Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Дальневосточный государственный университет

М.А. КНЯЗЕВА
Л.А. МОЛЧАНОВА
Г.В ТАРАСОВ

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методические указания и задания для
студентов специальности 351500
Математическое обеспечение и администрирование информационных систем

Владивосток
Издательство Дальневосточного университета
2007

ББК

Рецензенты:
Е.А.Нурминский, д.ф.-м.н. (ИМКН ДВГУ, РАН);

Князева М.А, Молчанова Л.А., Тарасов Г.В. Параллельное программирование. Методические указания и задания для студентов специальности 351500. - Владивосток: Изд-во Дальневост. ун-та, 2006. - 61 с.

Методические указания разработаны для студентов Института математики и компьютерных наук ДВГУ. Пособие представляет собой введение в методы программирования для параллельных ЭВМ. Вопросы распараллеливания конкретных алгоритмов рассматриваются на примерах на языке программирования Си.

Для студентов математических специальностей.

ББК

@ Князева М.А., 2007
@ Молчанова Л.А., 2007
@ Тарасов Г.В. , 2007

@ ИМКН ДВГУ, 2007

Содержание

Часть 1. Функции системы программирования MPI 4
Введение 4
Общие функции MPI 5
Сообщения 6
Коммуникаторы 7
Прием/передача сообщений между отдельными процес 8
Совмещение приема и передачи сообщений 15
Коллективные взаимодействия процессов 15
Синхронизация процессов 18
Работа с группами процессов 18
Часть 2. Примеры использования MPI 19
Задание 1. 19
Задание 2. 23
Задание 3. 25
Задание 4. 29
Задание 5. 31
Задание 6. 34
Задание 7. 36
Часть 3. Правила работы в среде MPI
1. Архитектура кластера 40
1.1. Командный интерфейс 40
2. Основные команды Linux 41
2.1. Команда ls 41
2.2. Команда cd 42
2.3. Команда md 42
2.4. Команда less 42
2.5. Команда vim 42
2.6. Команда cp 43
2.7. Команда mv 43
2.8. Команда rm 43
2.9. Команда passwd 43
3. Файловый интерфейс 43
4. Компиляция параллельных программ 43
5. Запуск параллельных программ 45
6. Рекомендации по отладке программ 47
6.1. Применение printf() 47
6.2. Утилита jumpshot 47
Задания 53
Литература 56
Приложение 1 58
Приложение 2 59

Часть 1. Функции системы программирования MPI

Введение

Широкое распространение компьютеров с распределенной памятью определило и появление соответствующих систем программирования. Как правило, в таких системах отсутствует единое адресное пространство, и для обмена данными между параллельными процессами используется явная передача сообщений через коммуникационную среду. Отдельные процессы описываются с помощью традиционных языков программирования, а для организации их взаимодействия вводятся дополнительные функции. По этой причине практически все системы программирования, основанные на явной передаче сообщений, существуют не в виде новых языков, а в виде интерфейсов и библиотек.

К настоящему времени примеров известных систем программирования на основе передачи сообщений накопилось довольно много: Shmem, Linda, PVM, MPI и др.

MPI представляет собой набор утилит и библиотечных функций (для языка C/C++, Fortran), позволяющих создавать и запускать приложения, работающие на параллельных вычислительных установках самой различной природы.

Все дополнительные объекты: имена функций, константы, предопределенные типы данных и т. п., используемые в MPI, имеют префикс **MPI_.** Например, функция посылки сообщения от одного процесса другому имеет имя **MPI_Send**. Все описания интерфейса **MPI** собраны в файле mpi.h, поэтому в начале MPI-программы должна стоять директива #inciude <mpi.h>.

MPI-программа - это множество параллельных взаимодействующих процессов. Все процессы порождаются один раз, образуя параллельную часть программы. В ходе выполнения MPI-программы порождение дополнительных процессов или уничтожение существующих не допускается.

Каждый процесс работает в своем адресном пространстве, никаких общих переменных или данных в **MPI** нет. Основным способом взаимодействия между процессами является явная посылка сообщений.

Для локализации взаимодействия параллельных процессов программы можно создавать *группы процессов*, предоставляя им отдельную среду для общения - *коммуникатор*. Состав образуемых групп произволен. Группы могут полностью входить одна в другую, не пересекаться или пересекаться частично. При старте программы всегда считается, что все порожденные процессы работают в рамках всеобъемлющего коммуникатора, имеющего предопределенное имя **MPI_COMM_WORLD**. Этот коммуникатор существует всегда и служит для взаимодействия всех процессов MPI-программы.

Каждый процесс MPI-программы имеет уникальный атрибут *номер процесса*, который является целым неотрицательным числом. С помощью этого атрибута происходит значительная часть взаимодействия процессов между собой. Ясно, что в одном и том же коммуникаторе все процессы имеют различные номера. Но поскольку процесс может одновременно входить в разные коммуникаторы, то его номер в одном коммуникаторе может отличаться от его номера в другом. Отсюда *два основных атрибута процесса: коммуникатор и номер в коммуникаторе*.

Если группа содержит n процессов, то номер любого процесса в данной группе лежит в пределах от 0 до n - 1.

Основным способом общения процессов между собой является посылка сообщений. *Сообщение* - это набор данных некоторого типа. Каждое сообщение имеет несколько атрибутов, в частности, номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и др. Одним из важных атрибутов сообщения является его идентификатор или тэг. По идентификатору процесс, принимающий сообщение, например, может различить два сообщения, пришедшие к нему от одного и того же процесса. Сам идентификатор сообщения является целым неотрицательным числом, лежащим в диапазоне от 0 до 32767. Для работы с атрибутами сообщений введена структура **MPI_Status**, поля которой дают доступ к значениям атрибутов.

На практике сообщение чаще всего является набором однотипных данных, расположенных подряд друг за другом в некотором буфере. Такое сообщение может состоять, например, из двухсот целых чисел, которые пользователь разместил в соответствующем целочисленном векторе. Это типичная ситуация, на нее ориентировано большинство функций **MPI**, однако такая ситуация имеет, по крайней мере, два ограничения. Во-первых, иногда необходимо составить сообщение из разнотипных данных. Конечно же, можно отдельным сообщением послать количество вещественных чисел, содержащихся в последующем сообщении, но это может быть и неудобно программисту, и не столь эффективно. Во-вторых, не всегда посылаемые данные занимают непрерывную область в памяти. Если в Fortran элементы столбцов матрицы расположены в памяти друг за другом, то элементы строк уже идут с некоторым шагом. Чтобы послать строку, данные нужно сначала упаковать, передать, а затем вновь распаковать.

Чтобы снять указанные ограничения, в **MPI** предусмотрен механизм для введения производных типов данных. Описав состав и схему размещения в памяти посылаемых данных, пользователь в дальнейшем работает с такими типами так же, как и со стандартными типами данных **MPI**.

Общие функции MPI

Имена всех функций, типов данных, констант и т.д., относящихся к **MPI**-библиотеке, начинаются с префикса **MPI_** и описаны в заголовочном файле mpi.h. Все функции (за исключением **MPI_Wtime** и **MPI_Wtick**) имеют тип возвращаемого значения int и возвращают код ошибки или **MPI_SUCCESS** в случае успеха. Однако в случае ошибки перед возратом из вызвавшей ее функции вызывается стандартный обработчик ошибки, который терминирует программу. При описании функций используется слово OUT для обозначения выходных параметров, через которые функция возвращает результаты.

До первого вызова любой **MPI**-функции необходимо вызывать функцию **MPI_Init**. Ее прототип:

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

где argc, argv - указатели на число аргументов программы и на вектор аргументов соответственно (это адреса аргументов функции main программы). Многие реализации M**PI** требуют, чтобы процесс до вызова **MPI_Init** не делал ничего, что могло бы изменить его состояние, например открытие или чтение/запись файлов, включая стандартный ввод и вывод.

Поскольку для сложных приложений, которые состоят из многих модулей и пишутся разными людьми, это отследить трудно, введена дополнительная функция **MPi_Initialized**:

MPI_Initialized (int *flag)

Здесь OUT flag - признак инициализации параллельной части программы.

Если функция **MPi_Init** уже была вызвана, то через параметр flag возвращается значение 1, в противном случае 0.

После окончания работы с MPI-функциями необходимо вызывать функцию **MPI_Final**ize. Ее прототип:

MPI_Finalize (void);

Все последующие обращения к любым **MPI**-функциям, в том числе к **MPI_Init**, запрещены. К моменту вызова **MPI_Finalize** каждым процессом программы все действия, требующие его участия в обмене сообщениями, должны быть завершены.

Общий вид MPI-программы:

```
#include "mpi.h"
/* Другие описания */
main(int argc, char *argv[])
 {
/* Описания локальных переменных */
MPI Init(&argc, &argv);
/* Тело программы */
```

```
MPI_Finalize();
return 0;
}
```

Сообщения

Различают обмен сообщениями:

попарный - сообщение посылается одним процессом другому, в обмене участвуют только два процесса.

коллективный - сообщение посылается процессом всем процессам из его группы (коммуникатора) и получается всем процессами из его группу (коммуникатора). Коллективный обмен может быть представлен как последовательность попарных обменов, однако специализированные **MPI**-функции для коллективных обменов учитывают специфику построения конкретной вычислительной установки и могут выполняться значительно быстрее соответствующей последовательности попарных обменов.

Различают обмен сообщениями:

синхронный - процесс-отправитель сообщения переходит в состояние ожидания, пока процесс-получатель не будет готов взять сообщение, а процесс-получатель сообщения переходит в состояние ожидания, пока процесс-отправитель не будет готов послать сообщение;

асинхронный - процесс-отправитель сообщения не ждет готовности процесса-получателя, сообщение копируется под системой MPI во внутренний буфер, и отправитель продолжает работу.

Поведение некачественной программы может быть разным при использовании синхронного или асинхронного режима. Например, если процесс А посылает сообщение процессу В, который не вызывает функцию получения сообщения, то это приводит к "зависанию" процесса А в синхронном режиме. В асинхронном режиме такая программа будет работать.

Основные составляющие сообщения:

1. **Блок данных сообщения** - представляется типом void*.

2. **Информация о данных сообщения**:

(a) **тип данных** - представляется типом MPI_Datatype; соответствие MPI-типов данных и типов языка С приведены в табл. 1.

(b) **количество данных** - количество единиц данного типа в блоке сообщения. Представляется типом int.

Таблица 1. Соответствие типов данных **MPI** и типов языка С

Тип **MPI**	Тип С
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_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	unsigned char

3. **Информация о получателе и отправителе сообщения**:

(a) **коммуникатор** - идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться между собой сообщениями. Представляется типом **MPI_Comm**. Как получатель, так и отправитель, должны принадлежат указанной группе. Процесс может принадлежать одновременно многим группам, все запущенные процессы принадлежат группе с идентификатором **MPI_COMM_WORLD**.

(b) **ранг получателя** - номер процесса-получателя в указанной группе (коммуникаторе). Представляется типом int. Это поле отсутствует при коллективных обменах сообщениями.

(c) **ранг отправителя** - номер процесса-отправителя в указанной группе (коммуникаторе). Представляется типом int. Получатель может указать, что он будет принимать только сообщения от отправителя с определенным рангом, или использовать константу **MPI_ANY_SOURCE**, позволяющую принимать сообщения от любого отправителя в группе с указанным идентификатором. Это поле отсутствует при коллективных обменах сообщениями.

4. **Тег сообщения** - произвольное число типа int (стандарт гарантирует возможность использования чисел от 0 до 2^{15}), приписываемое отправителем сообщению. Получатель может указать, что он будет принимать только сообщения, имеющие определенный тег, или использовать константу **MPI_ANY_TAG**, позволяющую принимать сообщения с любым тегом. Это поле отсутствует при коллективных обменах сообщениями, поскольку все процессы обмениваются одинаковыми по структуре данными.

Коммуникаторы

Коммуникатор - это объект типа **MPI_Comm**, представляющий собой идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться сообщениями. Коммуникатор является аргументов всех функций, осуществляющих обмен сообщениями. Программа может разделять исходную группу всех запущенных процессов, идентифицируемую коммуникатором **MPI_COMM_WORLD**, на подгруппы для:

организации коллективного обмена внутри этих подгрупп,

изоляции одних обменов на другие (так часто поступают параллельные библиотечные функции, чтобы их сообщения не пересекались с сообщениями вызвавшего их процесса),

учета топологии распределенной вычислительной установки (например, часто распределенный кластер можно представить в виде пространственной решетки, составленной из скоростных линий связи, в узлах которой находятся рабочие станции; скорости обмена данными между узлами тут различны и это можно учесть введением соответствующих коммуникаторов).

Получить количество процессов в группе (коммуникаторе) можно с помощью функции **MPI_Comm_size**. Ее прототип:

int MPI_Comm_size (MPI_Comm comm, int *size)

где comm - идентификатор коммуникатора (входной параметр), size - указатель на результат. Например, общее количество запущенных процессов можно получить следующим образом:

```
int p;
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

Получить ранг (номер) процесса в группе (коммуникаторе) можно с помощью функции **MPI_Comm_rank**. Ее прототип:

int MPI_Comm_rank(MPI_Comm comm, int *rank)

где comm - идентификатор коммуникатора, OUT rank - номер процесса в коммуникаторе comm.

Значение, возвращаемое функцией **MPI_comm_rank** через переменную rank, лежит в диапазоне от 0 до size-1.

Например, номер текущего процесса среди всех запущенных можно получить следующим образом:

```
Int my_rank;
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
double MPI_Wtime(void)
```

Эта функция возвращает астрономическое время в секундах (вещественное число), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время существования процесса. Заметим, что эта функция возвращает результат своей работы не через параметры, а явным образом.

Простейший пример программы, в которой использованы описанные выше функции, выглядит так:

