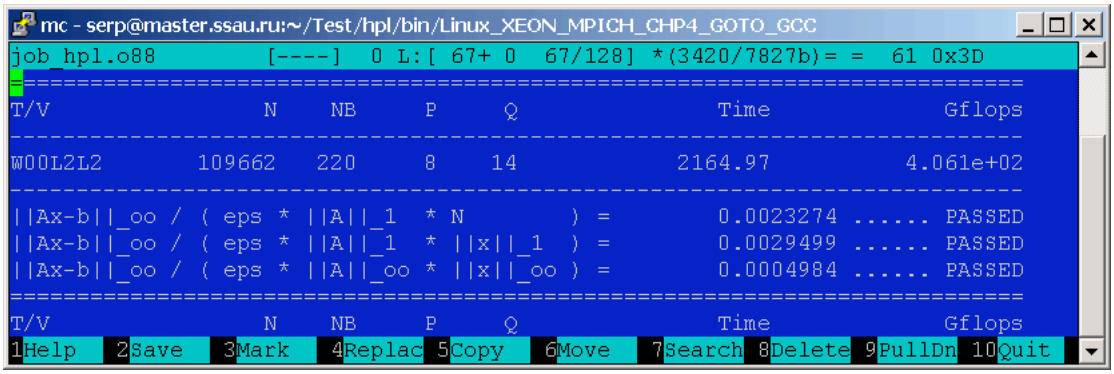

Рис. 15 Фрагмент результатов теста HPL

По результатам теста HPL можно рассчитать эффективность кластера по формуле

E = PHPL / PP ,

где

PHPL - производительность полученная на тесте HPL,
PP - пиковая производительность полученная как произведение

PP = n x nc x F x k

где

n = 14 - число узлов принимающих участие в вычислениях,
nc =8 - число вычислительных ядер на одном узле,
F = 2.33 - тактовая частота одного вычислительного ядра,
k - число операций с плавающей точкой выполняемых одним вычислительным ядром за такт (для процессоров Intel Xeon E67** k=4).

Приложение 1. Обзор необходимых команд Linux.

Ниже приводятся некоторые наиболее употребляемые команды Linux. Большинство этих команд можно выполнить на управляющем узле с помощью MidnightCommander. Однако на вычислительных узлах MidnightCommander, как правило, отсутствует.

Чтобы получить более полную информацию по любой отдельной команде command, нужно ввести

```
man command
```

Выход из описания команды производится при нажатии клавиши «q».

Работа с каталогами

`pwd` – показывает название текущей директории;

`cd dir` – устанавливает текущим каталогом каталог с именем dir, вызов команды cd без параметров возвращает в домашний каталог /home/username ($HOME);

`mkdir subdir` – создает новый подкаталог с именем subdir;

`rmdir subdir` – удаляет пустой подкаталог с именем subdir;

`ls` – показывает список файлов и подкаталогов текущей директории,

`ls dir` – показывает список файлов и подкаталогов каталога dir;

`ls –A` - показывает все файлы, в том числе и скрытые;

`ls –l` - показывает атрибуты (владелец, разрешение на доступ, размер файла и время последней модификации);

`mv oldname newname` - изменяет имя подкаталога или перемещает его;

`cp -R dirname destination` - копирует подкаталог dirname в другое место destination.

Работа с файлами

file filename(s) - определяет тип файла (например, ASCII , JPEG image data и др.);

cat filename(s) - показывает содержание файлов (используется только для текстовых файлов!);

more filename(s) - действует так же, как и cat, но позволяет листать страницы;

less filename(s) – улучшенный вариант команды more;

head filename - показывает первые десять строк файла filename;

tail filename - показывает последние десять строк файла filename;

wc filename(s) - показывает число строк, слов и байт для указанного файла;

rm filename(s) - уничтожает файлы или директории, для рекурсивного удаления следует использовать rm с ключом -rf.

cp filename newname - создает копии файлов с новыми именами;

cp filename(s) dir- копирует один или более файлов в другой каталог;

mv oldname newname - изменяет имя файла или каталога;

mv filename(s) dir - перемещает один или более файлов в другой каталог;

find dir –name filename - пытается локализовать файл (подкаталог) filename рекурсивно в подкаталоге dir.

