Министерство образования и науки Российской Федерации

ГОУВПО «Удмуртский государственный университет»

Организация параллельных вычислений для решения дифференциальных уравнений на blade-сервере

Учебно-методическое пособие

Ижевск 2011

УДК 004.382.2-027.1(07)+519.6(07)
ББК 32.973.202я7+22.193я7
О 641

Рекомендовано к изданию
учебно-методическим советом УдГУ

Составители: М.А. Клочков, К.Ю. Марков,
Ю.С. Митрохин, Л.С. Чиркова
Рецензент: д.ф.-м.н., профессор А.П. Бельтюков

О 641 **Организация параллельных вычислений для решения дифференциальных уравнений на blade-сервере:** учеб.-метод. пособие / сост. М.А. Клочков, К.Ю. Марков, Ю.С. Митрохин, Л.С. Чиркова. Ижевск: Изд-во «Удмуртский университет», 2011. 79 с.

Учебно-методическое пособие состоит из инструкций и рекомендаций по использованию кластера на базе HP ProLiant 260c для параллельных вычислений. Кластерные системы можно использовать для решения широкого спектра задач: решение систем дифференциальных уравнений, вычисления сверток, сумм, функций от каждого элемента массива и т.д.

Пособие может быть рекомендовано студентам, а также преподавателям для проведения практических занятий и организации самостоятельной работы студентов по учебным курсам: «Системное администрирование», «Параллельные вычисления», «Численные методы».

УДК 004.382.2-027.1(07)+519.6(07)
ББК 32.973.202я7+22.193я7

© сост. М.А. Клочков, К.Ю. Марков, Ю.С. Митрохин, Л.С. Чиркова, 2011

© Изд-во «Удмуртский университет», 2011

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **5**

НАСТРОЙКА СОЕДИНЕНИЯ И АВТОРИЗАЦИЯ **9**

Авторизация из ОС семейства Unix 9
Авторизация из ОС семейства Windows................................... 12

РАБОТА С ФАЙЛАМИ.. **15**

РАСПРЕДЕЛЕНИЕ НАГРУЗКИ **19**

ВЫХОД... **21**

ДЕКОМПОЗИЦИОННАЯ ФУНКЦИОНАЛЬНАЯ СТРУКТУРА БЛЕЙД-СЕРВЕРА **21**

АЛГОРИТМ МАТРИЧНОЙ ПРОГОНКИ **23**

Последовательная матричная прогонка (ПМП) 23
Распараллеливание матричной прогонки (РМП) 24
Программная реализация алгоритма РМП 25
Инструкция по компилированию и запуску программы 30

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ ВЫЧИСЛЕНИЯ ФУНКЦИИ ЦЕНЫ ДИФФЕРЕНЦИАЛЬНОЙ ИГРЫ **32**

Постановка задачи 32
Алгоритм вычисления функции цены 33
Задание 38

ПРИЛОЖЕНИЕ 1. КОМАНДЫ UNIX.. **41**

Работа с сетью 42
Права доступа к файлам (каталогам).. 42
Контроль процессов.. 43
Встроенные в Linux программные утилиты и языки 43

ПРИЛОЖЕНИЕ 2. MC **44**

ПРИЛОЖЕНИЕ 3. FTP.. **47**

ПРИЛОЖЕНИЕ 4. РЕДАКТОР VI **48**

История создания редактора VI 48
Режимы работы редактора VI 48
Командный режим *48*
Режим ввода командной строки *50*
Режим ввода текста *50*
Визуальный режим *50*
Вызов редактора 52
Выход из редактора 53
Поиск и замена текста 53
Использование регулярных выражений 54
Основные команды VI 55

ПРИЛОЖЕНИЕ 5. СТРУКТУРА ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ 58

ПРИЛОЖЕНИЕ 6. MPI 60

Основные понятия MPI. Парадигма SPMD 61
Общая организация MPI 62
Базовые функции MPI 63

ПРИЛОЖЕНИЕ 7. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ РЕШЕНИЯ УРАВНЕНИЯ ЛАПЛАСА 66

ВВЕДЕНИЕ

В данной работе изложена информация по использованию кластера на основе HP ProLiant BL260c для параллельных вычислений. Пособие предназначено для подготовки квалифицированных пользователей систем кластерного типа под управлением ОС Linux.

Кластер — это разновидность параллельной или распределённой системы, которая состоит из нескольких связанных между собой компьютеров и используется как единый, унифицированный компьютерный ресурс.

Структура данного пособия построена таким образом: сначала изложена подробная информация о том, как использовать Кластер – авторизоваться, работать с файлами, запускать программы, а затем описаны некоторые приемы по построению параллельных программ, примеры использования данных приемов и параллельного программирования для решения дифференциальных уравнений, а также для игровых задач.

Для работы на Кластере существует набор стандартных операций:

1) настройка соединения и авторизация;
2) создание файла, копирование и удаление файла;
3) параллельное программирование;
4) компиляция;
5) запуск задачи;
6) постановка задания в очередь, просмотр очередей;
7) выход.

Если читатель знаком с работой в операционной системе Unix, то приложения 1-4, разделы, связанные с настройкой соединения, авторизацией, выходом из системы, работой с файлами и, возможно, отладкой и выполнением готовых программ можно пропустить или ознакомиться бегло.

Для читателя, не имеющего начальных знаний по работе в ОС Unix, данные разделы являются обязательными для прочтения, без освоения этой минимальной информации к работе с Кластером приступать нельзя.

Изучение разделов, посвященных структуре параллельной программы, методов параллельного программирования, примеров использования данных методов и т.д., требует знаний курсов «Программирование», «Операционные системы» и «Численные методы», которые входят в программу подготовки специалистов по прикладной математике и информатике.

Пособием можно пользоваться не только в рамках учебных курсов для проведения лекционных и практических занятий, но и для обучения работе с Кластером научных сотрудников, инженеров, программистов, администраторов и преподавателей для проведения научных исследований в различных отраслях науки.

Кластеры используются для решения следующих исследовательских задач:

- построение распараллеливающих процедур для задач управления при дефиците информации в соответствии с алгоритмом стохастического синтеза;
- разработка разностных сеточных методов, позволяющих вычислять функцию цены дифференциальной игры и решение уравнений Гамильтона-Якоби;
- решение игровых задач преследования с неполной информацией, используя информационные множества из пространства состояний;
- распараллеливание вычислений в задачах гарантированного оценивания, возникающих при описании динамики систем с неопределенными параметрами;
- исследование методов высокоточной навигации и наведения движущихся объектов с коррекцией параметров движения по наблюдениям внешних информационных по-

лей; исследование внутренних и внешних задач газовой динамики в областях сложной конфигурации;

- расчеты движения управляющих поршней в задаче о неограниченной кумуляции энергии при безударном сжатии газа;
- использование параллельных алгоритмов при решении задач упругости и пластичности для тел вращения;
- построение оптимальных криволинейных сеток в многосвязных областях сложной геометрической формы;
- разработка алгоритмов и программ параллельного решения больших систем алгебраических уравнений с матрицами специального вида;
- разработка параллельных итерационных алгоритмов решения задач линейного и выпуклого программирования на базе фейеровских процессов; программные реализации параллельного симплекс-метода;
- создание параллельных алгоритмов декомпозиции задач линейного программирования;
- разработка параллельных алгоритмов и программ для задач навигации и наведения с использованием методов анализа трехмерных сцен;
- решение задач автоматизированного поиска и распознавания объектов на изображениях;
- исследование параллельных методов сжатия и восстановления картографической информации, информации о физических полях, пространственных сценах, фотографических, радиолокационных, телевизионных и иных изображений;
- реализация сложных расчетов в физике твердого тела и квантовой химии.

Кластерные системы и работа с ними – это новая, динамично развивающаяся отрасль практического знания. Высокая компетенция научного персонала в области высоко-

производительных вычислений часто приводит к научным открытиям, увеличивает потенциал каждого отдельного исследования и научного знания в целом.