```
main(int argc, char **argv)
{ int me, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&me);
MPI_Comm_size(MPI_COMM_WORLD,&size);
printf("Process %d size %d \n", me, size);
...
MPI_Finalize();
...
```

Строка, соответствующая функции printf, будет выведена столько раз, сколько процессов было порождено при вызове **MPI_Init**. Порядок появления строк заранее не определен и может быть, вообще говоря, любым. Гарантируется только то, что содержимое отдельных строк не будет перемешано друг с другом.

Прием/передача сообщений между отдельными процессами

Все функции данной группы делятся на два класса: функции с блокировкой (с синхронизацией) и без блокировки (асинхронные).

Прием/передача сообщений с блокировкой задаются конструкциями следующего вида.

Послать сообщение можно с помощью функции **MPI_Send**. Ее прототип:

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

где

buf - адрес начала буфера с посылаемым сообщением;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
dest - номер процесса-получателя;
msgtag-идентификатор сообщения;
comm - идентификатор коммуникатора.

Блокирующая посылка сообщения с идентификатором msgtag, состоящей из count элементов типа datatype, процессу с номером dest. Все элементы посылаемого сообщения расположены подряд в буфере buf. Значение count может быть нулем. Разрешается передавать сообщение самому себе. Тип передаваемых элементов datatype должен указываться с помощью предопределенных констант типа, например, **MPI_INT, MPI_LONG, Mpi_SHORT, MPI_LONG_DOUBLE, MPI_CHAR, MPI_UNSIGNED_CHAR**, **MPI_FLOAT** или с помощью введенных производных типов. Для каждого типа данных языков Fortran и C есть своя константа. Полный список предопределенных имен типов можно найти в файле mpi.h.

Эта функция может перевести текущий процесс в состояние ожидания, пока получатель с номером dest в группе comm не примет сообщение с тегом tag (если используется синхронный режим обмена).

Блокировка гарантирует корректность повторного использования всех параметров после возврата из подпрограммы. Это означает, что после возврата из данной функции можно использовать любые присутствующие в вызове функции переменные без опасения

испортить передаваемое сообщение. Выбор способа осуществления этой гарантии: копирование в промежуточный буфер или непосредственная передача процессу dest, остается за разработчиками конкретной реализации MPI.

Возврат из функции **MPI_Send** не означает ни того, что сообщение получено процессом dest, ни того, что сообщение покинуло процессорный элемент, на котором выполняется процесс, запустивший **MPI_Send**. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове данной функции.

Чтобы расширить возможности передачи сообщений, в **MPI** введены дополнительные три функции. Все параметры у этих функций такие же, как и у функции **MPI_Send**, однако у каждой из них есть своя особенность.

MPI_Bsend - передача сообщения с буферизацией. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в буфер и произойдет немедленный возврат из функции. Выполнение данной функции никак не зависит от соответствующего вызова функции приема сообщения. Тем не менее, функция может вернуть код ошибки, если места под буфер недостаточно.

MPI_Ssend - передача сообщения с синхронизацией. Выход из данной функции произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем. Таким образом, завершение передачи с синхронизацией говорит не только о возможности повторного использования буфера, но и о гарантированном достижении процессом-получателем точки приема сообщения в программе. Если приём сообщения также выполняется с блокировкой, то функция **MPI_Ssend** сохраняет семантику блокирующих вызовов.

MPI_Rsend - передача сообщения по готовности. Данной функцией можно пользоваться только в том случае, если процесс-получатель уже инициировал прием сообщения. В противном случае вызов функции, вообще говоря, является ошибочным и результат ее выполнения не определен. Во многих реализациях функция **MPI_Rsend** сокращает протокол взаимодействия между отправителем и получателем, уменьшая накладные расходы на организацию передачи.

Получить сообщение можно с помощью функции **MPI_Recv**. Ее прототип:

int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,
int msgtag, MPI Comm comm, MPI_Status *status)

где

OUT buf - адрес начала буфера для приема сообщения;
count - максимальное число элементов в принимаемом сообщении;
datatype - тип элементов принимаемого сообщения;
source - номер процесса-отправителя (может быть MPI_ANY_SOURCE);
msgtag - идентификатор принимаемого сообщения (может быть MPI_ANY_SOURCE);
comm - идентификатор коммуникатора;
OUT status - параметры принятого сообщения.

Прием сообщения с идентификатором msgtag от процесса source с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения count. Если число принятых элементов меньше значения count, то гарантируется, что в буфере buf изменятся только элементы, соответствующие элементам принятого сообщения.

Блокировка гарантирует, что после возврата из функции все элементы сообщения уже будут приняты и расположены в буфере buf.

При приеме сообщения в качестве номера процесса-отправителя можно указать предопределенную константу **MPI_ANY_SOURCE** - признак того, что подходит сообщение от любого процесса. В качестве идентификатора принимаемого сообщения можно указать константу **MPI_ANY_TAG** - это признак того, что подходит сообщение с любым идентификатором. При одновременном использовании этих двух констант будет принято любое сообщение от любого процесса.

Параметры принятого сообщения всегда можно определить по соответствующим полям структуры status. Предопределенный тип **MPI_Status** описан в файле mpi.h. В языке C параметр status является структурой, содержащей поля с именами **MPI_SOURCE**,

MPI_TAG и **MPI_ERROR**. Реальные значения номера процесса-отправителя, идентификатора сообщения и код ошибки доступны через status.**MPI_SOURCE**, status. **MPI_TAG** и status.**MPI_ERROR**. В языке Fortran параметр status является целочисленным массивом размера **MPI_STATUS_SIZE**. Константы **MPI_SOURCE**, **MPI_TAG** и **MPI_ERROR** являются индексами по данному массиву для доступа к значениям соответствующих полей, например, status(**MPI_SOURCE**).

Имеет место некоторая несимметричность операций посылки и приема сообщений. С помощью константы **MPI_ANY_SOURCE** можно принять сообщение от любого процесса. Однако в случае посылки данных требуется явно указать номер принимающего процесса.

В стандарте оговорено, что если один процесс последовательно посылает два сообщения другому процессу, и оба эти сообщения соответствуют одному и тому же вызову **MPI_Recv**, то первым будет принято сообщение, которое было отправлено раньше. Вместе с тем, если два сообщения были одновременно отправлены разными процессами и оба сообщения соответствуют одному и тому же вызову **MPI_Recv**, то порядок их получения принимающим процессом заранее не определен.

MPI не гарантирует выполнения свойства "справедливости" при распределении приходящих сообщений. Предположим, что процесс P1 послал сообщение, которое может быть принято процессом P2. Однако прием может не произойти, в принципе, сколь угодно долгое время. Такое возможно, например, если процесс P3, постоянно посылает сообщения процессу P2, также подходящие под шаблон **MPI_Recv**.

Последнее замечание относительно использования блокирующих функций приема и посылки связано с возможным возникновением тупиковой ситуации. Предположим, что работают два параллельных процесса и они хотят обменяться данными. Было бы вполне естественно в каждом процессе сначала воспользоваться функцией **MPI_Send**, а затем **MPI_Recv**. Но именно этого и не стоит делать. Дело в том, что заранее неизвестно, как реализована функция **MPI_Send**. Если разработчики для гарантии корректного повторного использования буфера посылки заложили схему, при которой посылающий процесс ждет начала приема, то возникнет классический тупик. Первый процесс не может вернуться из функции посылки, поскольку второй не начинает прием сообщения. А второй процесс не может начать прием сообщения, поскольку сам по похожей причине застрял на посылке. Выход из этой ситуации прост. Нужно использовать либо неблокирующие функции приема/передачи, либо функцию совмещенной передачи и приема.

Для определения размера области памяти, выделяемой для хранения сообщения, используется функция **MPI_Probe**. Ее прототип:

int MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI Status *status)

Здесь:

source - номер процесса-отправителя или MPI_ANY_SOORCE;

msgtag - идентификатор ожидаемого сообщения или MPI_ANY_TAG;

comm - идентификатор коммуникатора;

OUT status - параметры найденного подходящего сообщения.

Возврата из функции не произойдет до тех пор, пока сообщение с подходящим идентификатором и номером процесса-отправителя, не будет доступно для получения. Атрибуты доступного сообщения можно определить обычным образом с помощью параметра status. Функция определяет только факт прихода сообщения, но реально его не принимает.

Прием/передача сообщений без блокировки.

В отличие от функций с блокировкой, возврат из функций данной группы происходит сразу без какой либо блокировки процессов. На фоне дальнейшего выполнения программы одновременно происходит и обработка асинхронно запущенной операции. В принципе, данная возможность исключительно полезна для создания эффективных программ. В самом деле, программист знает, что в некоторый момент ему потребуется массив, который вычисляет другой процесс. Он заранее выставляет в программе асинхронный запрос на

получение данного массива, а до того момента, когда массив реально потребуется, он может выполнять любую другую полезную работу. Опять же, во многих случаях совершенно не обязательно дожидаться окончания посылки сообщения до выполнения последующих вычислений.

Сделанные замечания касаются только вопросов эффективности. В отношении предоставляемой функциональности асинхронные операции исключительно полезны, поэтому они присутствуют практически в каждой реальной программе.

Послать сообщение без блокирования процесса можно с помощью функции **MPI_Isend.** Ее прототип**:**

int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dast,
int msgtag, MPI_Comm comm, MPI_Request *request)

Здесь:

buf - адрес начала буфера с посылаемым сообщением;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
dest - номер процесса-получателя;
msgtag - идентификатор сообщения;
comm - идентификатор коммуникатора;
OUT request - идентификатор асинхронной операции.

Передача сообщения аналогична вызову **MPI_Send**, однако возврат из функции **MPI_Isend** происходит сразу после инициализации процесса передачи без ожидания обработки всего сообщения, находящегося в буфере buf. Это означает, что нельзя что-то записывать в данный буфер без получения дополнительной информации, подтверждающей завершение посылки. Определить тот момент времени, когда можно повторно использовать буфер buf без опасения испортить передаваемое сообщение, можно с помощью параметра request и функций **MPI_Wait** и **MPI_Test**.

Аналогично трем модификациям функции **MPI_Send**, предусмотрены три дополнительные варианта функций **MPI_Ibsend**, **MPI_Issend**, **MPI_Irsend**. К изложенной выше семантике работы этих функций добавляется асинхронность.

Прототип функции **MPI_Irecv:**

int MPI_Irecv(void *buf, int count, MPI_Datatype
datatype, int source, int msgtag, MPI_Comm comm, MPI_Request
***request)**

Здесь:

OUT buf - адрес начала буфера для приема сообщения;
count - максимальное число элементов в принимаемом сообщении;
datatype-тип элементов принимаемого сообщения;
source - номер процесса-отправителя;
msgtag - идентификатор принимаемого сообщения;
comm - идентификатор коммуникатора;
OUT request - идентификатор операции асинхронного приема сообщения.

Прием сообщения, аналогичный **MPI_Recv**, однако возврат из функции происходит сразу после инициализации процесса приема без ожидания получения и записи всего сообщения в буфере buf. Окончание процесса прием можно определить с помощью параметра request и процедур типа **MPI_Wait** и **MPI_Test**.

Сообщение, отправленное любой из функций **MPI_Send**, **MPI_Isend** и любой из трех их модификаций, может быть принято любой из процедур **MPI_Recv** и **MPI_Irecv**.

С помощью данной функции легко обойти возможную тупиковую ситуацию. Заменим вызов функции приема сообщения блокировкой на вызов функции **MPI_Irecv**. Расположим его перед вызове функции **MPI_Send**, т. е. преобразуем фрагмент:

```
...
MPI_Send (...)
MPI_Recv(.. .)
```

следующим образом

...

MPI_Irecv (...)

MPI Send (...)

В такой ситуации тупик гарантированно не возникнет, поскольку к момент вызова функции **MPI_Send** запрос на прием сообщения уже будет выставлен.

Дождаться доставки сообщения можно с помощью функции **MPI_Wait.** Ее прототип**:**

int MPI_Wait (MPI_Request *request, MPI_Status *status)

Здесь:

request - идентификатор операции асинхронного приема или передачи;

OUT status - параметры сообщения.

Ожидание завершения асинхронной операции, ассоциированной с идентификатором request и запущенной функциями **MPI_Isend** или **MPI_Irecv**. Пока асинхронная операция не будет завершена, процесс, выполнивший функцию **MPI_Wait**, будет заблокирован. Если речь идет о приеме, атрибуты и длину принятого сообщения можно определить обычным образом с помощью параметра status.

Прототип функции **MPI_Waitall:**

int MPI_Waitall (int count, MPI_Bequest *requests, MPi_Status *statuses)

Здесь:

court - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи;

OUT statuses - параметры сообщений.

Выполнение процесса блокируется до тех пор, пока все операции обмена, ассоциированные с указанными идентификаторами, не будут завершены. Если во время одной или нескольких операций обмена возникли ошибки, то поле ошибки в элементах массива statuses будет установлено в соответствующее значение.

Если требуется дождаться доставки одного из нескольких сообщений, то можно использовать функцию **MPI_Waitany**. Ее прототип:

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

Здесь:

count - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи;

OUT index - номер завершенной операции обмена;

OUT status - параметры сообщения.

Выполнение процесса блокируется до тех пор, пока какая-либо асинхронная операция обмена, ассоциированная с указанными идентификаторами, не будет завершена. Если завершились несколько операций, то случайным образом будет выбрана одна из них. Параметр index содержит номер элемент в массиве requests, содержащего идентификатор завершенной операции.

Прототип функции **MPI_Waitsome:**

int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indexes, MPI_Status *statuses)

Здесь:

incount - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи:

OUT outcount - число идентификаторов завершившихся операций обмен;

OUT indexes - массив номеров завершившихся операций обмена;

OUT statuses - параметры завершившихся операций приема сообщения.

Выполнение процесса блокируется до тех пор, пока одна из операций обмена, ассоциированных с указанными идентификаторами, не будет завершена. Параметр outcount содержит число завершенных операций, а первые outcount элементов массива indexes содержат номера элементов массива requests с их идентификаторами. Первые outcount элементов массива statuses содержат параметры завершенных операций.

Узнать, доставлено ли сообщение, можно с помощью функции **MPI_Test**. Ее прототип**:**

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

Здесь:

request - идентификатор операции асинхронного приема или передач;

OUT flag - признак завершенности операции обмена;

OUT status - параметры сообщения.

Проверка завершенности асинхронных функций **MPI_Isend** или **MPI_lrecv**, ассоциированных с идентификатором request. В параметре flag функция **MPI_Test** возвращает значение 1, если соответствующая операция завершена, и значение 0 в противном случае. Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра status.

Прототип функции **MPI_Testall:**

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

Здесь:

count - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи;

OUT flag - признак завершенности операций обмена;

OUT statuses - параметры сообщений.

В параметре flag функция возвращает значение 1, если все операции, ассоциированные с указанными идентификаторами, завершены. В этом случае параметры сообщений будут указаны в массиве statuses. Если какая-либо из операций не завершилась, то возвращается 0, и определенность элементов массива statuses не гарантируется.

Если требуется проверить доставку одного из нескольких сообщений, то можно использовать функцию **MPI_Testany**. Ее прототип:

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

Здесь:

count - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи;

OUT index - номер завершенной операции обмена;

OUT flag - признак завершенности операции обмена;

OUT status - параметры сообщения.

Если к моменту вызова функции **Mpi_Testany** хотя бы одна из операций асинхронного обмена завершилась, то в параметре flag возвращается значение 1, index содержит номер соответствующего элемента в массиве requests, а status - параметры сообщения. В противном случае в параметре flag будет возвращено значение 0.

int MPI_Testsome(int incount, MPI_Request *requests, int
***outcount, int *indexes, MPI_Status *statuses)**

Здесь:

incount - число идентификаторов асинхронных операций;

requests - идентификаторы операций асинхронного приема или передачи;

OUT outcount - число идентификаторов завершившихся операций обмена;

OUT indexes - массив номеров завершившихся операций обмена;

OUT statuses - параметры завершившихся операций приема сообщений.

Данная функция работает так же, как и **MPI_Waitsome**, за исключением того, что возврат происходит немедленно. Если ни одна из указанных операций не завершилась, то значение outcount будет равно нулю.

Прототип:

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

Здесь:

source - номер процесса-отправителя или MPI_ANY_SOURCE;

msgtag - идентификатор ожидаемого сообщения или MPI_ANY_TAG;

comm - идентификатор коммуникатора;
OUT flag - признак завершенности операции обмена;
OUT status - параметры подходящего сообщения.

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки. В параметре flag возвращается значение 1, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично **MPI_Probe**), и значение 0, если сообщения указанными атрибутами еще нет.

Объединение запросов на взаимодействие.

Процедуры данной группы позволяют снизить накладные расходы, возникающие в рамках одного процессора при обработке приема/передачи и перемещении необходимой информации между процессом и сетевым контроллером. Несколько запросов прием и/или передачу могут объединяться вместе для того, чтобы далее можно было бы запустить одной командой. Способ приема сообщения не зависит от способа его посылки: сообщение, отправленное с помощью объединения запросов либо обычным способом, может быть принято как обычным способом, так и с помощью объединения запросов.

int MPI_Send_init(void *buf, int count, MPI_Datatvpe datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)

Здесь:
buf - адрес начала буфера с посылаемым сообщением;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
dest - номер процесса-получателя;
msgtag - идентификатор сообщения;
comm - идентификатор коммуникатора;
OUT request-идентификатор асинхронной передачи.

Формирование запроса на выполнение пересылки данных. Все параметры точно такие же, как и у подпрограммы **MPI_Isend**, однако, в отличие от нее пересылка не начинается до вызова подпрограммы **MPI_Startall**. Как и прежде, дополнительно предусмотрены варианты и для трех модификаций посылки сообщений: **MPI_Bsend_init**, **MPI_Ssend_init**, **MPI_Rsend_init**.

int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int msgtag, MPI_Comm comm, MPI_Request *request)

Здесь:
OUT buf - адрес начала буфера приема сообщения;
count - число принимаемых элементов в сообщении;
datatype - тип принимаемых элементов;
source - номер процесса-отправителя;
msgtag - идентификатор сообщения;
comm - идентификатор коммуникатора;
OUT request - идентификатор асинхронного приема.

Формирование запроса на выполнение приема сообщения. Все параметры точно такие же, как и у функции **MPI_Irecv**, однако, в отличие от нее, реальный прием не начинается до вызова функции **MPI_Startall**.

MPI_Startall(int count, MPI_Request *requests)

Здесь:
count - число запросов на взаимодействие;
OUT requests - массив идентификаторов приема/передачи.

Запуск всех отложенных операций передачи и приема, ассоциированных с элементами массива запросов requests и инициированных функциями **MPI_Recv_init**, **MPI_Send_init** или

ее тремя модификациями. Все отложенные взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью функций семейств **MPI_Wait** и **MPI_Test**.

Совмещение приема и передачи сообщений

Совмещение приема и передачи сообщений между процессами позволяет легко обходить множество подводных камней, связанных с возможными тупиковыми ситуациями. Предположим, что в линейке процессов необходимо организовать обмен данными между i-м и i + 1-м процессами. Если воспользоваться стандартными блокирующими функциями посылки сообщений, то возможен тупик, обсуждавшийся ранее. Один из способов обхода такой ситуации состоит в использовании функции совмещенного приема и передачи.

Функция **MPI_Sendrecv** объединяет в едином запросе посылку и прием сообщений. Ее прототип:

int MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype,
int dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype,
int source, MPI_Datatype rtag, MPI_Comm comm, MPI_Status *status)

Здесь:

sbuf - адрес начала буфера с посылаемым сообщением;
scount - число передаваемых элементов в сообщении;
stype - тип передаваемых элементов;
dest - номер процесса- получателя;
stag - идентификатор посылаемого сообщения;
OUT rbuf - адрес начала буфера приема сообщения;
rcount - число принимаемых элементов сообщения;
rtype - тип принимаемых элементов;
source - номер процесса-отправителя;
rtag - идентификатор принимаемого сообщения;
comm - идентификатор коммуникатора;
OUT status - параметры принятого сообщения.

Эта функция может привести текущий процесс в состояние ожидания, пока получатель с номером dest в группе comm не примет сообщение с тегом sendtag (если используется синхронный режим обмена) или пока отправитель с номером source (или любым номером, если в качестве ранга используется константа **MPI_ANY_SOURCE**) в группе comm не отправит сообщение с тегом recvtag (или любым тегом, если в качестве тега используется константа **MPI_ANY_TAG**).

Принимающий и отправляющий процессы могут являться одним и тем процессом. Буфера приема и посылки обязательно должны быть различными. Сообщение, отправленное операцией **MPI_Sendrecv**, может быть принято обычным образом, и точно так же операция **MPI_Sendrecv** может принять сообщение, отправленное обычной операцией **MPI_Send**.

Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов *участвуют все* **процессы коммуникатора**. Соответствующая процедура должна быть вызвана каждым процессом, быть может, со своим набором параметров. Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено. Как и для блокирующих процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки. Асинхронных коллективных операций в **MPI** нет.

В коллективных операциях можно использовать те же коммуникаторы, что и были использованы для операций типа "точка-точка". **MPI** гарантирует, что сообщения, вызванные коллективными операциями, никак не повлияют и не пересекутся с сообщениями, появившимися в результате индивидуального взаимодействия процессов.

Вообще говоря, нельзя рассчитывать на синхронизацию процессов с помощью коллективных операций. Если какой-то процесс уже завершил свое участие в коллективной операции, то это не означает ни того, что данная операция завершена другими процессами коммуникатора, ни даже того, что она ими начата (конечно же, если это возможно по смыслу операции).

В коллективных операциях не используются идентификаторы сообщений.

Для рассылки данных из одного процесса всем остальным в его группе можно использовать функцию **MPI_Bcast**. Ее прототип:

int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)

Здесь:

OUT buf - адрес начала буфера посылки сообщения;
count - число передаваемых элементов в сообщении;
datatype - тип передаваемых элементов;
source - номер рассылающего процесса;
comm - идентификатор коммуникатора.

Рассылка сообщения от процесса source всем процессам, включая рассылающий процесс. При возврате из процедуры содержимое буфера buf процесса source будет скопировано в локальный буфер каждого процесса коммуникатора comm. Значения параметров count, datatype, source и comm должны быть одинаковыми у всех процессов. В результате выполнения следующего оператора всеми процессами коммуникатора comm

MPI_Bcast(array,100,MPI_INT,0,comm)

первые сто целых чисел из массива array нулевого процесса будут скопированы в локальные буфера array каждого процесса.

int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int roount, MPI_Datatype rtype, int dest, MPI_Comm comm)

Здесь:

sbuf - адрес начала буфера посылки;
scount - число элементов в посылаемом сообщении;
stype - тип элементов отсылаемого сообщения;
OUT rbuf - адрес начала буфера сборки данных;
rcount - число элементов в принимаемом сообщении;
rtype - тип элементов принимаемого сообщения;
dest - номер процесса, на котором происходит сборка данных;
comm - идентификатор коммуникатора.

Сборка данных со всех процессов в буфере rbuf процесса dest. Каждый процесс, включая dest, посылает содержимое своего буфера sbuf процессу dest. Собирающий процесс сохраняет данные в буфере rbuf, располагая их в порядке возрастания номеров процессов. На процессе dest существенными являются значения всех параметров, а на всех остальных процессах - только значения параметров sbuf, scount, stype, dest и comm. Значения параметров dest и comm должны быть одинаковыми у всех процессов. Параметр rcount у процесса dest обозначает число элементов типа rtype, принимаемых не от всех процессов в сумме, а от каждого процесса. С помощью похожей функции **MPI_Gatherv** можно принимать от процессов массивы данных разной длины.

Функция **MPI_Scatter** по своему действию является обратной к **MPI_Gather**. Ее прототип:

int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void *rbuf, int roount, MPI_Datatype rtype, int source, MPI_Comm comm)

Здесь:

sbuf - адрес начала буфера посылки;
scount - число элементов в посылаемом сообщении;
stype - тип элементов отсылаемого сообщения;

OUT rbuf - адрес начала буфера сборки данных;
rcount - число элементов в принимаемом сообщении;
rtype - тип элементов принимаемого сообщения;
source - номер процесса, на котором происходит сборка данных;
comm - идентификатор коммуникатора.

Процесс source рассылает порции данных из массива sbuf всем n процессам приложения. Можно считать, что массив sbuf делится на n равных частей, состоящих из scount элементов типа stype, после чего i-я часть посылается i-му процессу. На процессе source существенными являются значения всех параметров, а на всех остальных - только значения параметров rbuf,rcount,rtype,source и comm. Значения параметров source и comm должны быть одинаковы у всех процессов.

В следующем примере показано использование функции **MPI_Scatter** для рассылки строк массива.

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(argc,argv) ;
int argc;
char *argv[];  {
int numtasks, rank, sendcount,recvcount,source;
float sendbuf[SIZE][SIZE]={
 {1.0,2.0,3.0,4.0},
 {5.0,6.0,7.0,8.0},
 {9.0,10.0,11.0,12.0},
 {13.0,14.0,15.0,16.0}} ;
float recvbuf[SIZE];
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount=SIZE;
  MPI_Scatter(sendbuf,sendcount, MPI_FLOAT, recvbuf, recvcount,
            MPI_FLOAT,source,MPI_COMM_WORLD);
 printf("rank=%d Results: %f %f %f\n", rank, recvbuf[0],
       recvbuf[1],  recvbuf[2],  recvbuf[3]);
        }
        else
        printf("Число процессов должно быть равно %d\n", SIZE);
MPI_Finalize();
}
```