Другие полезные команды

passwd - изменяет пароль пользователя системы Linux; требует подтверждения старого;

who – показывает, кто в настоящее время работает в сети;

finger – дает более подробную информацию о пользователях сети;

write – позволяет послать сообщение пользователю, работающему в сети в данное время;

top - отображает информацию о процессах, использующих процессоры узла;

ps –U user_name - показывает номера процессов(pid), инициированных пользователем user_name;

kill xxxxx – досрочно завершает работы процесса с номером xxxxx;

killall proc_name - досрочно завершает работу процесса proc_name;

date - отображает дату и время;

cal – показывает календарь.

exit – выйти из терминала

clear – очистить окно терминала

du dir – показывает занятое место в директории dir

Приложение 2. Примеры PBS скриптов

Подробно о возможностях PBS Torque можно узнать из руководства пользователя.

Ниже приведены несколько примеров использования Torque.

#PBS -o $DIR/stdout.log

Определяет имя файла, в который будет перенаправлен стандартный поток stdout

#PBS -e $DIR/stderr.log

Определяет имя файла, в который будет перенаправлен стандартный поток stderr

#PBS -l nodes=8:ppn=2:cpp=1

Определяет какое количество узлов и процессоров на них необходимо задействовать.

nodes - количество узлов

ppn - число процессоров на узле

cpp - число процессов на процессоре

#PBS -l walltime=20:00:00

Определяет максимальное время счета задания

#PBS -l mem=1000mb

Определяет количество необходимой оперативной памяти

```
cat $PBS_NODEFILE | grep -v master | sort | uniq -c |
awk '{printf "%s:%s\n", $2, $1}' >
$PBS_O_WORKDIR/temp.tmp
```

Составляет список узлов в необходимом формате, на которых будет запущена задача и записывает их в файл temp.tmp

```
cd $PBS_O_WORKDIR
/usr/bin/mpirun -m temp.tmp –np 100 ./a.out
```

Запускает на узлах указанных в файле temp.tmp задачу 100 раз.

Пример скрипта:

```
#PBS -o $DIR/stdout.log
#PBS -e $DIR/stderr.log
#PBS -l nodes=50:ppn=2
#PBS -l walltime=20:00:00
#PBS -l mem=1000mb
```

```
cat $PBS_NODEFILE | grep -v master | sort | uniq -c |
awk '{printf "%s:%s\n", $2, $1}' >
$PBS_O_WORKDIR/script1.temp.sh.mf
cd $PBS_O_WORKDIR
/usr/bin/mpirun -m script1.temp.sh.mf -np 100 ./a.out
```

Здесь будет запущена параллельная программа a.out на 50 узлах, с каждого узла будет использоваться 2 процессора. Файл вывода стандартного потока stdout — stdout.log, стандартного потока stderr — stderr.log.

$DIR содержит путь к файлам stdout.log и stderr.log, например может принимать значение /home/user_name. Под задачу отведено 20 часов. Необходимое количество памяти 1000 мегабайт.

Для запуска последовательной программы first можно использовать следующий скрипт:

```
#PBS -o $DIR/stdout.log
#PBS -e $DIR/stderr.log
#PBS -l walltime=10:00
#PBS -l mem=100mb
./first
```

При запуске программы через команду qsub заданию присваивается уникальный целочисленный идентификатор.

qdel – утилита для удаления задачи.

В случае, если задача уже запущена, процесс ее работы будет прерван. Синтаксис данной утилиты следующий:

```
qdel [-W время задержки]идентификатор задачи
```

Выполнение такой команды удалит задачи с заданными идентификаторами через указанное время. Если часть вычислительных узлов, на которых выполнялась задача, недоступны, то принудительно удалить ее с сервера можно путем добавления ключа -p.