Создание сборника, содержащего как руководство по использованию Кластера, так и некоторые рекомендации по написанию и использованию параллельных программ, а также указания некоторых методов расчетов, актуальная на сегодняшний день задача. Литературы в этой новой отрасли знаний немного, часто она содержит недостаточно полную информацию.

Большинство параллельных программ на данный момент написано на языке программирования Fortran. В пособии освещаются технологии, которые применялись ранее, а также совершенно новые, тексты приведенных в примерах параллельных программ на языке высокого уровня – языке программирования Си.

Настройка соединения и авторизация

Кластер, работа с которым будет описана в данной инструкции, имеет семь узлов. На каждом узле имеется два четырехядерных процессора. Таким образом, на кластере имеется 56 ядер, что позволяет запустить 56 параллельных заданий или одну параллельную задачу на 56 ядрах.

Достаточно зайти на один из узлов и запустить на нем задачу, чтобы в вычислениях участвовали все остальные узлы.

Предполагается, что авторизация осуществляется с внутренних машин Удмуртского государственного университета.

Авторизация из ОС семейства Unix

Сначала рассмотрим случай, когда авторизация происходит из операционной системы семейства Unix. Для того, чтобы перейти на Кластер, нужно набрать в командной строке

```
ssh 192.168.42.236
```

Данная команда предназначена для установки **ssh**-соединения с узлом **n1**, который имеет IP-адрес 192.168.42.236.

SSH—сетевой протокол сеансового уровня, позволяющий производить удалённое управление операционной системой и туннелирование **TCP**-соединений (например, для передачи файлов), шифрует весь трафик, включая и передаваемые пароли. **SSH** допускает выбор различных алгоритмов шифрования.

Возможны такие варианты использования команды **ssh**:

`ssh user@host` – подключится к **host** под логином **user**

`ssh -p port user@host` – подключится к **host** на порт **port** под логином **user**

Если Ваш логин на Кластере отличается от логина на машине, с которой Вы хотите зайти на Кластер, то необходимо использовать команду:

```
ssh kllogin@192.168.42.236
```

где вместо **kllogin** нужно указать Ваш логин на Кластере.

```
94.25.116.117 - TuTTY
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$ ssh n1
```

Рис. 1. Команда **ssh** для перехода на узел Кластера

Нажмите клавишу **Enter**. Появится приглашение ввести пароль.

Логин и пароль у каждого пользователя свой. Чтобы иметь возможность работать на Кластере, необходимо связаться с администратором по адресу m4rk0v@gmail.com. *В письме предварительно указать Ваш логин и пароль.*

Введите пароль и нажмите клавишу **Enter**.

```
94.25.116.117 - TuTTY
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$ ssh n1
ls@n1's password:
Last login: Mon Jan  3 22:47:12 2011 from vkimdeb
ls@n1:~$ █
```

Рис. 2. Авторизация на узле Кластера

Теперь можно работать на узле Кластера как в традиционной Unix-среде. Например, чтобы поменять пароль, введите в командной строке

```
passwd
```

и нажмите клавишу **Enter**. В следующей строке будет запрошен Ваш текущий пароль

```
Enter login(LDAP) password:
```

Введите пароль и нажмите **Enter**. Затем появится запрос нового пароля:

```
New password:
```

Введите новый пароль и нажмите клавишу **Enter**. Повторите новый пароль:

```
Re-enter new password:
```

снова нажмите **Enter**. Если процедура смены пароля прошла успешно, то появится соответствующее сообщение.

```
94.25.116.117 - TuTTY
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$
ls@vkimdeb:~$ ssh n1
ls@n1's password:
Last login: Mon Jan  3 22:47:12 2011 from vkimdeb
ls@n1:~$ passwd
Enter login(LDAP) password:
New password:
Re-enter new password:
LDAP password information changed for ls
passwd: password updated successfully
ls@n1:~$ █
```

Рис. 3. Процедура смены пароля

Если авторизация происходит впервые, рекомендуется пройти процедуру смены пароля.

Отметим, что сейчас мы находимся на узле, который имеет имя **n1**, остальные узлы имеют имена **n2**, **n3**, **n4**, **n5**, **n6** и **n7** соответственно, но для того, чтобы работать с Кластером, достаточно зайти только на один из узлов.

Узлы Кластера–это объединенные по сети компьютеры, работающие как одно целое. Реализована авторизация по ключу, поэтому при переходе от узла к узлу при помощи команды **ssh** система требовать ввода пароля не будет.

Авторизация из ОС семейства Windows

Чтобы попасть на узел Кластера из ОС семейства **Windows**, необходимо воспользоваться любой программой, которая осуществляет удаленный доступ с использованием протокола **SSH**, например **PuTTY**.

PuTTY — свободно распространяемый клиент для различных протоколов удалённого доступа, включая SSH, позволяет подключиться и управлять удаленным узлом (например, сервером). В **PuTTY** реализована только клиентская сторона соединения — сторона отображения, в то время как сама работа выполняется на другой стороне.

Чтобы начать работу с приложением, наведите курсор мыши на ярлык на рабочем столе и дважды щелкните левой клавишей:

Рис. 4. Ярлык приложения **PuTYY**

Данное приложение также можно запустить при помощи меню **Пуск -> Программы** или найти файл **PuTTYPortable.exe** или **putty.exe** на диске и запустить.

После того, как приложение начнет работать, откроется окно настройки, в поле **Host Name (or IP address)** и в поле **Saved Session** введите внутренний адрес узла **n1**: **192.168.42.236**

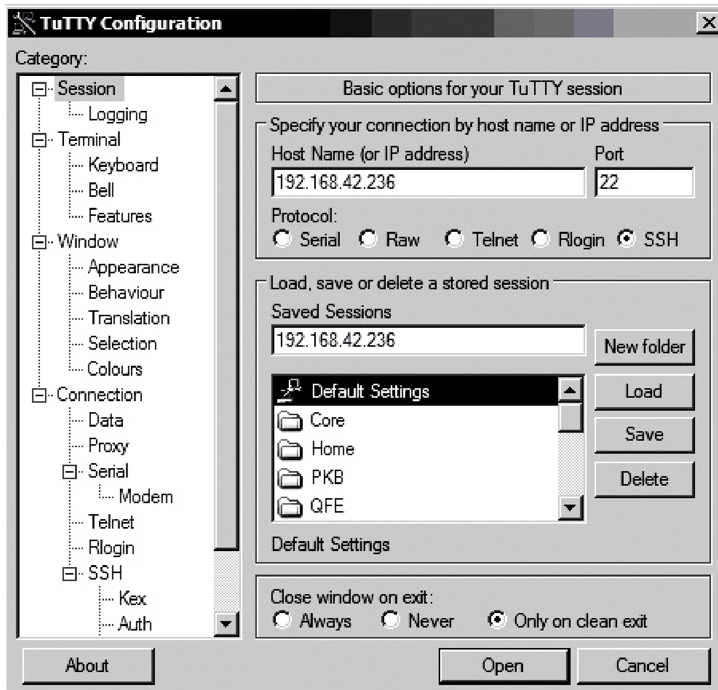

Рис. 5. Настройки соединения **Session**

В левой части данного окна организован иерархический список настроек. Для соединения с узлом Кластера используются настройки по умолчанию, поэтому имеет смысл указать только некоторые из них, наиболее часто изменяемые пользователями: **Keyboard**, **Translation**, **Data**.

Часто соединение не устанавливается правильно, если настройки соединения указаны неверно. Внимательно проверяйте адрес машины, поле ***Auto-login user name*** *и другие.*

Чтобы сохранить настройки соединения, нажмите кнопку **Save** в окне настроек **Session** (см. Рис. 5). В следующий раз, запуская приложение **PuTTY** для соединения с узлом Кластера, можно будет загрузить настройки соединения. Для этого выберите в списке сохраненных настроек (рас-

полагается ниже поля **Saved Sessions**) **192.168.42.236** и нажмите кнопку **Load**.