К коллективным операциям относятся и редукционные операции. Такие операции предполагают, что на каждом процессе хранятся некоторые данные, над которыми необходимо выполнить единую операцию, например, операцию сложения чисел или операцию нахождения максимального значения. Операция может быть либо предопределенной операцией **MPI**, либо операцией, определенной пользователем. Каждая предопределенная операция имеет свое имя, например, **MPI_MAX**, **MPI_MIN**, **MPI_SUM**, **MPI_PROD** и т. п.

Если требуется, чтобы результат, полученный посредством функции **MPI_Reduce**, стал известен всем процессам группы, то эффективным решением является не функция **MPI_Bcast**, а функция **MPI_Allreduce**. Ее прототип:

int MPI_Allreduce(void *sbuf, void *rbuf, int count,

MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Здесь:

sbuf - адрес начала буфера для аргументов операции op;

OUT rbuf - адрес начала буфера для результата операции op;

count - число аргументов у каждого процесса;

datatype - тип аргументов;

op - идентификатор глобальной операции;

comm - идентификатор коммуникатора.

Данная функция задает выполнение count независимых глобальных операций op. Предполагается, что в буфере sbuf каждого процесса расположено count аргументов, имеющих тип datatype. Первые элементы массивов sbuf участвуют в первой операции op, вторые элементы массивов sbuf участвуют во второй операции op и т. д. Результаты выполнения всех count операций записываются в буфер rbuf на каждом процессе. Значения параметров count, datatype, op и comm у всех процессов должны быть одинаковыми. Из соображений эффективности реализации предполагается, что операция op обладает свойствами ассоциативности и коммутативности.

Часто требуется выполнить в каком-то смысле обратную к производимой функцией **MPI_Bcast** операцию – переслать данные одному из процессов группы от всех остальных процессов группы; при этом часто нужно получить не сами эти данные, а некоторую функцию от них, например, сумму всех данных. Для этого можно использовать функцию **MPI_Reduce.** Ее прототип:

int MPI_Reduce (void *sbuf,void *rbuf,int count,MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)

Здесь:

sbuf - адрес начала буфера для аргументов;

OUT rbuf - адрес начала буфера для результата;

count - число аргументов у каждого процесса;

datatype - тип аргументов;

op - идентификатор глобальной операции;

root - процесс-получатель результата;

comm - идентификатор коммуникатора.

Функция аналогична предыдущей, но результат операции будет записан в буфер rbuf не у всех процессов, а только у процесса root.

Синхронизация процессов

Синхронизация процессов в MPI осуществляется с помощью единственной функции MPI_Barrier. Ее прототип:

int MPI_Barrier(MPI_Comm comm)

comm - идентификатор коммуникатора.

Функция блокирует работу вызвавших ее процессов до тех пор, пока все оставшиеся процессы коммуникатора comm также не выполнят эту процедуру.

Только после того, как последний процесс коммуникатора выполнит данную функцию, все процессы будут разблокированы и продолжат выполнение дальше.

Данная функция является коллективной. Все процессы должны вызвать **MPI_Barrier**, хотя реально исполненные вызовы различными процессами коммуникатора могут быть расположены в разных местах программы.

Работа с группами процессов

В MPI есть значительное множество функций, ориентированных на работу с коммуникаторами и группами процессов. Они позволяют сравнить состав групп, определить их пересечение, объединение, добавить процессы в группу, удалить группу и т. д. В качестве примера приведем лишь один способ образования новых групп на основе существующих.

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

Здесь:

comm - идентификатор существующего коммуникатора;

color - признак разделения на группы;

key - параметр, определяющий нумерацию в новых группах;

OUT newcomm - идентификатор нового коммуникатора.

Данная процедура разбивает все множество процессов, входящих в группу comm, на непересекающиеся подгруппы - одну подгруппу на каждое значение параметра color. Значение параметра color должно быть неотрицательным целым числом. Каждая новая подгруппа содержит все процессы которых параметр color имеет одно и то же значение, т. е. в каждой новой подгруппе будут собраны все процессы "одного цвета". Всем процесс подгруппы будет возвращено одно и то же значение newcomm. Параметр key определяет нумерацию в новой подгруппе.

Предположим, что нужно разделить все процессы программы на две подгруппы в зависимости от того, является ли номер процесса четным или нечетным. В этом случае в поле color достаточно поместить выражение My_Id%2, где My_Id - это номер процесса. Значением данного выражения может быть либо 0, либо 1. Все процессы с четными номерами автоматически попадут в одну группу, а процессы с нечетными в другую.

int MPI_Comm_free(MPl_Comm comm)

Здесь OUT comm - идентификатор коммуникатора.

Появление нового коммуникатора всегда вызывает создание новой структуры данных. Если созданный коммуникатор больше не нужен, то соответствующую область памяти необходимо освободить. Данная функция уничтожает коммуникатор, ассоциированный с идентификатором comm, который после возвращения из функции будет иметь значение **MPI_COMM_NULL**.

Часть 2. Примеры использования функций MPI

Задание №1. Написать реализацию стека строк в разделяемой памяти. При запуске программа создает блок разделяемой памяти (если его еще нет) или присоединяется к существующему блоку. Программа должна обеспечивать основные функции работы со стеком (добавить и удалить элемент) и возможность запуска себя во многих экземплярах.

Алгоритм

Сначала программа определяет свой ранг – если ранг равен нулю, то программа запускает процесс-мастер, который записывает 10 строк в стек, распределённый между дочерними процессами, а затем забирает 10 строк из распределённого стека. Если ранг не равен нулю, программа запускает дочерний процесс, который ждёт от процесса-мастера сообщение.

Есть три типа сообщений:

MSG_CAN_PUSH - запрос на запись в стек (дочерний процесс возвращает не ноль, если в его части стека осталось свободное место)

MSG_CAN_POP – запрос на извлечение из стека (дочерний процесс возвращает размер строки, находящейся в вершине стека или ноль, если буфер этого процесса пуст)

MSG_KILL – сигнал завершения дочернего процесса

Дочерний процесс запускает бесконечный цикл, обрабатывающий эти запросы. После исполнения запроса MSG_CAN_PUSH дочерний процесс получает от процесса-мастера строку, которую добавляет в свой буфер и увеличивает счётчик. Когда значение счётчика превышает размер буфера, дочерний процесс перестаёт получать строки. После исполнения запроса MSG_CAN_POP дочерний процесс отсылает процессу-мастеру верхнюю строчку из своего буфера, освобождает память и уменьшает счётчик.

Для записи строки в стек, процесс-мастер в цикле посылает запрос MSG_CAN_PUSH дочерним процессам, и отсылает строчку процессу, который ответит не нулевым сообщением (есть место в буфере).

Для извлечения строки из стека, процесс-мастер в обратном порядке посылает сообщение MSG_CAN_POP и когда ответ содержит ненулевое значение – длину строки (эта строка гарантированно вершина стека), получает эту строку.

Перед тем, как процесс- мастер завершает работу, он посылает всем дочерним процессам сообщение MSG_KILL и они завершаются.

Текст программы

(авторы: студенты 243 группы ИМКН Иванов С.Б., Безуглов К.А., 2005 г.)

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<mpi.h>
// размер участка распределённого стека у каждого процесса
#define SIZE 2
//сообщения
#define MSG_CAN_PUSH    0
#define MSG_CAN_POP     1
#define MSG_KILL        2
int nproc;
int push(char *str) //занести  строку в стек
{
   int r,message=MSG_CAN_PUSH;
   MPI_Status status;
   for(r=1;r<nproc;r++) //цикл по процессам
    {
//запрос на наличие места в памяти
      MPI_Send(&message,1,MPI_INT,r,0,MPI_COMM_WORLD);
//получаем ответ
       MPI_Recv(&message,1,MPI_INT,r,0,MPI_COMM_WORLD,&status);
       if(message) //если есть место, то..
       {
          int size = strlen(str)+1;
          //посылаем строку
          MPI_Send(&str,size,MPI_CHAR,r,1,MPI_COMM_WORLD);
          printf("Отослал строку '%s'\n",str);
          return 1;   //возвращаем "ура!"
       }
   }
   return 0; // не нашли ни одного процесса со свободным местом
}
char * pop() //извлекаем строку из стека
{
   int r,message;
   MPI_Status status;
   for(r=nproc-1;r>0;r--)
   {
      message=MSG_CAN_POP;
//запрос на наличие строк в памяти процесса
       MPI_Send(&message,1,MPI_INT,r,0,MPI_COMM_WORLD);
//получаем ответ (размер последней строки)
```

```
        MPI_Recv(&message,1,MPI_INT,r,0,MPI_COMM_WORLD,&status);
        if(message) //если строка есть, то...
        {
          char * str = (char *)malloc(message+1);
          MPI_Recv(&str,message,MPI_CHAR,r,2,MPI_COMM_WORLD,&status);
          return str;
        }
    }
    return NULL; //все процессы пусты
}
void killall() //говорим дочерним процессам, что пора и честь знать
{
    int r,message=MSG_KILL;
    for(r=1;r<nproc;r++)
//посылаем сообщение о завершении работы
        MPI_Send(&message,1,MPI_INT,r,0,MPI_COMM_WORLD);
}

void master()
{
    printf("Мастер запущен\n");
    char * str[10];
    str[0] = "Строка 1";
    str[1] = "Строка 2";
    str[2] = "Строка 3";
    str[3] = "Строка 4";
    str[4] = "Строка 5";
    str[5] = "Строка 6";
    str[6] = "Строка 7";
    str[7] = "Строка 8";
    str[8] = "Строка 9";
    str[9] = "Строка 10";
    int i=0;
    for(;i<10;i++)
//занести 10 строк в стек
        if(!push (str[i])) fprintf(stderr,"Не могу послать строку
                    %d\n",i+1);
    for(i=0;i<10;i++)
        printf("Получил строку: '%s'\n",pop()); //получаем из стека
    killall();
}
void slave(int rank)
{
    printf("Дочерний процесс %d запущен\n",rank);
    int message;
    int used=0;  //сколько сообщений содержится в стеке текущего процесса
    char *buf[SIZE];    //место для стека
    MPI_Status status;
    while(1)
    {
//получаем запрос от сервера
        MPI_Recv(&message,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        switch(message)
        {
```

```
//процесс-мастер спрашивает, есть ли свободное место в стеке
            case                    MSG_CAN_PUSH                          :
           message=SIZE-used;  //кол-во свободных ячеек в стеке
                MPI_Send(&message,1,MPI_INT,0,0,MPI_COMM_WORLD);
                if(message)       //если место есть, то...
                {
//ждём, пока нам пришлют строчку
                  MPI_Probe(0,1,MPI_COMM_WORLD,&status);
                  int size;
                  MPI_Get_count(&status,MPI_CHAR,&size);
//определяем её размер
                  buf[used] = (char *)malloc(size);
 //выделяем память

MPI_Recv(buf[used++],size,MPI_CHAR,0,1,MPI_COMM_WORLD,&status);
 //получаем строчку у процесса-мастера
                }
              break;
            case MSG_CAN_POP    :
      //процесс-мастер спрашивает размер строки в голове стека
                if(used)
                  message=strlen(buf[used-1]);
  //размер строки в голове стека
                else message=0;
                MPI_Send(&message,1,MPI_INT,0,0,MPI_COMM_WORLD);
                if(message)       //если место есть, то...
                {
                  MPI_Send(buf[used-1],message,MPI_CHAR,0,2,
                                  MPI_COMM_WORLD);
                  free(buf[used-1]);
                  used--;
                }
              break;
            case MSG_KILL       : return;
            default           :
                fprintf(stderr,"Неизвестное сообщение.\n");
                return;
        }
    }
}
int main(int argc, char *argv[])
{
    int rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank)
      slave(rank);
    else
      master();
    MPI_Finalize();
    return 0;
}
```

Задание №2. Написать MPI программу, получающую в качестве аргументов соответствующую часть массива n x n целых a чисел, целое число n, номер процесса k, общее количество процессов p и заменяющую матрицу a на её транспонированную. При этом должна быть обеспечена равномерная загрузка всех процессов.

Основная программа должна начинать работу с MPI, вводить число n и массив a (из файла или по заданной формуле), вызывать эту программу и выводить на экран результат её работы.

Алгоритм

Основная программа делит входной массив (задаётся по формуле) на k-1 частей, - каждая часть состоит из n/(k-1) столбцов (деление нацело) и n строк, а последняя (k-1) часть из n-(n/k-1)*(k-2) столбцов и n строк.

Затем каждая из полученных частей поочерёдно записывается в одномерный массив по столбцам (сверху-вниз и слева направо) и отсылается соответствующему процессу.

После того, как все части разосланы - главная программа начинает получать результаты обратно от подчинённых процессов в том же порядке, в котором части были разосланы.

Полученные части записываются построчно в массив. Массив сохраняется в файл 'out.txt'.

Подчинённые процессы просто получают блок данных и отсылают его обратно.

Тексты программ

(авторы: студенты 243 г. ИМКН Мелехин С.П., Дударева М.И., 2005г.)

Файл shared.h

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<mpi.h>
#define SIZE 1024
```

Файл master.c

```
#include "shared.h"
int array[SIZE][SIZE];
void save_result(char * fname);
int main(int argc, char *argv[])
{
   int myrank,nproc;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);
   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
   printf("Master started. My rank is %d, npr is %d\n",myrank,nproc);

   MPI_Status status;
   int tmp_dim = SIZE/(nproc-1);
   int tmp_begin=0;
   int tmp_end;
   int * tmp_arr;
   int tmp_size;
   int i,j,k;
   int r;

   for(i=0,k=0;i<SIZE;i++) //заполняем массив
  for(j=0;j<SIZE;j++)
     array[j][i]=k++;

   save_result("in.txt"); //сохраняем результат в файл
```

```
  for(r=0;r<nproc;r++) //рассылаем задания
  {
 if(r==myrank) continue;
 if(r==nproc-1 && nproc>2)
    tmp_dim = SIZE-(nproc-2)*tmp_dim;

 tmp_size      = tmp_dim*SIZE*sizeof(int);
 tmp_arr             = malloc(tmp_size);
 tmp_end             = tmp_begin+tmp_dim;

 for(i=tmp_begin,k=0;i<tmp_end;i++)
    for(j=0;j<SIZE;j++)
         tmp_arr[k++]=array[j][i];

 MPI_Send(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD);
 MPI_Send(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD);
 MPI_Send(tmp_arr,tmp_size/sizeof(int),MPI_INT,r,2,MPI_COMM_WORLD);

 printf("Master sent (%d-%d) to slave rank %d.\n",tmp_begin,tmp_end,r);

 free(tmp_arr);
 tmp_begin = tmp_end;
  }

  for(r=0;r<nproc;r++) //получаем ответы
  {
 if(r==myrank) continue;
 printf("Master waiting for slave rank %d...\n",r);
 MPI_Recv(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD,&status);
 MPI_Recv(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD,&status);
 MPI_Recv((int*)array+(tmp_begin*SIZE),(tmp_end-
tmp_begin)*SIZE,MPI_INT,r,2,MPI_COMM_WORLD,&status);
 printf("Master received (%d-%d) from slave rank %d.\n",tmp_begin,tmp_end,r);
  }

  save_result("out.txt"); //сохраняем результат в файл
  printf("Master ended.\n");
  MPI_Finalize();
  return 0;
}

void save_result(char * fname)
{
  int i,j;
  FILE* fd = fopen(fname,"w");
  if(!fd)
  {
 fprintf(stderr,"Error opening file for writing result.\n");
 MPI_Abort(MPI_COMM_WORLD,errno);
  }
  for(j=0;j<SIZE;j++)
  {
 for(i=0;i<SIZE;i++)
```

```
            fprintf(fd,"%d ",array[i][j]);
    fprintf(fd,"\n");
      }
      fclose(fd);
    }
```

Файл slave.c

```
#include "shared.h"

int main(int argc, char *argv[])
{
    int myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Slave rank %d started.\n",myrank);

    MPI_Status status;
    int master;
    int tmp_begin;
    int tmp_end;
    //получаем данные

MPI_Recv(&tmp_begin,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
    master = status.MPI_SOURCE;
    printf("Slave rank %d has found master rank %d.\n",myrank,master);
    MPI_Recv(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD,&status);
    int tmp_size = (tmp_end-tmp_begin)*SIZE*sizeof(int);
    int * tmp_arr = malloc(tmp_size);
    MPI_Recv(tmp_arr,tmp_size,MPI_INT,master,2,MPI_COMM_WORLD,&status);
    printf("Slave rank %d has received data from master.\n",myrank);
    ///////////////////////
    //делаем с ними НИЧЕГО//
    ///////////////////////
    //посылаем обратно
    MPI_Send(&tmp_begin,1,MPI_INT,master,0,MPI_COMM_WORLD);
    MPI_Send(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD);
    MPI_Send(tmp_arr,tmp_size/sizeof(int),MPI_INT,master,2,MPI_COMM_WORLD);
    printf("Slave rank %d has sent data to master and ended.\n",myrank);
    free(tmp_arr);
    MPI_Finalize();
    return 0;
}
```

Задание №3. Написать реализацию набора конфигурационных параметров для многих одновременно работающих экземпляров программы. Набор параметров задается некоторой структурой данных и хранится в разделяемой памяти. Блок разделяемой памяти создается при запуске первого экземпляра программы и заполняется из файла. Перед окончанием работы последнего экземпляра набор параметров сохраняется в файл, а блок разделяемой памяти удаляется. Программа должна обеспечивать основные функции работы с набором параметров (прочитать и изменить элемент), причем в случае изменения данных одним из экземпляров, он оповещает все остальные экземпляры с помощью сигнала.

Алгоритм

Первый из запущенных процессов считывает из файла input.txt параметры – числа epsilon, n, x. Epsilon – требуемая точность, n – степень, x – начальное значение. Программа

решает задачу нахождения lim(x^-n). Точное решение всегда 0. Далее, первый процесс рассылает остальным сообщение о том, что его копия конфигурационных параметров изменилась.

С этого места все процессы действуют одинаково:

Если процесс получает сообщение о том, что у другого процесса есть новая версия данных, он отсылает этому другому сообщение, что хочет их получить и получает.

Если процесс получает сообщение о том, что у него хотят получить данные - он отсылает их тому процессу, который их запросил. Тому, кто запросил их первым, отсылается сообщение начать их пересчёт.

Если процессу пришло сообщение начать пересчёт, он увеличивает значение x и считает новое значение функции x^-n. Если разность между значением до увеличения x и после него меньше epsilon, то выдаётся сообщение что предел найден и отсылается сообщение всем процессам, чтобы те завершали работу. Если разница больше либо равна epsilon, то процесс отсылает всем другим сообщение о том, что его параметры обновились (x=x+1).

Текст программы

(авторы: студенты 243 г. ИМКН Кожевников С.А., Беляев И.В., 2005г.)

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>
//пусть надо найти lim(x_i^-n) с точностью до epsilon,
//где x,n,epsilon заданы во входном файле
//точное решение всегда 0
struct cfg {
      float epsilon;  //точность, с которой мы ищем предел последовательности
      unsigned int n; //степень
      float x;        //начальное значение x_0
   } config;          //структура с "настройками"
//типы сообщений
//запрос
#define TAG_QRY 0
//данные
#define TAG_DAT 1
//сообщения:
//запрос на получение конфигурации
#define MSG_ASK_CONFIG  0
//сообщение об изменении конфигурации
#define MSG_CHG_CONFIG  1
//сигнал завершения процессов
#define MSG_KILL        2
//сигнал запуска вычислений
#define MSG_START       3
int nproc;      //кол-во процессов
int read_config() //читаем настройки из файла
{
   int retval=0;
   FILE* fd = fopen("input.txt","r");
   if(fd)
   {
      fscanf(fd,"%f\n",&config.epsilon);
      fscanf(fd,"%u\n",&config.n);
      fscanf(fd,"%f",&config.x);
```

```
        retval=0;
        fclose(fd);
     } else retval=1;
     return retval;
  }
  float next(unsigned int n,float x) //считаем следующее значение x
  {
     return (float)1/pow(x,n);
  }
  void killall(int rank) //говорим процессам, что пора и честь знать
  {
     int r,message=MSG_KILL;
     for(r=0;r<nproc;r++)
        if(r!=rank)
           MPI_Send(&message,1,MPI_INT,r,TAG_QRY,MPI_COMM_WORLD); //посылаем
сообщение о завершении работы
  }
  void say(int rank,char * msg)
  {
     printf("Процесс %d: %s.\n",rank,msg);
     return;
  }

  void msg_loop(int rank,int nproc)
  {
     say(rank,"жду сообщений");
     int r,message,sender,first=1;
     MPI_Status status;

     while(1)
     {

MPI_Recv(&message,1,MPI_INT,MPI_ANY_SOURCE,TAG_QRY,MPI_COMM_WORLD,&stat
us); //получаем сообщение
        sender=status.MPI_SOURCE; //определяем, от кого пришло сообщение
        if(sender!=rank)
        switch(message)
        {
           case                    MSG_CHG_CONFIG                    :
 //у кого-то данные изменились- заберём их у него
              say(rank,"получил сообщение MSG_CHG_CONFIG");
              message=MSG_ASK_CONFIG;
              MPI_Send(&message,1,MPI_INT,sender,TAG_QRY,MPI_COMM_WORLD);
//посылаем запрос
              MPI_Recv(&config,sizeof(struct
cfg),MPI_BYTE,sender,TAG_DAT,MPI_COMM_WORLD,&status); //получаем настройки
              say(rank,"получил новые данные");
              break;
           case MSG_ASK_CONFIG : //у нас попросили выслать данные
              MPI_Send(&config,sizeof(struct
cfg),MPI_BYTE,sender,TAG_DAT,MPI_COMM_WORLD); //посылаем настройки
              if(first)
              {
                 first=0;
```

```
                message=MSG_START;
                MPI_Send(&message,1,MPI_INT,sender,TAG_QRY,MPI_COMM_WORLD);
                say(rank,"отослал данные");
            }
            break;
        case MSG_START      : //пересчитываем
            say(rank,"пересчитываю");
            float nextx = next(config.n,config.x+1);
            if((next(config.n,config.x) - nextx)<config.epsilon)
            {
                char *msg = malloc(128);
                killall(rank); //убиваем все процессы, кроме себя
                sprintf(msg,"предел последовательности найден: %.5f",nextx);
                say(rank,msg);
                return;
            } else
            {
                config.x += 1;
                message=MSG_CHG_CONFIG;
                for(r=0;r<nproc;r++)
                    if(r!=rank)
                        MPI_Send(&message,1,MPI_INT,r,TAG_QRY,MPI_COMM_WORLD);
                first=1;
            }
            break;
        case MSG_KILL       : return;
        default             :
                fprintf(stderr,"Неизвестное сообщение.\n");
                return;
      }
    }
}


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

    int rank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==0) //первый процесс
    {
        if(read_config()==1) //загружаем данные
        {
            fprintf(stderr,"Не могу открыть файл.\n");
            exit(1);
        }
        int r,message=MSG_CHG_CONFIG;
        for(r=1;r<nproc;r++)
            MPI_Send(&message,1,MPI_INT,r,TAG_QRY,MPI_COMM_WORLD); //посылаем
всем процессам сообщение, что у нас есть данные
    }
```

```
    msg_loop(rank,nproc);   //главная процедура, начиная отсюда процессы равноправны
    MPI_Finalize();
    return 0;
}
```

Задание №4. Написать MPI подпрограмму, заменяющую каждый элемент массива вещественных чисел на среднее арифметическое его соседей (если это возможно). При этом должна быть обеспечена равномерная загрузка всех процессов.

Алгоритм

Программа состоит из двух ветвей – master и slave.

Ветвь master генерирует массив псевдослучайных чисел (размер массива задаётся константой SIZE) и отсылает равные части этого массива процессам slave. Последняя часть может быть иного размера, если размер массива не делится нацело на количество процессов slave. Затем master поочерёдно ожидает каждый процесс slave и получает результат его работы. Когда master получит данные от всех процессов slave, он выводит результат на экран.

Ветвь slave ожидает часть массива от процесса master. После получения пересчитывает значения так, чтобы все значения кроме первого и последнего были средним арифметическим соседних в исходном массиве. Затем slave отсылает данные обратно в процесс master и завершает работу.

Текст программы

(авторы: студент 243 г. ИМКН Соловей А.А., 2005г.)

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>

// размер массива
#define SIZE 32

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

    int rank,nproc;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc); //определяем кол-во процессов
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);  //определяем наш ранг (номер)
    MPI_Status status;
    double * tmp_mas;


    if(rank)
    {   //дочерний процесс
        int i,tmp_size;
        double j,k;

        MPI_Probe(0,0,MPI_COMM_WORLD,&status);
//ждём, пока нам пришлют часть массива
        MPI_Get_count(&status,MPI_DOUBLE,&tmp_size);//определяем его размер
        tmp_mas  = (double *)malloc(tmp_size*sizeof(double));
        MPI_Recv(tmp_mas,tmp_size,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
        j=tmp_mas[0];
```

```
    for(i=1;i<tmp_size-1;i++) //заменяем числа в массиве средним //арифметическим их соседей
      {
        k=tmp_mas[i];
        tmp_mas[i]=(j+tmp_mas[i+1])/2.0;
        j=k;
      }

    MPI_Send(tmp_mas,tmp_size,MPI_DOUBLE,0,1,MPI_COMM_WORLD);
// отсылаем результат
  }
  else
  {  //процесс мастер
    double mas[SIZE];
    int i,j,k,tmp_size;
    srand(time(NULL));
    for(i=0;i<SIZE;i++)
   mas[i]=((double)rand()/(double)RAND_MAX)*10; //заполняем массив
    printf("Исходный массив: ("); // вывод массива на экран
    for(i=0;i<SIZE-1;i++)
      printf("%.2f, ",mas[i]);
    printf("%.2f)\n",mas[SIZE-1]);

    for(i=1,k=0;i<nproc;i++)      //цикл по дочерним процессам
    {
      if(nproc>1 && i<(nproc-1))
     //определяем размер посылаемой части
        tmp_size = SIZE/(nproc-1);
      else
        tmp_size = SIZE-(tmp_size*(nproc-2));

      tmp_mas = (double *)malloc(tmp_size*sizeof(double));
//выделяем память под временный массив

      for(j=k;j<(k+tmp_size);j++)  //заполняем временный массив
        tmp_mas[j-k]=mas[j];
      k = k + tmp_size;

      MPI_Send((double *)tmp_mas,tmp_size,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
 //посылаем часть массива дочернему процессу
      printf("Отослал %d чисел процессу %d.\n",tmp_size,i);
      free(tmp_mas);
    }

    for(i=1;i<nproc;i++)
    {
        MPI_Probe(i,1,MPI_COMM_WORLD,&status);
       //ждём, пока нам пришлют часть массива
        MPI_Get_count(&status,MPI_DOUBLE,&tmp_size);    //определяем его размер
        MPI_Recv((double*)mas+(i-1)
*(SIZE/nproc),tmp_size,MPI_DOUBLE,i,1,MPI_COMM_WORLD,&status);
        printf("Получил %d чисел от процесса %d.\n",tmp_size,i);
    }
```

```
    printf("Получен массив: ("); //вывод массива на экран
    for(i=0;i<SIZE-1;i++)
        printf("%.2f, ",mas[i]);
    printf("%.2f)\n",mas[SIZE-1]);

  }

  MPI_Finalize();
  return 0;
}
```

Задание №5. Написать программу, вычисляющую сумму элементов n последовательностей вещественных чисел, находящихся в n файлах, имена которых заданы массивом a. Ответ должен быть записан в файл res.txt. Программа запускает n процессов, вычисляющих ответ для каждого из файлов и прибавляющих его к результату, находящемуся в выходном файле.