Нажмите кнопку **Open** в левой нижней части окна настроек соединения, и откроется окно для работы на узле Кластера как с обычной Unix-системой.

Работа с файлами

Небольшой справочник по командам Unix есть в **Приложении 1**, описание процесса работы в **Midnight Commander** (**mc**) – аналоге **Far** в ОС **Windows**, читайте в **Приложении 2**, о копировании файлов с локальной машины на узел Кластера посредством **ftp** читайте в **Приложении 3**.

Во всех Unix-системах имеется очень мощный и удобный тестовый редактор **vi** (**vim** — современная реализация редактора **vi**). Он был разработан вместе с самой системой Unix. В то время еще не было дисплеев, а были примитивные терминалы типа пишущих машинок или телетайпов.

Этот редактор развивался вместе с системой, при этом сохранялась преемственность предыдущих реализаций редактора **vi**. Такой подход позволил сохранить командную строку даже в современных экранных реализаций редактора **vi** (**gvim**).

Кстати сказать, многие операции в командной строке выполняются значительно быстрее, чем в экранном режиме, и многие опытные пользователи предпочитают работать в командной строке.

Другое важное достоинство редактора **vi** состоит в том, что он одинаково хорошо работает на всех типах терминалов.

В редакторе имеется более тысячи команд и различных скриптов. Это сначала отпугивает начинающих пользователей. Но для практической работы достаточно запомнить и довести до автоматизма 10-20 команд. Краткий список

наиболее часто используемых команд приведен в **Приложении 4**.

Наберите в командной строке

```
vim hello.c
```

для создания и редактирования файла.

Введите следующий текст:

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

void main( int argc, char *argv[] )
{
char      processor_name[MPI_MAX_PROCESSOR_NAME];
int       numprocs, namelen, rank ;

MPI_Init( &argc, &argv ) ;
MPI_Comm_size( MPI_COMM_WORLD, &numprocs ) ;
MPI_Comm_rank( MPI_COMM_WORLD, &rank) ;
MPI_Get_processor_name( processor_name,
&namelen) ;
printf("Process %d on %s out of %d\n", rank,
processor_name, numproc ) ;
MPI_Finalize();
}
```

сохраните файл – нажмите Esc, чтобы войти в режим команд, а затем (w — записать ,q — выйти):

```
:wq
```

Разберемся в тексте программы, чтобы понять, какой результат мы должны получить. Сначала создается при помощи функции **MPI_Init** группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором **MPI_COMM_WORLD**.

Затем в переменную **numprocs** при помощи функции **MPI_Comm_size** записывается количество процессов в области связи коммуникатора **MPI_COMM_WORLD**.

В переменную **rank** при помощи функции **MPI_Comm_rank** записывается номер процесса, вызвавшего эту функцию.

Процедура **MPI_Get_processor_name** возвращает имя процессора, на котором был выполнен вызов.

На экран выводится результат - для каждого процесса указывается, на каком узле Кластера он выполняется и сколько процессов сейчас запущено.

Примечание. О примерной структуре программы, реализующей параллельные вычисления, читайте в **Приложении 5**.

Для проверки программы на наличие ошибок, используйте транслятор языка Си:

```
gcc -c hello.c
```

При этом будет создан файл hello.o (объектный модуль), если в исходном модуле не было ошибок. При наличии ошибок на экране появится диагностика, а объектный модуль не будет создан.

Для создания работающего загрузочного модуля hello.x, необходимо использовать скрипт **mpicc**. Он содержит в себе транслятор с языка Си и загрузчик вместе со ссылками на все нужные библиотеки MPI.

Наберите в командной строке следующую строку, чтобы скомпилировать файл:

```
mpicc hello.c
```

В текущей директории появится файл **a.out**. По умолчанию, если явно не указано имя файла, результат компиляции помещается в файл **a.out**. Чтобы выполнить этот исполнимый файл, введите в командной строке

```
./a.out
```

Должен получится следующий результат:

Process 0 on n1 out of 1

Всего у нас запущен один процесс на узле **n1**. Теперь наберите команду

vim a

введите названия узлов Кластера: n1, n2, n3, n4, n5, n6, n7.

Сохраните файл и выполните команду, в которой задано, что имена узлов Кластера надо взять из файла с именем а и запустить 7 процессов:

mpirun –machinefile a –np 7 ./a.out

получим следующий результат:

```
ls@n1:~$ vi a
ls@n1:~$
ls@n1:~$
ls@n1:~$
ls@n1:~$ mpirun -machinefile a -np 7 ./a.out
Process 6 on n7 out of 7
Process 5 on n6 out of 7
Process 0 on n1 out of 7
Process 1 on n2 out of 7
Process 2 on n3 out of 7
Process 3 on n4 out of 7
Process 4 on n5 out of 7
ls@n1:~$ ls
a  aaa  a.out  hello.c  pr.txt
ls@n1:~$ █
```

Рис. 6. Результат программы

Примечание. О стандарте на программный инструментарий для обеспечения связи между отдельными процессами параллельной задачи **MPI** читайте в **Приложении 6**.

Распределение нагрузки

Средства **MPI** сами по себе позволяют осуществлять запуск параллельных задач, но правильнее использовать менеджеры ресурсов.

Система управления заданиями **Torque** предназначена для управления запуском задач на многопроцессорных вычислительных установках (в том числе кластерных). Она позволяет автоматически распределять вычислительные ресурсы между задачами, управлять порядком их запуска, временем работы, получать информацию о состоянии очередей. При невозможности запуска задач немедленно, они ставятся в очередь и ожидают, пока не освободятся нужные ресурсы.

Каждая параллельная задача, которую мы собираемся ставить в очередь на исполнение должна быть оформлена в виде пакета. Пакет представляет собой набор параметров запуска и инструкции, что конкретно мы хотим запустить. Здесь мы рассмотрим самые необходимые параметры и команды, которые потребуются в большинстве случаев [10].

Предположим, что мы собираемся запускать тестовую программу **./a.out**. Требуется, чтобы выполнялись следующие условия:

1. использовать три вычислительных узла;
2. на каждом из них мы займем по одному доступному процессору;
3. на вычислительных узлах должно быть доступно не менее 100Мб оперативной памяти;
4. о событиях запуска задачи и ее завершения (с ошибкой или без) мы должны быть уведомлены сообщением, направленным на электронную почту по адресу: kluser@udsu.ru .

Для описания такой задачи создадим скрипт (назовем его **a.pbs**):

```
#!/bin/sh
#
#PBS -l nodes=n1:ppn=1+2:ppn=1
#PBS -N ATest
#PBS -m abe
#PBS -M kluser@udsu.ru
#PBS -l pmem=100mb
cd /home/ls
mpirun ./a.out
```

Количество задействованных процессоров мы описываем параметром **nodes=n1:ppn=1+2:ppn=1**. Указываем, что первым узлом нашей параллельной задачи будет хост **n1**. Именно так мы назвали нашу консоль кластера. Кроме него задаче будет выделено еще два узла. Для каждого из этих трех узлов мы запрашиваем по одному процессору (**ppn=1**).

Параметр **-N ATest** - это название нашей задачи, так она будет отображена в очереди задач.

Параметры **-m abe** и **-M kluser@udsu.ru** указывают, какие должны быть уведомления и куда их посылать.

Последний параметр(**pmem=100mb)** - запрашиваемый размер оперативной памяти.

Остальные строчки - это собственно задача, которая должна быть исполнена на кластере: переход в каталог с программой и запуск ее на параллельное исполнение **OpenMPI**.

Размещение задания в очереди на исполнение, осуществляется командой **qsub**, единственным параметром которой будет имя скрипта:

```
qsub a.pbs
```

Посмотреть состояние очереди можно при помощи команды:

```
qstat
```

Выход

Для завершения работы с кластером нажмите сочетание клавиш **CTRL-D** или наберите в командной строке команду **exit**, чтобы покинуть окно удаленного соединения с узлом Кластера.

Декомпозиционная функциональная структура блейд-сервера

Рис. 7. Декомпозиционная функциональная структура блейд-сервера

Для того чтобы проектировать параллельные программы, необходимо знать физическое устройство Кластера.

Блейд-система [12] содержит следующие подсистемы (см. рис. 8):

1) Системная шина блейд-сервера, являющаяся средством коммуникаций компонентов сервера.

2) Центральный процессор или процессоры. Процессор является элементарным вычислительным компонентом сервера.

3) Оперативная память сервера, из которой берёт и в которую возвращает данные процессор.

4) Дисковый контроллер с физическими дисками, на которых постоянно хранится информация, с которой работает сервер.

5) Объединительная плата (т.н. backplane), к которой подключаются блейд-серверы.

6) Модуль питания блейд-системы. Передаёт электроэнергию от электросети к компонентам блейд-системы.

7) Модуль охлаждения блейд-системы. Отводит, выделяемое при работе серверов, тепло во внешнюю среду.

8) Коммутационные и управляющие модули. Производят обмен информацией между блейд-системой и окружением, в котором она располагается.

Обычно, кластеры данного типа помещены в стойку, обеспечивающую не только компактное расположение и защиту вычислительного оборудования, но и обмен между составными частями кластера.

В стойку монтируются вычислительные серверы, часто многопроцессорные, системы бесперебойного питания, сетевые карты для обычной и «быстрой» сети, устройства ввод-вывода (DVD-ROM), небольшие экраны для работы с консолью. Один из вычислительных узлов выделяют для управления, остальные – для расчетов.

Алгоритм матричной прогонки

При помощи алгоритма матричной прогонки программируют задачи линейной алгебры (системы линейных уравнений) и задачи по нахождению решений систем дифференциальных уравнений [5].

Последовательная матричная прогонка (ПМП)

Рассмотрим систему линейных алгебраических уравнений (СЛАУ) с матрицей блочно-трехдиагональной структуры ($A_1 = 0, C_M = 0$) :

$A_i y_{i-1} + B_i y_i + C_i y_{i+1} = f_i$ для $i = 1,.., M$, (1a)

где M - количество блоков на главной диагонали, A_i, B_i, C_i - квадратные матрицы (блоки) размерности N , y_i , f_i - векторы размерности N .

Согласно известному алгоритму метода ПМП [3] от системы (1а) переходим к системе (2а) (ход вперед):

$$\begin{cases} y_i + \widetilde{C}_i y_{i+1} = \widetilde{f}_i \\ y_M = \widetilde{f}_M \end{cases} \quad \text{для } i = 1,.., M-1, \qquad (2a)$$

По рекуррентным формулам:

$$\begin{cases} \widetilde{C}_1 = B_1^{-1} C_1, \\ \widetilde{C}_i = (B_i - A_i \widetilde{C}_{i-1})^{-1} (f_i - A_i \widetilde{f}_{i-1}), i = 2,.., M-1, \\ \widetilde{f}_1 = B_1^{-1} f_1 \\ \widetilde{f}_i = (B_i - A_i \widetilde{C}_{i-1})^{-1} (f_i - A_i \widetilde{f}_{i-1}), i = 2,.., M. \end{cases} \qquad (3a)$$

Затем окончательно рассчитываем решение СЛАУ (ход назад):

$$\begin{cases} y_M = \widetilde{f}_M , \\ y_i = \widetilde{f}_i - \widetilde{C}_i y_{i+1} \end{cases} \quad \text{для } i = 2,.., M \qquad (4a)$$

Распараллеливание матричной прогонки (РМП)

Алгоритм РПМ основан на известном алгоритме распараллеливания скалярной прогонки [4].

Пусть в вычислительной сети имеется $L+1$ процессор ($2L < M$), тогда назначим L векторов y_{i_k} ($1 < i_1 < i_2 < ... < i_L < M$, где $i_{k+1} - i_k > 1$) параметрическими неизвестными. Обозначим через S_k диапазон индексов от $i_{k-1}+1$ до $i_k - 1$ ($i_0 = 0, i_{L+1} = M$), а число диагональных блоков B_i, попавших в S_k, - через $m_k = i_k - i_{k-1} + 1$. На каждом процессоре одновременно с помощью алгоритма ПМП решаем систему СЛАУ с m_k неизвестными векторами y_i. Например, для k-го процессора

$$\begin{cases} B_j y_j + C_j y_{j+1} = f_j - A_j y_{i_{k-1}}, \\ A_{j+i} y_{j+i-1} + B_{j+i} y_{j+i} + C_{j+i} y_{j+i+1} = f_{j+i}, \\ A_{j+m_k} y_{j+m_k-1} + B_{j+m} y_{j+m_k} = f_{j+m_k} - C_{j+m_k} y_{i_k}, \end{cases} \quad (5a)$$

где $j = 1 + i_{k-1}$, а i берется из диапазона $[2, m_k - 1]$. Причем, если $m_k = 2$, то в системе (5а) второго уравнения нет, а при $m_k = 1$ система вырождается в одно уравнение: $B_j y_i = f_j - A_j y_{i_{k-1}} - C_j y_{i_k}$.

Из-за присутствия в правой части параметрических векторов $y_{i_{k-1}}$ и y_{i_k} решение системы (5а) получится в виде

$$y_i = \hat{f}_i - \hat{A}_i y_{i_{k-1}} - \hat{C}_i y_{i_k}, i = i_{k-1}+1,.., i_k - 1, \quad (6a)$$

где $\hat{f}_i, \hat{A}_i, \hat{C}_i$ находим по аналогичным рекуррентным формулам ПМП (1а-4а) для правой части системы, например, для A_i:

$$\begin{cases} D_1 = B_j^{-1} A_j, \\ D_i = (B_{j+i} - A_{j+i} C_{j+i-1})^{-1} A_{j+i} D_{i-1}, i = 2,.., m_k \end{cases} \text{(ход вперед)},$$

$$\begin{cases} \hat{A}_{i_k-1} = D_{m_k}, \\ \hat{A}_{j+i} = D_{j+i} - \hat{C}_{j+i} D_{j+i+1}, i = 1,.., m_k - 1, \end{cases} \text{(ход назад)}, \qquad (7а)$$

где D_i - рабочие матрицы k -й ветви параллельного алгоритма (k -го процессора).

Заметим, что по сравнению с алгоритмом ПМП в РМП из-за представления (6а) потребуется выделять память не только для хранения матриц ПМП - $\tilde{C}_i$, но и для $\hat{A}_i, \hat{C}_i$, что увеличивает запрашиваемую память почти в три раза.

После подстановки решений (6а) в оставшиеся L уравнений системы (1а), формируем редуцированную систему уравнений для определения параметрических неизвестных y_{i_k} ($A_{i_1} = 0, C_{i_L} = 0$) :

$$A_{i_k} y_{i_{k-1}} + B_{i_k} y_{i_k} + C_{i_{k+1}} = f_i, k = 1,.., L, \qquad (8а)$$

где

$$f_{i_k} = f_{i_k} - A_{i_k} \hat{f}_{i_k-1} - C_{i_k-1} \hat{f}_{i_k+1},$$

$$A_{i_k} = A_{i_k} \hat{A}_{i_k-1}, C_{i_k} = C_{i_k} \hat{C}_{i_k+1},$$