Входные данные

1. Последовательность текстовых файлов. Каждый файл содержит последовательность целых чисел, записанных в столбец. Каждое число находится в интервале от – (232) до +(232)-1. Все файлы находятся в той же директории, что и исполняемый файл программы.

2. Множество имён текстовых файлов, содержащих целые числа. Их количество не должно превышать 216.

Выходные данные

1. Множество процессов, запущенных на данный момент.

2. Текстовое поле, отражающее содержимое виртуального файла, содержащего результат суммирования всех чисел.

Формулы связи

Считывается очередной элемент массива имён текстовых файлов. Для полученного файла заводится отдельный процесс, который начинает выполняться параллельно с уже заведёнными. При этом, увеличивается значение текстового поля, отражающего количество запущенных в текущий момент процессов. Такой процесс открывает полученный текстовый файл на чтение и начинает извлекать из него целые числа, попутно проверяя их на корректность и суммируя в своём хранилище. После того, как очередной процесс завершается, происходит вызов функции осуществляющей суммирование полученных значений сумм чисел, хранящихся в хранилищах процессов. На момент записи нового значения, вместо уже имеющегося в качестве результата числа, происходит блокирование всех хранилищ, из которых экспортируются рассчитываемые процессами суммы чисел. Т.о. это позволяет избежать попытки обращения процедуры финального суммирования к ещё не полученному результату суммирования чисел какого-либо из запущенных процессов. После того, как очередной процесс завершается, происходит уменьшение на единице значение текстового поля, отражающего количество запущенных в текущий момент процессов.

Текст программы

(авторы: студенты 243 г. Торгунский В.С., Святенко А.А., 2005г.)

```
unit Unit1; ()
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
//Наследуем потоковый класс
  TPTh=class(TThread)
  private
     name:string; // Имя файла, для которого заводится поток
  protected
```

```
    //описание методов ниже.
    procedure Execute; override;
    procedure calculate;
  public
    constructor create1(name:string;flag:boolean);
  end;
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    GroupBox1: TGroupBox;
    Memo1: TMemo;
    GroupBox2: TGroupBox;
    Edit2: TEdit;
    GroupBox3: TGroupBox;
    Label3: TLabel;
    Edit3: TEdit;
    Button2: TButton;
    //обработчик нажатия на кнопку запуска
    procedure Button1Click(Sender: TObject);
    //процедура суммирования всех чисел
    procedure Summiging;
    procedure FormCreate(Sender: TObject);
    //обработчик нажатия на кнопку выхода
    procedure Button2Click(Sender: TObject);
    private
      //объект потокового класса.
      PT:TPTh;
  public
  end;
var
  Form1: TForm1;
  // множество результатов потоков
  arr:array of int64;
  // множество имён файлов.
  arrfiles:array of string;
implementation
{$R *.dfm}
```

//конструктор для объектов потокового класса. На вход идут имя файла и флаг, //отвечающий за мгновенное или не мгновенное включение потокапараллельно с //остальными.

```
constructor TPTh.create1(name:string;flag:boolean);
begin
  self.name:=name;
  inherited Create(flag);
end;
```

//Реализация процесса чтения из файла чисел, их суммирования и занесения в //хранилище. //Как видим, предпоследней строкой мы останавливаем поток и лишь тогда заносим //результат в хранилище. Избегаем тем самым ошибок незавершённости потока. //(обращение к уже освобождённой памяти)

```
procedure TPTH.Calculate;
var f:textfile;
  str:string;
  buf,int1:int64;
```

```
begin
   buf:=0;
   setlength(arr,length(arr)+1);
   assignfile(f,name);
   reset(f);
     while not(eof(f)) do
     begin
       readln(f,str);
       if not(trystrtoint64(str,int1)) then Messagedlg(' Один из файликов,а именно '+name+',
содержит запись, отличную отчисловой',mtInformation,[mbOk],0) else
       buf:=buf+int1;
     end;
   closefile(f);
   self.Terminate;
   arr[length(arr)-1]:=buf;
end;

procedure TPth.Execute;
begin
   repeat
//защищаем нашу память от доступа сразу нескольких потоков к одной и той же //ячейке!!!
     synchronize(calculate);
   until Terminated;
   form1.summiging;
   //по завершении потока суммируем.
   form1.Edit3.Text:=inttostr(strtoint(form1.edit3.Text)-1);
end;

//обработка нажатия на стартовую кнопку РАСЧИТАТЬ
procedure TForm1.Button1Click(Sender: TObject);
var i:word;
begin
  //Расчитали, сколько потоков нужно будет заводить (колько файлов).
  //Завели нужное хранилище!
  setlength(arrfiles,memo1.Lines.Count);
  for i:=0 to memo1.Lines.Count-1 do
  begin
    edit3.Text:=inttostr(1+strtoint(edit3.Text));
    PT:=TPTH.Create1(memo1.Lines[i],false);
  end;

end;
//Финальная процедура расчёта результата. Смотрим на хранилище и делаем //последнее
суммирование.
procedure TForm1.Summiging;
var i:word;
begin
  for i:=0 to length(arr)-1 do
edit2.Text:=inttostr(strtoint64(edit2.Text)+arr[i]);
end;
//Обрабока стартовой функции
procedure TForm1.FormCreate(Sender: TObject);
begin
 memo1.Text:='';
```

```
end;
//Обработка нажатия на Выход.
procedure TForm1.Button2Click(Sender: TObject);
begin
form1.Close;
end;
```

Задание №6. Написать multithread-функцию, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (thread) p, и заменяющую матрицу a на матрицу (a+at)/2 (1), где at – транспонированная матрица a. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

Однопоточный алгоритм

С помощью двух вложенных циклов пройтись по всем элементам матрицы a и матрицы at, складывая их. А затем сумму элементов разделить на два.

Многопоточный алгоритм

Задача разбивается на n атомарных задач, каждая из которых состоит в том, чтобы вычислить значение выражения (aij+aji)/2, для n2/p, где aij – элементы входной матрицы, n – количество элементов в строке или столбце матрицы, p – количество процессов, запущенных для вычисления выражения (1).

Внутренние спецификации

void save_result(char * fname);

Сохраняет в файл fname матрицу array[n][n].

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

Сначала инициализирует матрицу a порядка n следующим образом: array[i][j] = i+j-1. Затем, сохраняет полученную матрицу в файл. Разделяет задачу между процессами, которые выполняют подзадачи. Затем получает результаты от каждого процесса и формирует общий результат, а затем сохраняет его в файл.

MPI_Status status; Переменная для хранения статуса приема

int tmp_dim = SIZE/(nproc-1); Переменная для хранения количества элементов, которые будет переданы в каждый процесс

int tmp_begin=0; Переменная для хранения индекса начала массива, передаваемого в процесс

int tmp_end; Переменная для хранения индекса конца массива, передаваемого в процесс

int * tmp_arr; Переменная для хранения всех элементов матрицы размерности nxn в виде вектора размерности n2

int tmp_size; Переменная для хранения объёма памяти необходимого для хранения массива, передаваемого в процесс

Текст программы

(авторы: студенты 243 г. ИМКН Анисимова Я.П., Селиверстова Ю.А., 2005г.)

```
#include "shared.h"
int main(int argc, char *argv[])
{
   int myrank;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
   printf("Slave rank %d started.\n",myrank);
   MPI_Status status;
   int master;
   int tmp_begin;
   int tmp_end;
```

```
    //получаем данные

MPI_Recv(&tmp_begin,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
    master = status.MPI_SOURCE;
    printf("Slave rank %d has found master rank %d.\n",myrank,master);

    MPI_Recv(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD,&status);

    int tmp_size = (tmp_end-tmp_begin)*SIZE*sizeof(int);
    int * tmp_arr = malloc(tmp_size);

    MPI_Recv(tmp_arr,tmp_size,MPI_INT,master,2,MPI_COMM_WORLD,&status);
    printf("Slave rank %d has received data from master.\n",myrank);
    //посылаем обратно
    MPI_Send(&tmp_begin,1,MPI_INT,master,0,MPI_COMM_WORLD);
    MPI_Send(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD);
    MPI_Send(tmp_arr,tmp_size/sizeof(int),MPI_INT,master,2,MPI_COMM_WORLD);

    printf("Slave rank %d has sent data to master and ended.\n",myrank);
    free(tmp_arr);
    MPI_Finalize();
    return 0;
  }

  #include "shared.h"
  int array[SIZE][SIZE];
  void save_result(char * fname);
  int main(int argc, char *argv[])
  {
    int myrank,nproc;
  // инициализация MPI
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Master started. My rank is %d, npr is %d\n",myrank,nproc);

    MPI_Status status;
    int tmp_dim = SIZE/(nproc-1);
    int tmp_begin=0;
    int tmp_end;
    int * tmp_arr;
    int tmp_size;
    int i,j,k;
    int r;

   for(i=0,k=0;i<SIZE;i++) for(j=0;j<SIZE;j++) array[j][i]=k++;//заполняем массив
   save_result("in.txt"); //сохраняем результат в файл
   for(r=0;r<nproc;r++){ //рассылаем задания
          if(r==myrank) continue;
          if(r==nproc-1 && nproc>2)tmp_dim = SIZE-(nproc-2)*tmp_dim;
          tmp_size      = tmp_dim*SIZE*sizeof(int);
          tmp_arr               = malloc(tmp_size);
          tmp_end               = tmp_begin+tmp_dim;
```

```
            for(i=tmp_begin,k=0;i<tmp_end;i++)
for(j=0;j<SIZE;j++)tmp_arr[k++]=(array[j][i]+array[i][j])/2;
            MPI_Send(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD);
            MPI_Send(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD);
            MPI_Send(tmp_arr,tmp_size/sizeof(int),MPI_INT,r,2,MPI_COMM_WORLD);
            printf("Master sent (%d-%d) to slave rank %d.\n",tmp_begin,tmp_end,r);
            free(tmp_arr);
            tmp_begin = tmp_end;
        }

        for(r=0;r<nproc;r++){ //получаем ответы
            if(r==myrank) continue;
            printf("Master waiting for slave rank %d...\n",r);
            MPI_Recv(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD,&status);
            MPI_Recv(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD,&status);
            MPI_Recv((int*)array+(tmp_begin*SIZE),(tmp_end-
tmp_begin)*SIZE,MPI_INT,r,2,MPI_COMM_WORLD,&status);
            printf("Master received (%d-%d) from slave rank %d.\n",tmp_begin,tmp_end,r);
        }
        for(i=0,k=0;i<SIZE;i++) for(j=0;j<SIZE;j++) array[j][i]=array[i][j];
        save_result("out.txt"); //сохраняем результат в файл
        printf("Master ended.\n");
        MPI_Finalize();
        return 0;
    }
    void save_result(char * fname)
    {
        int i,j;
        FILE* fd = fopen(fname,"w");
        if(!fd)
        {
       fprintf(stderr,"Error opening file for writing result.\n");
       MPI_Abort(MPI_COMM_WORLD,errno);
        }
        for(j=0;j<SIZE;j++)
        {
       for(i=0;i<SIZE;i++)   fprintf(fd,"%d ",array[i][j]);
       fprintf(fd,"\n");
        }
        fclose(fd);
    }
```

Задание №7.

Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, являющееся длиной этого массива, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент массива (для которого это возможно) на среднее арифметическое сосежних элементов. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

Алгоритм.

Основная программа делит входной массив (задается по формуле) на k-1 частей, - каждая часть состоит из n/(k-1) столбцов (деление нацело) и n строк, а последняя (k-1) часть из n- n/(k-1)* (k-2) столбцов и n строк.

Затем каждая из полученных частей массива поочередно записывается в одномерный массив по столбцам (сверху-вниз и слева направо) и отсылается соответствующему процессу. После того, как все части разосланы, главная программа начинает получать результаты обратно от подчиненных процессов в том же порядке, в котором части были разосланы. Полученные части записываются построчно в массив . Массив сохраняется в файл ‘out.txt’. Подчиненные процессы просто получают блок данных и отсылают его обратно.

Текст программы.

(авторы: студенты 243 г. Кузнецов И.В., Николайчук Е.Н., 2005г.)

```
#include "shared.h"
int array[SIZE];
void save_result(char * fname);
int main(int argc, char *argv[])
{
   int myrank,nproc;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);
   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
   printf("Master started. My rank is %d, npr is %d\n",myrank,nproc);

   MPI_Status status;
   int tmp_dim = SIZE/(nproc-1);
   int tmp_begin=0;
   int tmp_end;
   int * tmp_arr;
   int tmp_size;
   int i,j,k;
   int r;

   for(i=0;i<SIZE;i++)
  array[i]=(i+1)*3;

   save_result("in.txt");

   for(r=0;r<nproc;r++)
    {
  if(r==myrank) continue;
  if(r==nproc-1 && nproc>2)
     tmp_dim = SIZE-(nproc-2)*tmp_dim;

  tmp_size        = tmp_dim;
  tmp_arr                  = malloc(tmp_size);
  tmp_end                  = tmp_begin+tmp_dim;

//printf("begin = %d \n", tmp_begin);
//printf("end = %d \n", tmp_end);
//printf("size = %d \n", tmp_size);
//printf("Sendet array to rank %d [",r);
```

```
  for(i=tmp_begin,k=0;i<tmp_end;i++){
          tmp_arr[k++]=array[i];
//        printf("%d, ",array[i]);
  }
//printf("] \n");
  MPI_Send(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD);
  MPI_Send(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD);
  MPI_Send(tmp_arr,tmp_size,MPI_INT,r,2,MPI_COMM_WORLD);

  printf("Master sent (%d-%d) to slave rank %d.\n",tmp_begin,tmp_end,r);

  tmp_begin = tmp_end;
   }

   for(r=0;r<nproc;r++)
   {
  if(r==myrank) continue;
  printf("Master waiting for slave rank %d...\n",r);
  MPI_Recv(&tmp_begin,1,MPI_INT,r,0,MPI_COMM_WORLD,&status);
//printf(" yo1 = %d\n", tmp_begin);
  MPI_Recv(&tmp_end,1,MPI_INT,r,1,MPI_COMM_WORLD,&status);
//printf(" \n yo2 \n", tmp_end);
  MPI_Recv(((int*)array+tmp_begin),(tmp_end-
tmp_begin),MPI_INT,r,2,MPI_COMM_WORLD,&status);
//printf(" \n yo3 [");

//for(i=0;i<SIZE;i++){
//        printf("%d, ",array[i]);
//}
//printf("] \n");

  printf("Master received (%d-%d) from slave rank %d.\n",tmp_begin,tmp_end,r);
   }
   save_result("out.txt");
   printf("Master ended.\n");
   MPI_Finalize();
   return 0;
}

void save_result(char * fname)
{
   int i;
   FILE* fd = fopen(fname,"w");

   if(!fd){
  fprintf(stderr,"Error opening file for writing result.\n");
  MPI_Abort(MPI_COMM_WORLD,errno);
   }

   fprintf(fd,"Array: [");
   for(i=0;i<SIZE;i++){
  fprintf(fd,"%d, ",array[i]);
   }
   fprintf(fd,"]\n");
```

```
    fclose(fd);
    return 0;
}
#include "shared.h"

int main(int argc, char *argv[])
{
    int myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Slave rank %d started.\n",myrank);

    MPI_Status status;
    int master;
    int tmp_begin;
    int tmp_end;


MPI_Recv(&tmp_begin,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);

    master = status.MPI_SOURCE;
    printf("Slave rank %d has found master rank %d.\n",myrank,master);
    MPI_Recv(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD,&status);

    int tmp_size = (tmp_end-tmp_begin);
    int * tmp_arr = malloc(tmp_size);

    MPI_Recv(tmp_arr,tmp_size,MPI_INT,master,2,MPI_COMM_WORLD,&status);
    printf("Slave rank %d has received data from master.\n",myrank);


    int i;
//    printf("Prinyal array from %d \n [",myrank);
//    for(i=0; i<tmp_size; i++){
//printf("%d, ", tmp_arr[i]);
//    }
//    printf("]\n");

//    printf("TMP_BEGIN = %d \n",tmp_begin);
//    printf("TMP_END = %d \n",tmp_end);
//    printf("TMP_SIZE = %d \n",tmp_size);

    for(i=0; i<tmp_size; i++){
   tmp_arr[i]=tmp_arr[i]-5;
    }

//    printf("Preobrazoval array from %d \n [",myrank);
//    for(i=0; i<tmp_size; i++){
//printf("%d, ", tmp_arr[i]);
//    }
//    printf("]\n");
```

```
    MPI_Send(&tmp_begin,1,MPI_INT,master,0,MPI_COMM_WORLD);
    MPI_Send(&tmp_end,1,MPI_INT,master,1,MPI_COMM_WORLD);
    MPI_Send(tmp_arr,tmp_size,MPI_INT,master,2,MPI_COMM_WORLD);

    printf("Slave rank %d has sent data to master and ended.\n",myrank);
    free(tmp_arr);
    MPI_Finalize();
    return 0;
}
```

Часть 3. Правила для работы в среде MPI

1. Архитектура кластера

В ВЦКП ДВО РАН (ИАПУ ДВО РАН) имеется ряд вычислительных кластеров различной архитектуры. Одним из них является кластер МВС-1000/16, состоящий из 16 однопроцессорных узлов, соединенных двумя Fast Ethernet сетями (по 100Мбит каждая). На каждом узле установлен процессор Pentium-III 800MHz и память 512Мб. Рабочая частота системной шины составляет 133MHz. Операционная система – Linux. Данная вычислительная машина предназначена для решения несложных вычислительных задач и использования в учебных целях для практических и лабораторных работ.

Один из узлов кластера является главным (головной узел). Он доступен из внешней (по отношению к кластеру) сети, предназначен для хранения всех настроек и файлов пользователей, отвечает за авторизацию пользователей во всей системе и запускает все необходимые серверные компоненты для поддержания основных сервисов: SSH (удаленный командный и файловый доступ) и NFS (сетевая файловая система).

На кластере пользователю для работы доступны следующие сервисы:

- Командный интерфейс (работа в режиме удаленного терминала) для удаленного запуска команд на кластере.
- В рамках стандартного набора командного языка sh доступны команды по управлению файлами, например, копирование, удаление, переименование и т.п.
- Файловый интерфейс для копирования исходных текстов программ и получения файлов результатов (используется SFTP).
- Сервис компиляции для создания исполняемых модулей параллельных MPI-программ.
- Сервис запуска параллельной MPI-программы на кластере.
- Сервис отладки в рамках возможностей используемой реализации стандарта MPI (на кластере используется пакет LAM 7.0).

1.1. Командный интерфейс

Для доступа на узловую машину кластера можно использовать любой ssh-совместимый клиент (рекомендуется использовать пакет PuTTY, который является свободно распространяемым и может быть бесплатно получен с сайта разработчиков http://www.putty.nl/ или локально с сервера сети ДВО РАН ftp://ftp.dvo.ru/limited/Win32-Soft/Internet/Utils/putty-0.58-installer.exe). Для доступа на кластер в рамках проведения лабораторных и практических работ необходимы следующие параметры. Имя шлюзовой машины доступа на кластер `mvs16.cc.dvo.ru`. Пользователь и пароль смотрите в Приложении 1. Тем, кто использует PuTTY необходимо выполнить следующие шаги. Открыть главное окно программы (см. рис.). В графе “Host Name” набрать `mvs16.cc.dvo.ru`. В графе “Saved Sessions” задать какое-нибудь удобное имя (например, mvs1000/16). Нажать Save. Данные настройки выполняются один раз на одной машине пользователя. В дальнейшем в списке будет имя mvs1000/16. Для запуска терминальной сессии с кластером выберите mvs1000/16 и нажмите Logon. (Для других терминал-клиентов

способ настройки может немного отличаться. Обратитесь к документации используемых программ). Также отметим, что слева можно произвести дополнительные сервисные настройки размера и типа шрифта, который будет использоваться для отображения передаваемой информации. **ВНИМАНИЕ! Приведенные настройки действительны в случае работы из учебных классов ИАПУ ДВО РАН. Для самостоятельной работы из дома (или других мест) используйте настройки, перечисленные в приложении 2.** После нажатия Logon откроется новое текстовое окно, в котором в левом верхнем углу будет написано

```
login as: _
```

Это приглашение для ввода параметров авторизации пользователя. В первую очередь необходимо ввести имя пользователя и нажать Enter, затем пароль (также нажать Enter)

```
login as: ivanov
Using keyboard-interactive authentication.
Password: _
```

ПРЕДУПРЕЖДЕНИЕ! Пароль при вводе не отображается! Если вы чувствуете, что ошиблись в написании, нажмите Ctrl+U для сброса и начните набор пароля заново. Также обращайте внимание на текущий выбранный язык ввода. Если Вы не правильно набрали пароль, то процесс ввода пользователя или пароля повторится. После 3-5 неправильных попыток терминальный сеанс будет прерван сервером и окно закроется. В этой ситуации нужно начать с самого начала.

В случае успешного ввода пароля Вам высветится приглашение сервера

```
ivanov@1[~]$
```

После успешного входа, система переводит Вас в домашний каталог (физически он располагается в директории /home/FENU-IMCS243-XX-YY/USERNAME, где XX и YY – учебный год (08 – 2008 год), USERNAME – имя пользователя по приложению 1). После первого входа в систему рекомендуем сменить пароль (см. далее раздел 2).

2. Основные команды Linux

Следует отметить, что все команды (а также имена файлов) в операционной системе Linux являются контекстно-зависимыми. То есть файлы my.cpp и MY.cpp будут совершенно разными файлами. Приведем список основных команд, которые могут понадобиться для выполнения практических и лабораторных работ.

2.1. Команда ls

Выводит содержимое каталогов и информацию об указанных файлах. Команда имеет формат

```
ls [option] [file]
```

Приведем некоторые опции (option):
`-l` – выводит детальное описание информации о файлах (размер, дата и т.п.)
`-h` – выводит размер файлов с суффиксом “M” (мегабайты)
`-R` – выводит информацию рекурсивно для всех поддиректорий начинаю с текущей

В качестве параметра file можно указать конкретный файл или использовать символы регулярных выражений. Например команда `ls *.cpp` отобразит имена всех cpp-файлов в текущем каталоге. Дополнительную информацию о других (многочисленных) опциях этой

команды можно прочитать во встроенной справочной системе. Вызов справки осуществляется командой

```
ivanov@1[~]$ man ls
```

Выход из листинга справки осуществляется нажатием на “q”. Аналогично справочную информацию можно получить и о других командах, рассмотренных далее.

2.2 Команда cd

Изменяет текущую директорию. Формат

```
cd <new-directory-name>
```

Аналогично краткую информацию о дополнительных опциях и правилах использования команды можно посмотреть во встроенной справочной системе (man cd).

2.3. Команда md

Создать новый каталог (по умолчанию не делает его текущим каталогом). Формат

```
md <new-directory-name>
```

2.4. Команда less

Просмотр содержимого файла. Формат

```
less <file-name>
```

При просмотре содержимого файла доступны следующие основные функции: “q” - выход, “/” - поиск подстроки в файле. Для поиска нажмите “/” внизу экрана появится символ “/” и курсор. Далее введите подстроку которую вы хотите найти и нажмите <Enter>. По тексту подсветятся найденные подстроки. Повторное нажатие на “/” и <Enter> приведет к поиску следующей заданной подстроки в файле начиная с уже найденной позиции.

2.5. Команда vim

Текстовый редактор. Формат

```
vim <file-name>
```

Данная команда используется для редактирования текстовых файлов и имеет собственный внутренний командный язык для редактирования текста. В данном языке предусмотрены: режим просмотра, режимы редактирования, режим выделения (мы приведем только некоторую небольшую часть возможностей vim; более подробную информацию смотрите в справочной системе man vim или на сайте разработчика http://www.vim.org/). В режиме просмотра vim работает аналогично команде less (просмотр и поиск). Переключение в режим редактирования из режима просмотра осуществляется нажатием на <i> (внизу слева появится --- INSERT ---). Выход из режима редактирования нажатием на ESC. Удаление строки осуществляется нажатием на <d> в режиме просмотра. Объединение строк осуществляется нажатием на <Shift+J>. Нажатие на <v> переводите в режим выделения текста (внизу слева появится ---VISUAL---). В режиме выделения нажатием на <d> - удаляет текст в буффер обмена; <y> - копирует текст в буфер обмена. Выход из режима выделения также ESC. Нажатие на <p> в режиме просмотра вставляет содержимое буфера обмена,

начиная со следующей строки от позиции курсора. Нажатие на <u> выполняет операцию отмены действий редактирования (аналогично команде Undo в редакторах операционной системы Microsoft Windows ®).

2.6. Команда cp

Копирование файла(ов). Формат команды

```
cp [options] <file1> <file2>
```

2.7. Команда mv

Перемещение файла(ов). Форма команды

```
cp [options] <file1> <file2>
```

2.8. Команда rm

Удаление файла.

```
rm [options] <file>
```

2.9 Команда passwd

Изменение пароля пользователя. Для смены собственного пароля наберите команду passwd без параметров. Вам будет предложено ввести новый пароль. По окончанию ввода система сообщит об успешном изменении пароля.

```
passwd [user-name]
```

3. Файловый интерфейс

Для обмена файлами между компьютером пользователя и кластером на кластере установлен и работает SFTP-сервер (Secured FTP). Используя любой совместимый SFTP-клиент можно соединиться с узловой машиной кластера и скопировать или переместить набор файлов или директорий. В службе поддержки ВЦКП ИАПУ ДВО РАН рекомендуют использовать пакет (бесплатно распространяемый) WinSCP (сайт разработчика: http://winscp.net/eng/index.php или локально в сети ДВО РАН ftp://ftp.dvo.ru/limited/Win32-Soft/Internet/Utils/winscp382setup.exe). Аналогично настройки командного интерфейса в параметрах данной утилиты необходимо указать адрес машины доступа к кластеру (`mvs16.cc.dvo.ru`) и параметры пользователя (см. Приложение 1; **для работы за пределами ИАПУ ДВО РАН см. комментарии в Приложении 2**). Аналогично многим другим утилитам, интерфейс winscp построен по принципу двух панелей. Левая панель – локальная машина пользователя. Правая панель – кластер. Выделение, копирование и перемещение файлов производится аналогично как в Far Manager ®, TotalCommander ® и других похожих системах.

4. Компиляция параллельных программ

Общим принципом сборки всех исполняемых модулей в среде UNIX является использование универсальной утилиты make. Входными данными для этой утилиты является файл (по умолчанию его имя Makefile), в котором расписан сценарий сборки исполняемых файлов из имеющихся файлов с исходным кодом. Принципы построения Makefile'ов достаточно сложны. Опишем только некоторые возможности, которые необходимы для

выполнения практических заданий (полная информация представлена http://www.gnu.org/software/make/).

Makefile представляет собой текстовый файл, который состоит из двух основных блоков. Блок деклараций и блок правил. В блоке деклараций объявляются необходимые переменные, которые требуются в процессе сборки исполняемых модулей. В блоке правил расписываются правила сборки исполняемых модулей или библиотек из набора имеющихся исходных файлов.

Объявление имеет следующую структуру:

```
Имя = значение
```

Правило имеет следующую структуру:

```
Имя_правила    : список зависимых правил или файлов
          Команды, которые необходимо выполнить, для выполнения \
          данного правила
```

Рассмотрим на примере (шаблон Makefile'а находится в домашнем каталоге группы на узловой машине кластера /home/FENU-IMCS243-YY-YY; имя файла Makefile.example). Пусть исходный текст нашей программы состоит из следующего набора файлов: main.cpp, file1.h, file1.cpp, file2.h, file2.cpp. Необходимо собрать исполняемый модуль example. Тогда структура Makefile'а для сборки исполняемого файла example может выглядеть следующим образом.

```
CPP=mpiCC

all:      example

example:  main.o file1.o file2.o
          $(CPP) -o example main.o file1.o file2.o

main.o:   main.cpp
          $(CPP) -c main.cpp

file1.o:  file1.cpp file1.h
          $(CPP) -c file1.cpp

file2.o:  file2.cpp file2.h
          $(CPP)-c file2.cpp
```

Некоторые комментарии. Конструкция $(имя_переменной) является подстановкой значения переменной. Переменная CPP задает компилятор, который используется для сборки файлов. Записанные правила в приведенном примере, определяют следующий порядок сборки.

```
mpiCC -c file1.cpp
mpiCC -c file2.cpp
mpiCC -c main.cpp
mpiCC -o example main.o file1.o file2.o
```

Строчки с 1 по 3 компилируют имеющиеся cpp-файлы. В результате будет создано три объектных файла (с расширением ".o"). Последняя строчка компонует все объектные файлы в единый исполняемый модуль с именем example.

Далее приводится еще один пример Makefile’а для сборки двух исполняемых файлов одновременно.

```
CPP=mpiCC

all:      master slave

master:   master.o common.o
          $(CPP) -o master master.o common.o

slave:    slave.o common.o
          $(CPP) -o slave slave.o common.o

master.o: master.cpp master.h
          $(CPP) -c master.cpp

slave.o:  slave.cpp slave.h
          $(CPP) -c slave.cpp

common.o: common.cpp common.h
          $(CPP) -c common.cpp
```

5. Запуск параллельной программы

Запуском параллельных программ на кластере управляет система очередей. Для каждой задачи, которая запсукается с использованием системы очередй, должен быть сформирован паспорт задачи, в котором указываются основные параметра запуска: название очереди, количесто параллельных процессов, название исполняемого файла и другие параметры. После этого паспорт задачи обрабатывается системой очередей и, таким образом, ваша задача принимается к исполнению.

В целях упрощения работы с системой очередей для вычислительных кластеров ИАПУ была разаработана вспомагательная утилита, автоматизирующая процессы создания папорта задачи и постановки задачи на исполнение. Название данной утилиты соответствует стандартному для всех реализаций MPI файлу **mpirun**. Текущая версия **mpirun** поддерживает следующие параметры запуска параллельных программ:

```
mpirun -np <num> <program> [<prog args>]
```

- <num> - количество экземпляров <program>, которые будут запущены параллельно на узлах кластера;
- <program> - название исполняемого файла параллельной программы;
- <prog args> - аргументы, командной строки, которые можно передать программе.

Работу с утилитой mpirun рассмотрим на примере. Предположим, что после компиляции мы имеем один исполняемый файл example. Также, по условиям решаемой задачи известно, что необходимо параллельно запустить 5 экземпляров example. Чтобы поставить такую задачу в очередь на исполнение, необходимо ввести команду

```
ivanov@mvs16 ~ $ mpirun -np 5 example
```

В результате выполнения этой команды на консоли появится следующая информация

```
Creating job description file example.job
61.mvs16.cc.dvo.ru
```

Первая строчка вывода, говорит о том, автоматически сгенерирован паспорт задачи – файл с именем example.job. Паспорт задачи содержит необходимую служебную информацию для системы очередей и содержит ряд служебных команд, предназанченных для организации корректного запуска параллельной программы и последующего ее завершения.

Вторая строчка вывода сообщает о том, что задача поставлена в очередь на исполнение и ей присвоен уникальный идентификатор 61.mvs16.cc.dvo.ru.

После того как задача поставлена в очередь, возможны два вариант развития событий:

1. если для выполнения параллельной программы достаточно свободных узлов, то задача немедленно начинает исполняться;
2. если количество свободных узлов кластера меньше запрошенного количества (меньше <num>), то задача будет ожидать в очереди до тех пор, пока не осободится достаточное количество узлов.

Для просмотра состояния узлов кластера, используйте команду **pbsnodes**. Для каждого узла кластера команда выдает следующую служебную информацию

```
mvs16-1
     state = job-exclusive
     np = 1
     ntype = cluster
     jobs = 0/65.mvs16.cc.dvo.ru
     status = opsys=linux,uname=Linux mvs16 2.6.20-gentoo-r8 #15 Thu Aug 16 14:57:13 VLAST 2007
       i686,sessions=31751       10800       12920       15647       17046       17395       19749
       21751,nsessions=8,nusers=7,idletime=57,totmem=1502004kb,availmem=832636kb,physmem=514016kb,n
       cpus=1,loadave=0.03,netload=248617508,state=free,jobs=65.mvs16.cc.dvo.ru,rectime=1203494086
```

- state – состояние узла кластера может принимать следующие значения;
    - job-exclusive – занят под задачу и не может использоваться для других задач;
    - free – свободен и в текущий момент времени может быть использован для запуска задач;
- np – число физических процессоров узла;
- ntype = cluster – узел используется для решения прикладных задач в рамках кластера;
- jobs – идентификатор задачи, которая на нем исполняется;
- status – системная информация.