$$B_{i_k} = B_{i_k} + A_{i_k} \hat{C}_{i_k-1} + \hat{C}_{i_k} A_{i_k+1}.$$

Найдя методом ПМП решение СЛАУ (8а), по формулам (6а) получим оставшиеся неизвестные в каждом из процессоров.

Программная реализация алгоритма РМП

Программа строится исходя из трех посылок:

1. Для k -го процессора вводятся и вычисляются данные (A_i, B_i, C_i, f_i, y_i) только для своего диапазона индексов S_k , что уменьшает количество и объем обменов.

2. Предполагается, что поступление исходных матриц A_i, B_i, C_i на вход программы будет организовано так, чтобы позволить рекуррентно формировать систему (2а), не накапливая матрицы системы (1а) в памяти процессоров.

3. Редуцированная система (8а) вся решается на одном, управляющем всем счетом 0-процессоре (Host-процессоре). Кроме того, Host также решает одну из систем (5а) (т.е. имеет свою параллельную ветвь алгоритма), что увеличивает степень параллелизма и уменьшает общее время счета.

Эти посылки и сама логика алгоритма приводят во время счета к следующим организованным в программе, обменам между процессорами (табл. 1).

Таблица 1

Источник сообщения	Приемник сообщения	Данные обмена	Кол-во слов	Кол-во обменов
Host	Root	L, M, i_k, N, i_{k+1} и др.	7	L
Root	Host	$\hat{A}_{i_k-1}\hat{C}_{i_k-1}\hat{f}_{i_k-1}$ $\hat{A}_{i_k+1}\hat{C}_{i_k+1}\hat{f}_{i_k+1}$	$4N^2+2$	L
Host	Root	$y_{i_k} y_{i_{k+1}}$	$2N$	L

Программа состоит из двух модулей для Root- и Host-процессора.

Host-модуль в начале работы, исходя из принципа равнораспределения нагрузки, находит индексы параметрических неизвестных и рассылает их по процессорам вместе с другими начальными входными параметрами.

Затем находит решение редуцированной системы (8а), прерываясь, по мере необходимости, для приема необходимых для этого массивов (см. табл. 1) с рабочих процессоров. Потом рассылает найденные векторы параметрических неизвестных по процессорам и завершает работу.

Работа Root-модуля начинается с ожидания приема задания, в котором указаны параметры блоков и диапазон индексов S_k. Затем рассчитывается методом ПМП своя система уравнений (5а) и посылаются в Host-процессор необходимые для вычисления редуцированной системы массивы. Потом модуль переходит в ожидание приема векторов $y_{i_k}, y_{i_{k+1}}$ для окончательного расчета решения.

Каждый модуль использует свою подпрограмму обращения матриц (методом Гаусса), что сделано для уменьшения времени счета, так как в процессе обращения необходимо обрабатывать разный состав данных: вектор и матрицу – в редуцированной системе и вектор и три матрицы – в системе (5а).

Уравнение Лапласа

В пространстве R^2 рассмотрим уравнение Лапласа [11]:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0; x, y \in \Omega; \Omega = [a, b] \times [c, d];$$

$$u(x, c) = u(x, d) = 0; u(a, y) = f(y); u(b, y) = g(y);$$

с квадратной матрицей размерности (4m+2) x (4m+2).

В нашем случае Ω - квадрат. Пусть дана квадратная матрица

$$A = \begin{pmatrix} a_{00} & \cdots & a_{0n} \\ \cdots & \cdots & \cdots \\ a_{n0} & \cdots & a_{nn} \end{pmatrix}$$

Элементы a_{00}, a_{0n}, a_{n0}, a_{nn} матрицы A будем называть угловыми элементами.

Элементы a_{0j}, a_{nj}, a_{j0}, a_{jn}, где j = 1..n-1, матрицы A будем называть крайними элементами.

Элементы a_{ij}, где i = 1..n-1, j = 1..n-1, матрицы A назовем внутренними узлами матрицы.

Ниже приведен листинг программы для матрицы размерности 48x48, распределенной на 4 процессора. Каждый процессор считает одну из своих частей матрицы размерности 24x24.

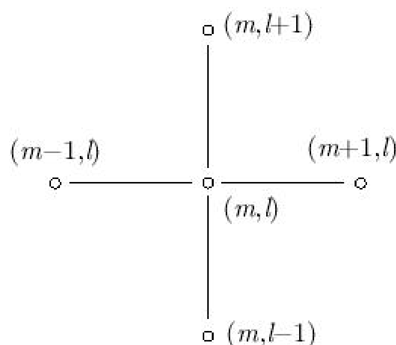

Рис. 8. Шаблон конечно-разностной схемы

Пусть заданы числа m, l = 0, 1, 2,…, 47.

Для построения конечно-разностной схемы будем использовать простой пятиточечный шаблон «крест». Запишем аппроксимирующее разностное уравнение, производные заменим вторыми разностями:

$$\frac{u_{m-1,l}-2u_{m,l}+u_{m+1,l}}{h^2}+\frac{u_{m,l-1}-2u_{m,l}+u_{m,l+1}}{h^2}=0$$

, где h – шаг по оси координат x,y. Эту же конечно-разностную схему можно записать в каноническом виде:

$$u_{m,l}=\frac{1}{4}(u_{m-1,l}+u_{m+1,l}+u_{m,l-1}+u_{m,l+1})$$

Каждый процессор определяет с чем он будет общаться на каждой итерации: с крайним узлом, угловым или внутренним.

Начальные значения в крайних узлах равны глобальным граничным условиям, аналогично получают свои локальные значения соседние для них внутренние узлы. Начальное значение всех остальных точек определяется средним значением из глобальных граничных условий.

Последовательность действий процессоров в каждой итерации:

1) Каждый процессор обменивается крайними значениями со своими 4-мя соседями.

2) Потом вычисляются новые значения для левого верхнего и нижнего правого углов («красные углы», если представлять матрицу как шахматную доску) для каждой части матрицы, которую обрабатывает процессор.

3) Процессоры обмениваются крайними значениями снова.

4) Затем вычисляются значения для правого верхнего и левого нижнего углов («черных углов»).

Каждые 20 итераций, процессоры вычисляют среднюю разность значений каждого узла разностной схемы сейчас и 20 итераций назад.

Задача с номером 0 получает данные расчетов, рассчитывается глобальная средняя разность (для всей матрицы). Если необходимая точность вычислений достигнута, задача с номером 0 собирает значения со всех частей матрицы. В противном случае, выполняется еще 20 итераций.

Инструкция по компилированию и запуску программы

Приведенная ниже программа – параллельная реализация, использующая конечно-разностную схему для решения уравнения Лапласа в пространстве R^2, работающая на четырех процессорах в режиме SPMD. В нее включены следующие функции: обмен процессорами значений в узлах разностной схемы, проверка сходимости, и использование некоторых межпроцессорных коммуникационных MPI-функций.

Скопируйте текст из **Приложения 7** в файл в вашей домашней директории на узле Кластера, и сохраните его под именем **laplace.c**. В командной строке выполните команду:

```
mpicc laplace.c
```

В каталоге, где лежит файл **laplace.c**, появится результат компиляции – файл **a.out**. Введите команду

```
mpirun -np 4 ./a.out
```

чтобы запустить данный исполнимый файл на четырех процессорах.

Результат выполнения программы появится в файле **parallel.laplace.out** в том же самом каталоге.

Параллельный алгоритм вычисления функции цены дифференциальной игры

Рассматривается одна дифференциальная игра, а также реализация вычисления цены этой игры [1].

Постановка задачи

Дана дифференциальная игра

$$\dot{x} = g(t,x) + B(t,x)u + C(t,x)v,$$
$$u \in P \subset R^p, v \in Q \subset R^q \tag{1}$$

на фиксированном интервале времени $[t_0, \vartheta]$ с терминальным функционалом

$$\gamma(x(\cdot)) = \sigma(x(\vartheta)). \tag{2}$$

Здесь x - n - мерный фазовый вектор системы; u и v - векторы управляющих воздействий первого и второго игроков соответственно, P и Q - выпуклые многогранники.

Вектор-функция $g(t,x)$ и матрицы-функции $B(t,x)$ и $C(t,x)$ предполагаются непрерывными по (t,x); функция $\sigma(\cdot): R^n \to R$ удовлетворяет локальному условию Липшица.

При некоторых ограничениях на правую часть системы (1) существует функция цены $w(t,x): R^n \to R$ дифференциальной игры (1), совпадающая с обобщенным решением задачи Коши для уравнения в частных производных первого порядка

$$\frac{dw}{dt}(t,x) + H(t,x,\nabla w(t,x)) = 0 \tag{3}$$

с граничными условиями, заданными на правом конце интервала времени $[t_0, \vartheta]$:

$$w(\vartheta, x) = \sigma(x). \tag{4}$$

Здесь
$\nabla w(t,x) = (\partial w(t,x)/\partial x_1, ..., \partial w(t,x)/\partial x_n)$; $H(t,x,s)$ - гамильтониан системы.

Алгоритм вычисления функции цены

Пусть $\Gamma = \{t_0, t_1, ..., t_N = \vartheta\}$ - разбиение отрезка $[t_0, \vartheta]$. По строение аппроксимации функции осуществляется в узлах сеточной области. Используется аппроксимационный разностный оператор

$$V(t_i, x) = \max_{y_j \in O_{K(t_i,x)\Delta_i}(x)} \max_{(p,q)} \max_{s \in \partial COV(t_{i+1}, x, y_j, p, q)} \{\Delta_i[< s, g(t_i, x) > +$$

$$+ < s, B(t_i, x)u^p > + < s, C(t_i, x)v^q >] + < s, x - y_j > + coV(t_{i+1}, x; y_j)\}.$$

здесь $V(\cdot)$ - аппроксимирующая функция, конструируемая на сеточной области, покрывающей множество $D; t_i, t_{i+1} \in \Gamma; \Delta_i = t_{i+1} - t_i; x$ - внутренняя точка, находящаяся на расстоянии от границы сеточной области, не меньшем чем

$K(t_i, x)\Delta_i;\ K(t_i, x) = \max_{u \in P, v \in Q} \| g(t_i, x) + B(t_i, x)u + C(t_i, x)v \|$; через

y_j обозначены точки сетки, принадлежащие замыканию окрестности $O_{K(t_i,x)\Delta_i}$ узла x; p и q - натуральные числа, номера вершин многогранников P и Q; $\partial coV(t_{i+1}, x, y_j; p, q)$ - пересечение субдифференциала локально выпуклой оболочки $coV(t_{i+1}, x, \cdot)$ сеточной функции $V(t_{i+1}, \cdot)$, посчитанного в узле y_j, с конусом линейности гамильтониана.

При выполнении некоторых условий (см. [6-9]) последовательное, отвечающее разбиению Γ, применение оператора шага обеспечивает построение аппроксимации решения задачи (3). При этом скорость сходимости имеет порядок $1/2$ по отношению к $diam\ \Gamma$.

Традиционный подход к построению на сетке решения краевой задачи здесь невозможен, поскольку обобщенное решение, вообще говоря, является негладким.

Особенностью данного алгоритма является независимость от сетки по пространственной переменной и размерности задачи, но при этом используются процедуры, требующие достаточно много машинного времени:

- поиск узлов сетки, принадлежащих окрестности текущего узла;
- построение выпуклой оболочки функции (в пространстве размерности больше трех);
- пересечение субдифференциала выпуклой оболочки с конусами линейности функции Гамильтона, реализованное как решение системы линейных неравенств.

Таким образом, возникает необходимость распараллеливания этого алгоритма.

Возможность распараллеливания алгоритма основана на том, что на каждом шаге по времени все узлы сеточной области можно обрабатывать одновременно. Алгоритм хорошо укладывается в известную схему «процессорной фермы» - мастер, рассылающий задания, и рабочие, выполняющие их. Шаги по времени обрабатываются последовательно.