Для просмотра состония очереди, используйте команду **qstat -a**. В результате отобразится информация примерно следующего содержания

```
ivanov@mvs16 ~ $ qstat -a

mvs16.cc.dvo.ru:
                                                                   Req'd  Req'd   Elap
Job ID               Username Queue    Jobname    SessID NDS   TSK Memory Time  S Time
-------------------- -------- -------- ---------- ------ ----- --- ------ ----- - -----
65.mvs16.cc.dvo.ru   ivanov   mvs16    example1.j    --      7  --    --     -- R   --
66.mvs16.cc.dvo.ru   petrov   mvs16    example2.j    --      7  --    --     -- Q   --
```

В примере показано, что сейчас в очереди задач находится две задачи. Первая колонка отображает идентификатор задачи. Вторая колонка – имя пользователя. Третья колонка – название очереди задач. Четвертая колонка – название задачи. NDS – количество задействованных узлов. S – состояние задачи (R – исполняется; Q – нахоится в очереди ожидания).

После того как задача завершилась, она удаляется из очереди, и в текущей директории пользователя создается два служебных файла. Названия данных файлов имеют следующий шаблон

```
<program>.job.o<jobid>
<program>.job.e<jobid>
```

Для рассмотренного выше примера имена файлов выглядят example.job.o61 и example.job.e61. Первый файл содержит всю информацию, которую процессы выводили на консоль. Второй файл содержит всю информацию об ошибках, происходивших в процессе выполнения параллельной программы.

6. Рекомендации по отладке программ

6.1. Применение printf()

Самый примитивный (самый трудоемкий) способ отладки программ. Весь вывод на экран всех запущенных экземпляров программы (процессов) перенаправляется на Вашу консоль. Обязательно указывайте в выводимой строке ранг процесса, чтобы понимать от кого именно данное сообщение. Например, фрагмент кода

```
…
MPI_Comm_rank(MPI_COMM_WOLRD, &my_rank);
…
printf("rank=%i; Start sending MSG_TAG to %i\n", my_rank,
rank_to);
MPI_Send(&buf, buf_len, MPI_CHAR, rank_to, MSG_TAG,
MPI_COMM_WORLD);
…
```

6.2. Утилита jumpshot

JUMPSHOT - удобное средство отладки и анализа производительности параллельных программ. Основная функция данной утилиты – визуализировать порядок выполнения MPI-вызовов в параллельной программе с привякой ко времени выполнения. Данная утилита входит в состав пакета MPICH, который установлен на кластере МВС1000/16 и используется для написания и запуска параллельных программ. Суть работы jumpshot такой же, как и применением printf'ов. В моменты выполнения MPI-вызовов в специальный файл (log-файл) заносится основная информация о типе вызова, характере передаваемых данных и времени выполнения вызова. В результате по окончанию работы параллельной программы будет сформирован log-файл, в котором в специальном формате хранится все статистика обмена сообщениями.

Графическая часть утилиты jumpshot использует технологию X-Window для отображения информации на экране и взаимодействия с пользователем (см. рис. 1). Для запуска утилиты на рабочем компьютере потребуется установить какую-либо совместимую версию X-Server'а. Пользователям Microsoft Windows ® рекомендуется бесплатный пакет X-ming (его можно загрузить по адресу ftp://ftp.dvo.ru/pub/win/Xming/Xming-mesa-6-9-0-31-setup.exe). Установку пакету следует осуществить с указанием всех параметров по умолчанию (просто нажимайте «Далее»). По умолчанию установка производится в директорию C:\Program Files\Xming. Чтобы установленный X-Server мог принимать удаленные запросы от кластера, необходимо в файл X0.hosts внести строчку mvs16.cc.dvo.ru. Запуск X-Server'а осуществляется через Пуск – Программы – Xming – Xming. Пользователям ОС Unix (Linux) рекомендуется проконсультироваться у специалистов или обратиться в службу поддержки центра коллективного пользования (настройка данной конфигурации может сильно зависеть от особенностей Вашей системы и выходит за рамки данных методических указаний).

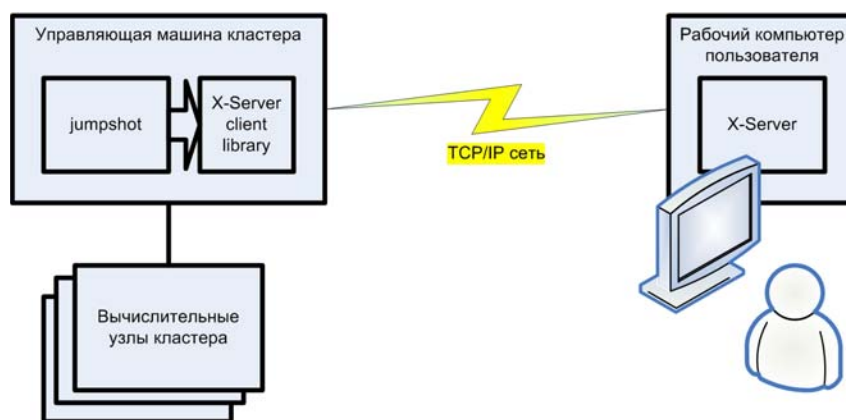


Рис. 1. Технология X-Window

Запуск jumpshot происходит с консоли (командный интерфейс). Перед запуском необходимо указать переменную среды DISPLAY, которая определяет дисплей, используемый X-Window для вывода изображения. DISPLAY имеет синтаксис:

```
DISPLAY=host_name:display_num
```

где `host_name` – имя или IP-адрес машины, на котором запущен X-Server (в Вашем случае это должен быть IP-адрес Вашей машины; проконсультируйтесь у администраторов), `display_num` – по умолчанию используйте значение 0.

Для присвоения значения переменной среды в Unix (bash оболочка) используется команда export. Например

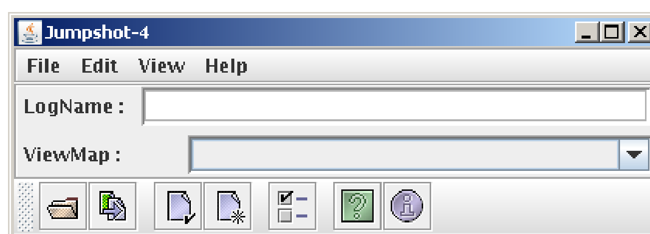
```
ivanov@1[~]$ export DISPLAY=192.168.2.1:0
```

После этого можно запустить XMPI. В консоли введите команду

```
ivanov@1[~]$ jumpshot &
```

Внимание! Для использования xmpi за пределами ИАПУ обратитесь к комментариям в Приложении 2.

В результате откроется основное окно программы, которое показано на рисунке ниже. Оно имеет три основные зоны. В верхней части окна расположена строка меню. В нижней части окна расположена инструментальная панель, в которой находятся кнопки для вызова дополнительных окон программы. В центральной части окна содержится панель, где необходимо указать log-файл с данными о трассировке.



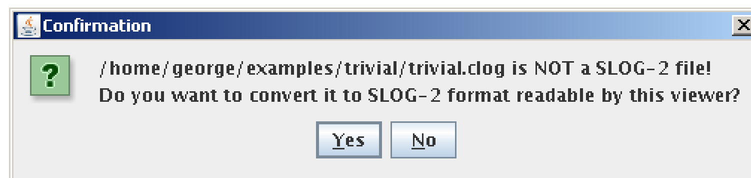
Прежде чем можно будет использовать jumpshot, параллельная программа должны быть специальным образом подготовлена. На стадии компиляции необходимо включить опцию «-mpilog», которая вносит дополнительный код в исполняемый файл, обеспечивающий сборку и хранение трассировочной информации.

Ниже приведен пример makefile'a с включенными параметром mpilog.

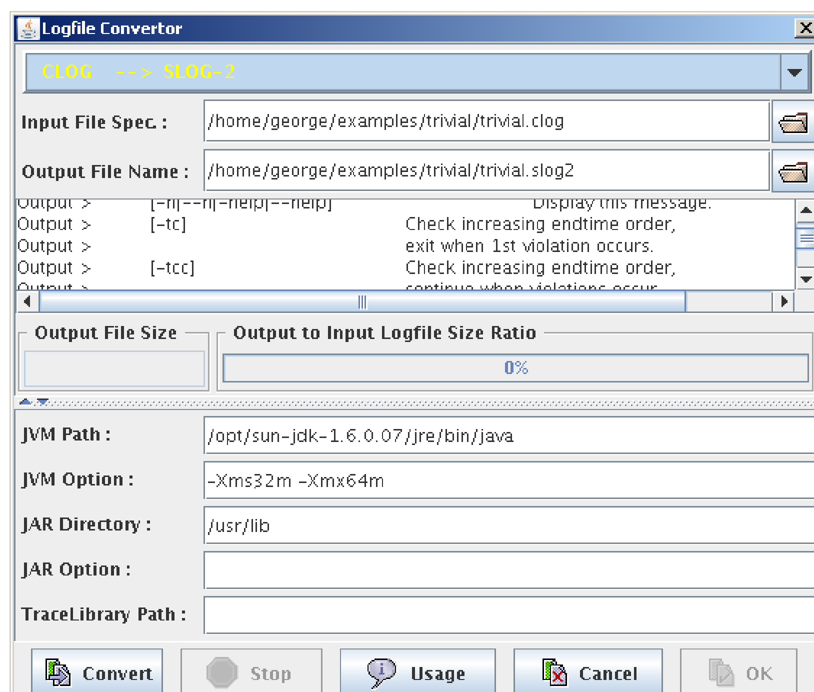
```
CXX=mpiCC
CPPFLAGS=-mpilog

all:      example

example:  main.o file1.o file2.o
          $(CXX) $(CPPFLAGS) -o example main.o file1.o file2.o

main.o:   main.cpp
          $(CXX) $(CPPFLAGS) -c main.cpp

file1.o:  file1.cpp file1.h
          $(CXX) $(CPPFLAGS) -c file1.cpp

file2.o:  file2.cpp file2.h
          $(CXX) $(CPPFLAGS)  -c file2.cpp
```

Запуск и выполнение параллельной программы происходит аналогично за исключением того, что по окончанию выполнения в директории запуска будет создан файл с расширением .clog. Имя файла будет совпадать с именем исполняемого файла вашей параллельной программы.

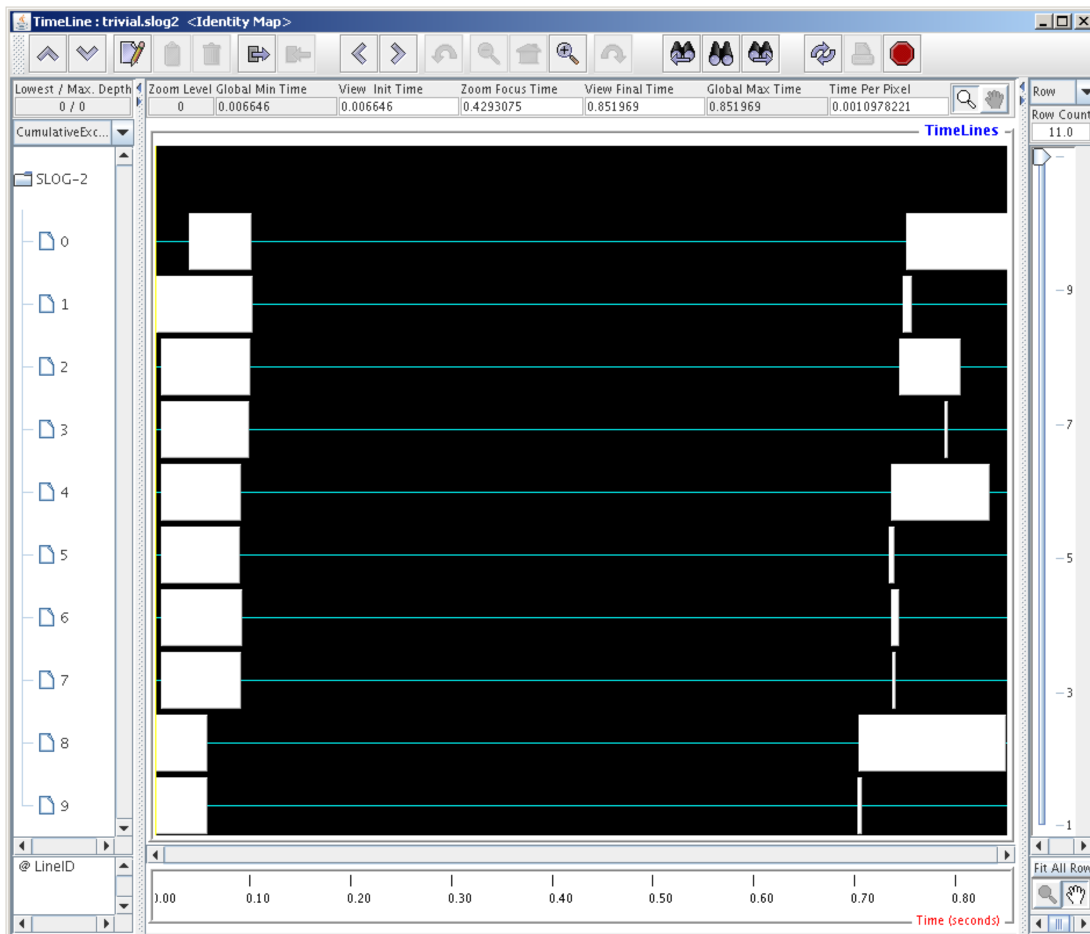
После того как clog-файл создан, и ваша параллельная программа закончила работу, его можно открыть в jumpshot. В начале будет предложено переконверитировать исходный файл в более новый формат.



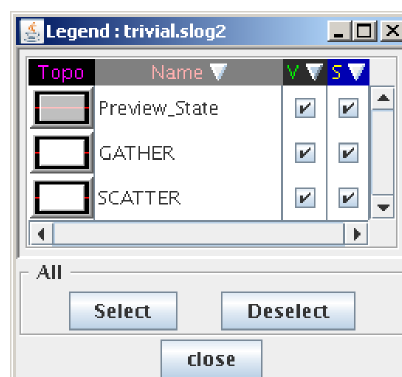
Необходимо нажать Yes и далее в следующем окне нажать Convert и Ok.



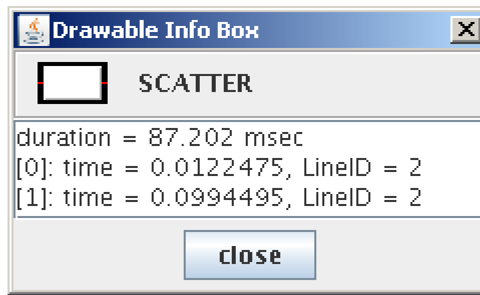
В результате будет создан slog2-файл, содержимое которого будет отражено в главном окне jumpshot.



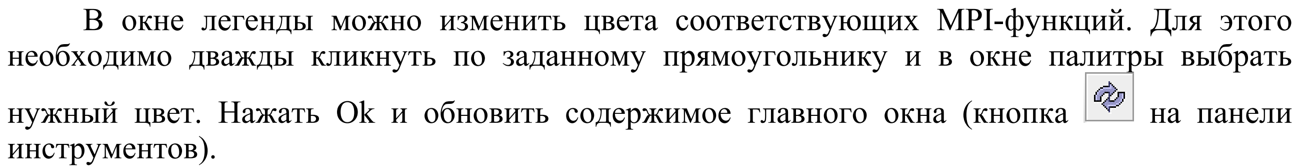
Главное окно содержит следующие зоны. Левая часть отображает ранги процессов параллельной программы. Нижняя часть окна отображает временную шкалу в секундах. Центральная часть окна отображает работу соответствующих процессов с привязкой к временной шкале. Горизонтальнае линии показывают привязку к процессам. Горизонтальные прямоугольники показывают выполнение соответствующих MPI-функций. Список выполненных в программе функций по типам отображается в окне легенды (Legend).

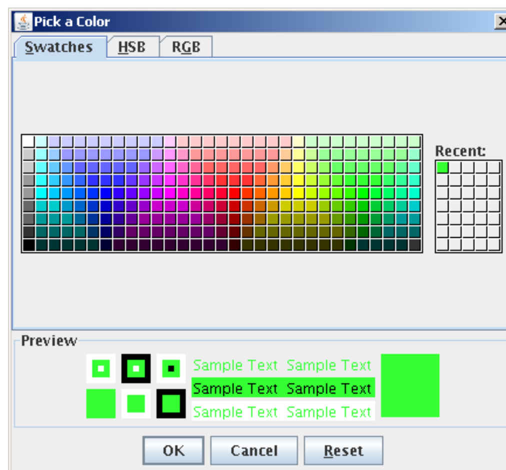


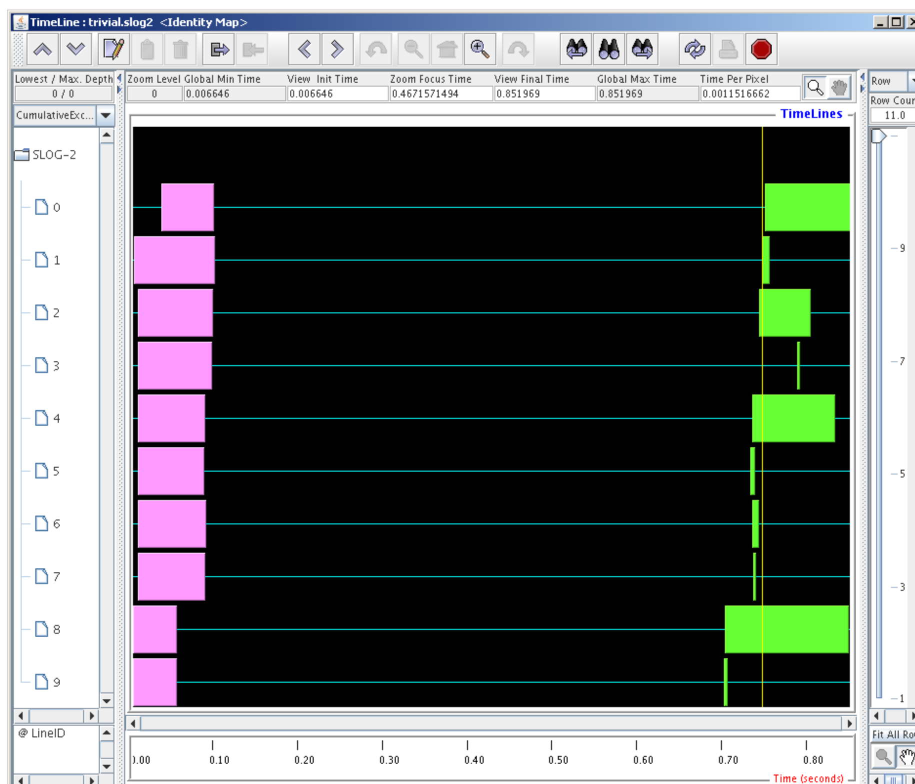
В данном примере параллельной программы используется только две MPI-функции: MPI_Scatter и MPI_Gather. Нажатием правой кнопки мыши на соответствующем прямоугольнике можно посмотреть детали выполнения соответствующей MPI-функции.



Используя кнопки панели управления (расположена в верхней части главного окна), можно регулировать масштаб временной шкалы, а также осуществлять поиск необходимых MPI-вызовов.

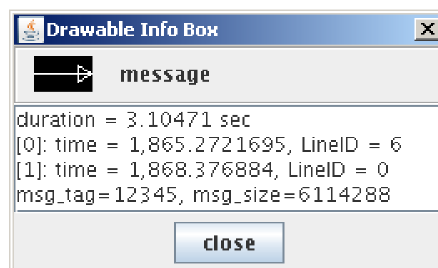
В окне легенды можно изменить цвета соответствующих MPI-функций. Для этого необходимо дважды кликнуть по заданному прямоугольнику и в окне палитры выбрать нужный цвет. Нажать Ok и обновить содержимое главного окна (кнопка  на панели инструментов).





Дополнительно на временной диаграмме процессов стрелками отображается передача парных сообщений между процессами. В данном примере синим цветом отображаются функции MPI_Send, а зеленым цветом – MPI_Recv.

Нажав на стрелочку правой кнопкой можно посмотреть основные параметры сообщения, которое было передано от одного процесса к другому. Например, следующий рисунок показывает, что сообщение передавалось от процесса с рангом 6 (LineId=6) к процессу с рангом 0 (LineId = 0). Сообщение передавалось со значением тега 12345, а длина сообщения равна 6114288 байт (~6Мб). На передачу сообщения потребовалось 3 секунды общего времени работы программы.



Чтобы зваершить работу с программой jumpshot в меню основного окна выберите File-Exit.

Задания

1+. Написать программу, вычисляющую сумму элементов n последовательностей вещественных чисел, находящихся в n файлах, имена которых заданы массивом a. Ответ должен быть записан в файл res.txt. Программа запускает n процессов, вычисляющих ответ для каждого из файлов и прибавляющих его к результату, находящемуся в выходном файле.

2. Написать программу, получающую в качестве аргументов целое число n и массив длины n с именами файлов, содержащих единую последовательность вещественных чисел неизвестной длины, и возвращающую количество участков постоянства этой последовательности. Программа возвращает –1, -2, и т.д., если она не смогла открыть какой-либо файл, прочитать элемент и т.д. Программа должна запускать n процессов, обрабатывающих свой файл и передающих результаты в основной процесс через очередь сообщений для формирования ответа для последовательности в целом.

3. Написать программу, осуществляющую мониторинг и перезапуск в случае завершения работы заданного количества приложений. Приложения задаются массивом строк, являющихся их полным путевым именем, и не имеют аргументов.

4+. Написать реализацию стека строк в разделяемой памяти. При запуске программа создает блок разделяемой памяти (если его еще нет) или присоединяется к существующему блоку. Программа должна обеспечивать основные функции работы со стеком (добавить и удалить элемент) и возможность запуска себя во многих экземплярах.

5+. Написать реализацию набора конфигурационных параметров для многих одновременно работающих экземпляров программы. Набор параметров задается некоторой структурой данных и хранится в разделяемой памяти. Блок разделяемой памяти создается при запуске первого экземпляра программы и заполняется из файла. Перед окончанием работы последнего экземпляра набор параметров сохраняется в файл, а блок разделяемой памяти удаляется. Программа должна обеспечивать основные функции работы с набором параметров (прочитать и изменить элемент), причем в случае изменения данных одним из экземпляров, он оповещает все остальные экземпляры с помощью сигнала.

6. Написать multithread-функцию, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и возвращающую ненулевое значение, если массив a симметричен (т.е. $a_{ij} = a_{ji}$), 0 в противном случае. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту функцию и выводить на экран результат ее работы.

7. Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на ее транспонированную. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

8. Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на матрицу (a+at)/2 (1), где at – транспонированная матрица a. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

9. Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, являющееся длиной этого массива, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент массива (для которого это возможно) на среднее арифметическое соседних элементов. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна

вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

10. Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, вспомогательный массив в вещественных чисел длины n (в каждом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{ij}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

11. Написать multithread-подпрограмму, получающую в качестве аргументов n x n массив a вещественных чисел, вспомогательный массив в вещественных чисел длины 2n (в каждом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+2,j} + a_{i-2,j} + a_{i,j+2} + a_{i,j-2} - 4a_{ij}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

12. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и возвращающую ненулевое значение, если массив a симметричен (т.е. $a_{ij} = a_{ji}$), 0 в противном случае. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту функцию и выводить на экран результат ее работы.

13. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на ее транспонированную. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

14. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую матрицу a на матрицу (a+at)/2 (1), где at – транспонированная матрица a. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

15. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, целое число n, являющееся длиной этого массива, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент массива (для которого это возможно) на среднее арифметическое соседних элементов. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

16. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, вспомогательный массив в вещественных чисел длины n (в каждом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+1,j} + a_{i-1,j} + a_{i,j+1} + a_{i,j-1} - 4a_{ij}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной

формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

17. Написать MPI-программу, получающую в качестве аргументов n x n массив a вещественных чисел, вспомогательный массив в вещественных чисел длины 2n (в каждом thread свой), целое число n, номер задачи (thread) k, общее количество задач (threads) p, и заменяющую каждый элемент a_{ij} матрицы a (для которого это возможно) на $a_{i+2,j} + a_{i-2,j} + a_{i,j+2} + a_{i,j-2} - 4a_{ij}$. При этом должна быть обеспечена равномерная загрузка всех задач. Основная программа должна вводить числа p, n и массив a (из файла или по заданной формуле), запускать задачи, вызывать эту подпрограмму и выводить на экран результат ее работы.

18. Написать MPI программу, вычисляющую минимальный элемент **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в выходной файл. Программа запускает **n** процессов, вычисляющих ответ для каждого из файлов и сравнивающих его с результатом, находящимся в выходном файле.

19. Написать MPI программу, вычисляющую максимальные элементы каждой из **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в виде последовательности чисел в выходной файл. Программа запускает **n** процессов, вычисляющих ответ для каждого из файлов и заносящих его в выходной файл.

20. Написать MPI программу, вычисляющую максимум из сумм элементов каждой из **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в выходной файл. Программа запускает **n** процессов, вычисляющих ответ для каждого из файлов и сравнивающих его с результатом, находящимся в выходном файле.

21. Написать MPI программу, сортирующую по возрастанию элементы **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в **n** новых файлов с именами, полученными из имён исходных файлов добавлением окончания “_res”. Программа запускает **n** процессов, сортирующих элементы соответствующих файлов.

22. Написать MPI программу, вычисляющую квадраты элементов **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в **n** новых файлов с именами, полученными из имён исходных файлов добавлением окончания “_res”. Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

23. Написать MPI программу, подсчитывающую число положительных элементов **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в выходной файл. Программа запускает **n** процессов, вычисляющих ответ для каждого из файлов и добавляющих его к результату, находящемуся в выходном файле.

24. Написать MPI программу, подсчитывающую разницу между суммой положительных элементов и модулем суммы отрицательных элементов **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в выходной файл. Программа запускает **n** процессов, вычисляющих ответ для каждого из файлов и добавляющих его к результату, находящемуся в выходном файле.

25. Написать MPI программу, удаляющую нулевые элементы из **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в **n** новых файлов с именами, полученными из имён исходных файлов добавлением окончания “_res”. Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

26. Написать MPI программу, заменяющую отрицательные элементы их модулями в **n** последовательностях вещественных чисел, находящихся в **n** файлах, имена которых заданы

массивом. Ответ должен быть записан в **n** новых файлов с именами, полученными из имён исходных файлов добавлением окончания "_res". Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

27. Написать MPI программу, заменяющую элементы их целыми частями в **n** последовательностях вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом. Ответ должен быть записан в **n** новых файлов с именами, полученными из имён исходных файлов добавлением окончания "_res". Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

28. Написать MPI программу, вычисляющую произведение положительных элементов **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом.Ответ должен быть записан в **n** новых файлов с именами , полученными из имен исходных файлов добавлением окончания "_res". Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

29. Написать MPI программу, вычисляющую произведение отрицательных элементов **n** последовательностей вещественных чисел, находящихся в **n** файлах, имена которых заданы массивом.Ответ должен быть записан в **n** новых файлов с именами , полученными из имен исходных файлов добавлением окончания "_res". Программа запускает **n** процессов, обрабатывающих элементы соответствующих файлов.

Литература

1. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ, Лаборатория знаний, 2003.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. - СПб.: БХВ-Петербург, 2002.
3. http://www-unix.mcs.anl.gov/mpi/,
4. http://parallel.ru

Учебное издание

Маргарита Александровна Князева
Лилия Александровна Молчанова
Георгий Витальевич Тарасов

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методические указания и задания для студентов
специальности 351500
Математическое обеспечение и администрирование информационных систем

В авторской редакции
Технический редактор Л.М. Гурова
Компьютерный набор и верстка автора

Подписано в печать 31.01.07
Формат 60\84 1/16. Усл. печ. л. 1,1. Уч.-изд. л. 0,9.
Тираж 25 экз.

Издательство Дальневосточного университета
690950, Владивосток, ул. Октябрьская, 27.
Отпечатано в лаборатории
кафедры компьютерных наук ИМКН ДВГУ
690950, Владивосток, ул. Октябрьская, 27, к. 132.

Приложение 1

Данные для авторизации пользователей на 2008/2009 учебный год группы 243

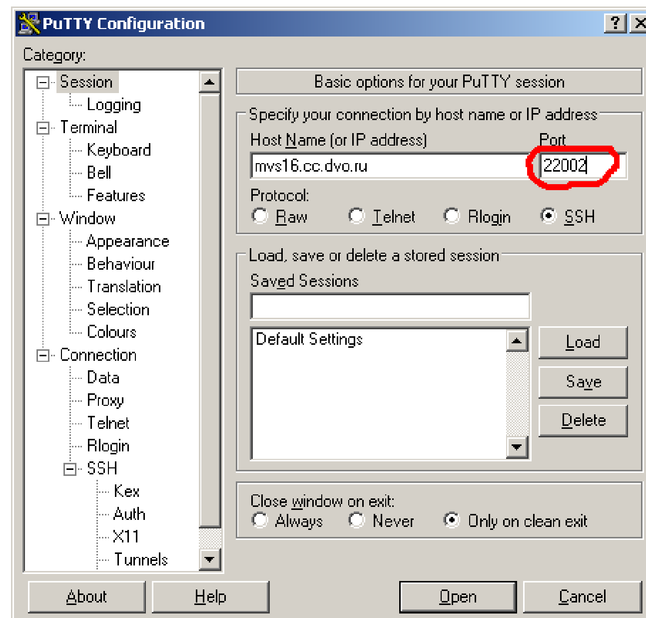
№	ФИО	Имя	Пароль
1	Адушев Александр		
2	Бабенко Евгения		
3	Беломестнов Егор		
4	Болгов Михаил		
5	Воронцов Сергей		
6	Гончарук Андрей		
7	Кушнаренко Анна		
8	Ларин Максим		
9	Лукина Елена		
10	Лялякин Никита		
11	Немцев Сергей		
12	Потапенко Богдан		
13	Пушкин Алексей		
14	Сухомлинов Александр		
15	Ткалин Владимир		
16	Хомченко Евгений		
17	Вельмисева Вероника		
18	Соколов Сергей		
19			
20			
21			

Доступ к кластру из внешних сетей

1. PuTTY

Hostname: **mvs16.cc.dvo.ru**
Port: **22002** (!!! Стандартный порт 22 не работает !!!)

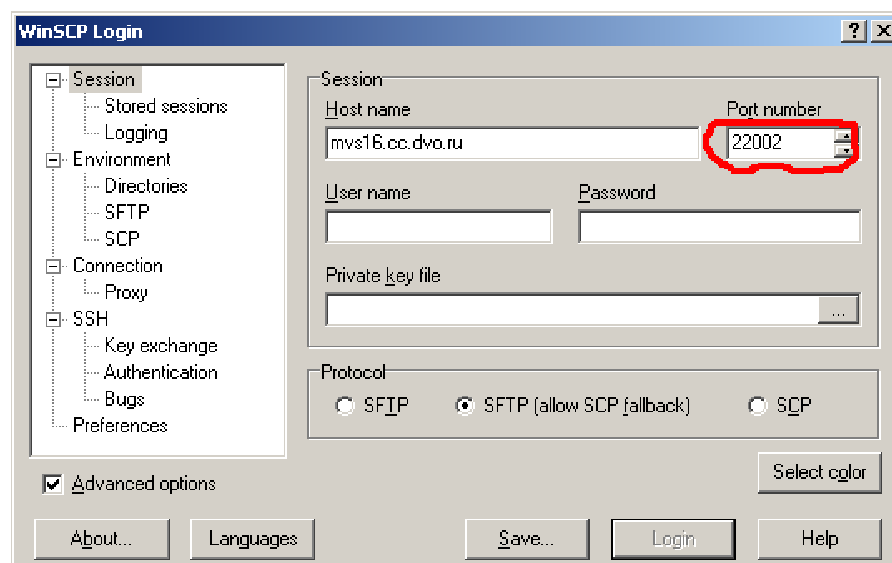
Пример



2. WinSCP

Hostname: **mvs16.cc.dvo.ru**
Port: **22002** (!!! Стандартный порт 22 не работает !!!)

Пример



3. Jumpshot

Для проброса X-сессии через SSH необходимо в PuTTY при установлении соединения указать дополнительный параметр слева в дереве Connection > SSH > X11 > Enable X11 forwarding. При использовании этой опции нет необходимости делать `export DISPLAY` – SSH сделает это самостоятельно. После авторизации и установления соединения запустите jumpshot так, как описано на стр. 48.

Пример