На каждом шаге t_i на мастере проводится цикл по внутренним узлам сеточной области:

- если не все задания переданы – раздача заданий незанятым рабочим процессорам;
- опрос рабочих процессоров, ожидание приема и прием результатов;
- если все переданные задания обработаны – запись в файл приближенных значений функций в момент времени t_i и переход к моменту t_{i-1} .

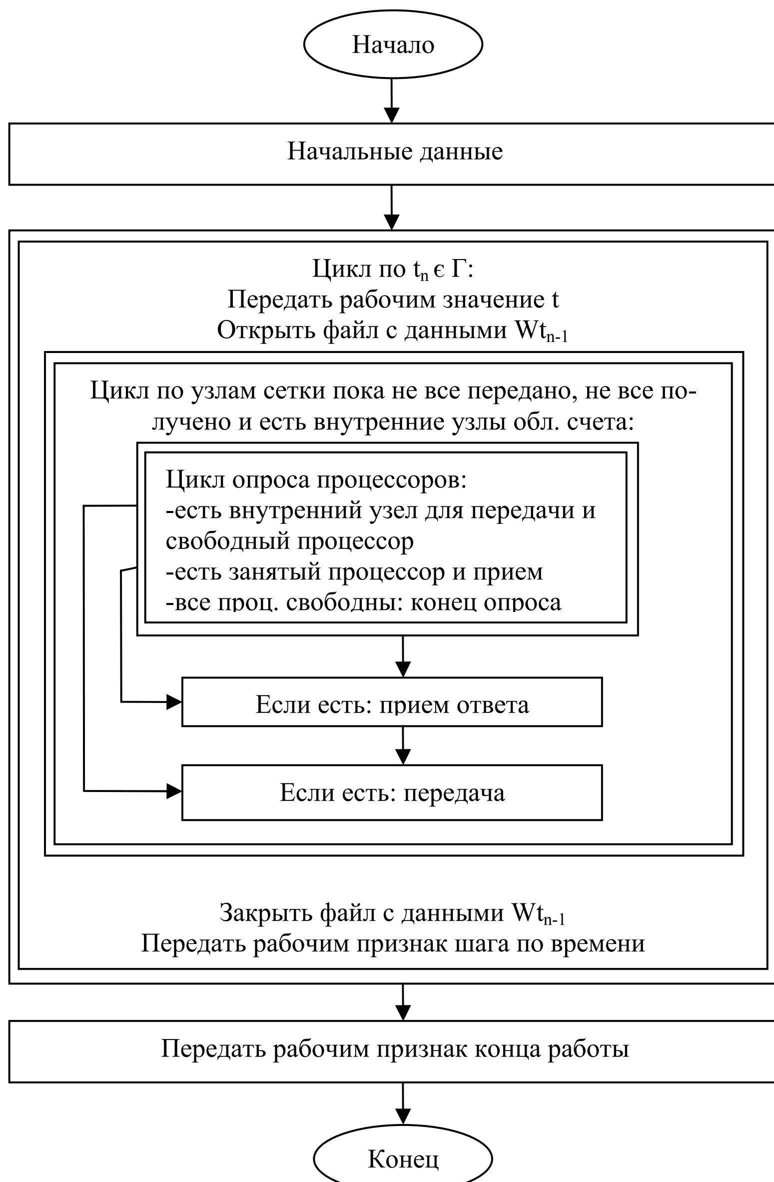

Рис. 9. Блок-схема задачи-мастера

Рабочий принимает от мастера информацию о текущем моменте времени, считывает из соответствующего файла данные о приближенных значениях функции в предыдущий момент времени, выполняет цикл:

- прием сообщения от мастера с номером узла x сетки, вычисление для него радиуса окрестности, поиск узлов сетки, принадлежащих этой окрестности;
- для внутренних узлов – вычисление приближенного значения функции цены;
- пересылка значения мастеру до тех пор, пока не будет получен признак перехода к следующему моменту времени, заканчивает работу, если получен признак окончания работы.

Начальные данные: необходимо задать динамику системы, размерности фазового пространства и управлений игроков, размеры некоторых массивов (количество узлов сетки, а также вершин многогранников P и Q и пр.) и создать файлы с информацией о сетке, начальных данных (4), ограничениях на управления (вершины многогранниках), некоторой дополнительной информацией (количество процессоров, интервал времени, шаг разбиения).

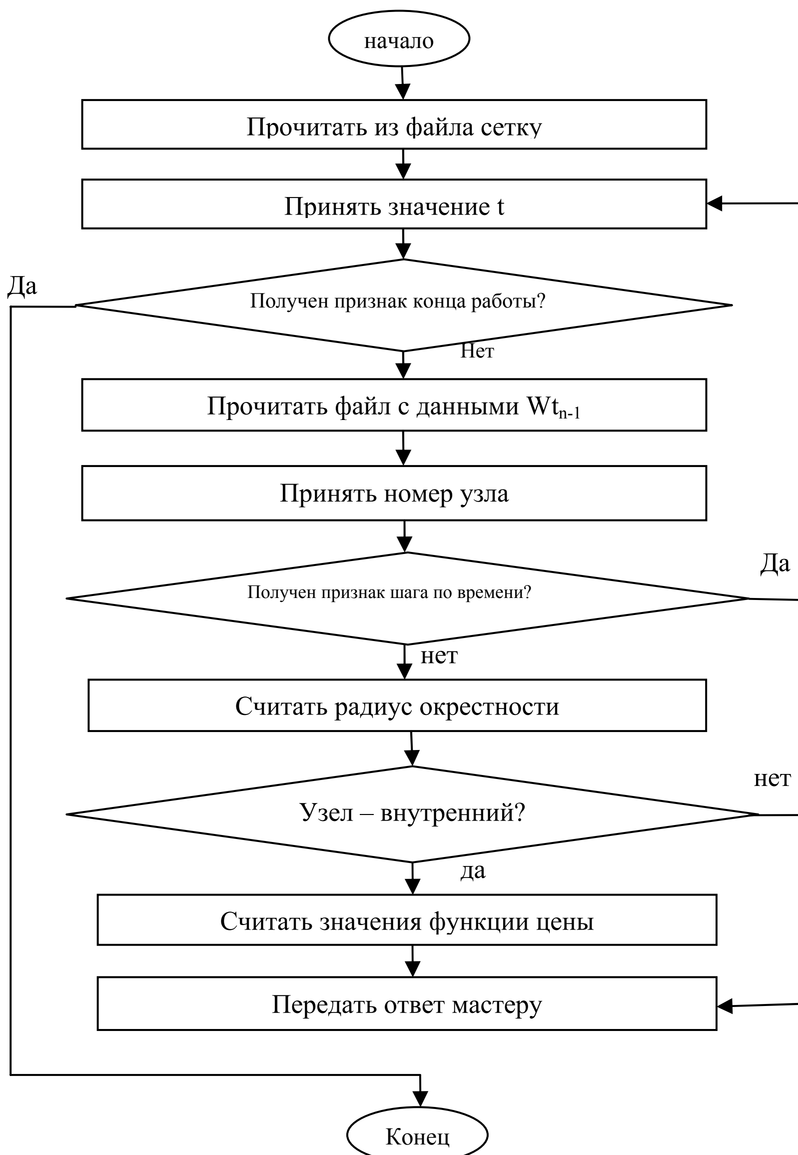

Рис.10. Блок-схема задачи-рабочего

Задание

Вычислить функцию цены дифференциальной игры:

$$\begin{cases} \dot{x}_1 = x_2 + v, \\ \dot{x}_2 = u, \end{cases}$$

$u \in [-1,1]$, $v \in [-1,1]$; $\sigma(x) = \max\{|x_1|, |x_2|\}$, $t \in [1,0]$.

Квадрат $D = \{(x_1, x_2) : -1 \le x_1 \le 1; -1 \le x_2 \le 1\}$ - область фазовой плоскости, на которой необходимо построить функцию цены.

Список литературы

1. С.В. Григорьева, А.А. Успенский, В.Н. Ушаков. Параллельный алгоритм вычисления функции цены дифференциальной игры // Алгоритмы и программные средства параллельных вычислений: [Сб.науч.тр.] Екатеринбург: УрО РАН, 1998. Вып. 2. С.86-94.

2. А.Ф. Сидоров, В.Л. Гасилов, А.П. Кукушкин. Разработка высокопроизводительных алгоритмических и программных средств на базе параллельных технологий // Алгоритмы и программные средства параллельных вычислений: [Сб. науч. тр.] Екатеринбург: УрО РАН, 1995. С. 3-20.

3. Самарский А.А. Теория разностных схем. М.: Наука, 1978. 656 с.

4. Яненко Н.Н., Коновалов А.Н., Бугров А.Н., Шустов Г.В. Об организации параллельных и «распараллеливании» прогонки // Численные методы механики сплошной среды. Новосибирск, 1978. Т.9, №7. С. 139-146.

5. В.А. Мальцев. Алгоритм распараллеливания матричной прогонки и его численная реализация // Алгоритмы и программные средства параллельных вычислений:[Сб. науч. тр.] Екатеринбург: УрО РАН, 1995. С. 115-124.

6. А.М. Тарасьев Аппроксимационные схемы построения минимаксных решений уравнений Гамильтона-Якоби // ПММ. 1994. Т.58, вып. 2. С. 22-36.

7. А.М. Тарасьев, А.А. Успенский, В.Н. Ушаков Аппроксимационные схемы и конечно-разностные операторы для построения обобщенных решений уравнений Гамильтона-Якоби // Изв. РАН. Техн. Кибернетика. 1994. № 3. С. 173-185.

8. Grigorjeva S.V., Tarasjev A.M., Ushakov V.N., Uspenskii A.A. Constructions of nonsmooth analysis in numerical methods for solving Hamilton-Jacobi Equations // Nova Jornal of mathematics, game theory, and algebra.1996. V. 6, N 1. P. 27-43.

9. Souganidis P.E. Approximation schemes for viscosity solutions of Hamilton-Jacobi equations // J. Different. Equat. 1985. V. 59, P. 1-43.

10. URL: http://cluster.linux-ekb.info/ (дата обращения: 11.01.11)

11. http://www.pdc.kth.se/training/Tutor/MPI/Templates/laplace/ (дата обращения: 14.03.11)

12. http://www.server-unit.ru/biblioteka/WKgyKfQfNK/ (дата обращения 20.03.11)

Приложение 1. Команды Unix

Справочную информацию о любой команде в Unix можно получить с помощью команды **man**:

```
man <имя изучаемой команды>
```

или

```
man -k <ключевое слово>
```

ls - выдать список файлов в текущем каталоге

cd [каталог] - сменить текущий каталог, если имя каталога не указывается, то текущим становится домашний каталог пользователя

cp <что_копировать> <куда_копировать> - копировать файлы

mv <что_перемещать> <куда_перемещать> - переместить или переименовать файл

rm <файлы> - удалить файлы

mkdir <каталог> - создать новый каталог

rmdir <каталог> - удалить пустой каталог

cat <имя_файла> - вывод содержимого файла на стандартный вывод (по умолчанию - на экран)

Можно записать вводимый текст в файл:

```
cat > <имя_файла>
```

more <имя_файла> - просмотр содержимого файла по страницам

less <имя_файла> - просмотр содержимого текстового файла с возможностью вернуться к предыдущим страницам. Нажмите q, чтобы выйти из режима просмотра файла.

tar -zxvf <файл> - распаковать архив **tgz** или **tar.gz**

find <каталог> -name имя_файла - найти файл с именем «имя файла» и отобразить результат поиска на экране. Поиск начинается с каталога <каталог>; «имя_файла» может содержать маску для поиска

./Имя_Программы - запустить на исполнение исполняемый файл в текущем каталоге

pwd - вывести имя текущего каталога

whoami - вывести имя, под которым Вы зарегистрированы

who — список работающих на кластере пользователей

ps a - вывести список текущих процессов в Вашем сеансе работы

top - интерактивный список текущих процессов, отсортированных по использованию центрального процессора

uname -a - вывести информацию о версии операционной системы

set|more - вывести текущие значения переменных окружения

echo $PATH - вывести значение переменной окружения "**PATH**"

Работа с сетью

scp <имя_файла_на_локальном_компьютере> <Ваше_имя_пользователя_на удаленной машине>**@**<имя_удаленной_машины>**:** - скопирует файл с локального компьютера в Вашу корневую директорию на удаленном компьютере («:» в конце команды обязательно).

hostname -i - показывает **IP** адрес компьютера, на котором Вы работаете.

Права доступа к файлам (каталогам)

chmod <права доступа> <файл> - изменить права доступа к файлу, владельцем которого вы являетесь

Есть три способа доступа к файлу:
чтение - read (r), запись - write (w), исполнение - execute (x)

и три типа пользователей:
владелец файла - owner (u), члены той же группы, что и владелец файла (g) и все остальные (o).

Поверить текущие права доступа можно следующим способом:

```
ls -l имя_файла
```

Если файл доступен всеми способами всем пользователям, то напротив имени файла будет следующая комбинация букв: **rwxrwxrwx**

Первые три буквы - это права доступа для владельца файла, второй триплет - права доступа для его группы, следующая тройка - права доступа для остальных. Отсутствие права доступа показывается как «-».; Например: Эта команда позволит вам установить права доступа на чтение для файла «junk» для всех (all=user+group+others):

`chmod a+r junk`

Эта команда отнимет право доступа на исполнение файла у всех кроме пользователя и группы:

`chmod o-x junk`

chown <новый_владелец> <файлы> - изменить владельца файлов

chgrp <новая_группа> <файлы> - изменить группу для файла

Контроль процессов

ps axu | grep <имя_пользователя> - отобразить все процессы, запущенные в системе от имени пользователя

kill –«убить» (завершить) процесс (сначала определите **PID** процесса при помощи **ps)**

killall <имя_программы> - «убить» (завершить) все процессы по имени программы

Встроенные в Linux программные утилиты и языки

gcc - GNU C компилятор, **g++** - GNU C++ компилятор

perl - язык для создания скриптов

python - современный объектно-ориентированный интерпретатор

gfortran - GNU FORTRAN компилятор Fortran 95

grep - поиск фрагмента текста в файлах, удовлетворяющего набранной маске

sed -утилита для обработки текстовых файлов

vi (vim) - текстовый (штатный) редактор системы Unix

Приложение 2. MC

Для работы с файлами удобно использовать файловый менеджер Midnight Commander (**mc**). Наберите в командной строке

```
mc
```

В окне терминала появятся две панели, отображающие списки файлов двух каталогов:

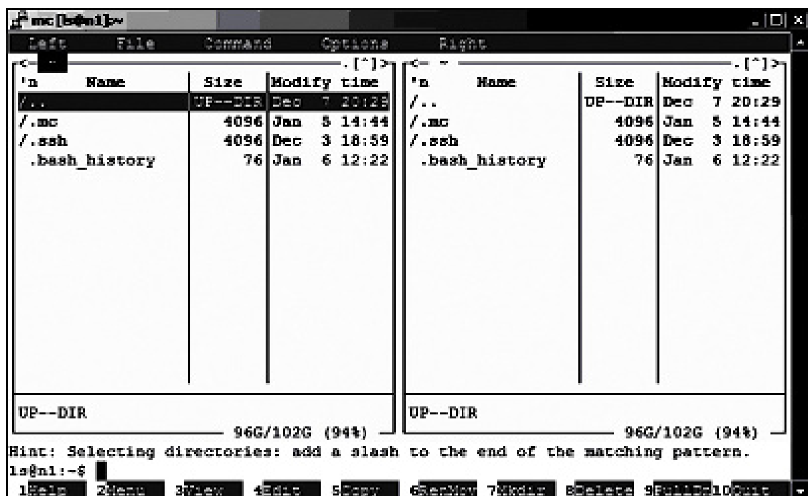

Рис. 11. Midnight Commander

Над панелями расположена строка меню, к выбору в этом меню можно переключиться при помощи клавиши <F9>.

Самая нижняя строка представляет собой ряд экранных кнопок, каждая из которых ассоциирована с одной из функциональных клавиш <F1> — <F10>.

Вторая снизу строка на экране — это командная строка, где можно ввести и выполнить любую команду системы.

Каждая панель состоит из заголовка, списка файлов какого-либо каталога и строки мини-статуса.

Одна из панелей является текущей (активной), о чем свидетельствует подсветка имени каталога в заголовке панели и подсветка одной из ее строк. Соответственно, в той оболочке, из которой была запущена программа Midnight Commander, текущим является каталог, отображаемый в активной панели. В этом каталоге и выполняются почти все операции.

На рис. 15 мы видим, что попали в свой домашний каталог (символ ~). Нажмите сочетание клавиш SHIFT+F4, откроется встроенный текстовый редактор:

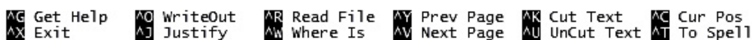

Рис. 12. Встроенный редактор

Наберите фразу «Hello world!» и нажмите сочетание клавиш CTRL+X. В нижней части терминала появится вопрос:

Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?

Нажмите клавишу **Y** (Yes). Появится строка-запрос имени нового файла. Наберите имя файла, нажмите **Enter**. Таким образом, Вы выйдете из встроенного редактора, и

снова на экране появятся две панели. В текущем каталоге появится новый только что созданный файл.

Только просмотр (но не редактирование) файла - <F3>, редактирование файла - <F4>. Создать каталог - <F7>, удалить файл или каталог - <F8>.

Операции типа копирования (<F5>) или переноса файла (<F6>) используют каталог, отображаемый на второй панели, в качестве целевого каталога (в который осуществляется копирование или перенос).

Чтобы выйти из **mc**, нажмите <F10>, появится диалоговое окно, нажмите **Enter** или **Y**, чтобы выйти, или стрелочку вправо+**Enter** или **N**, чтобы не покидать окно работы с **mc**.

Приложение 3. FTP

Чтобы скопировать файлы с локальной машины (ОС Windows), можно воспользоваться протоколом **FTP**.

Откройте **FAR** и нажмите **ALT-F1**(**ALT-F2**), в появившемся окне выберите **FTP** и нажмите **Enter**. Состояние активной панели поменяется – появится список всех сохраненных **ftp**-соединений. Нажмите **SHIFT-F4**, появится окно:

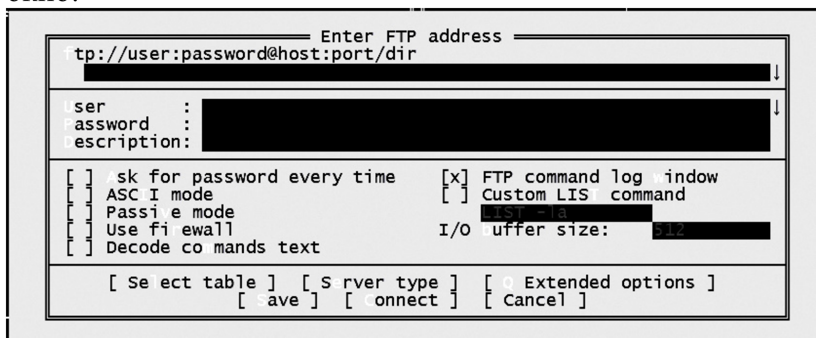

Рис. 13. Открытие соединения по **ftp**

Введите в верхнее поле kllogin@192.168.42.236, где **kllogin** – это Ваш логин на узле Кластера, и выберите **Save**.

Данное **ftp**-соединение сохранится, если подвести курсор к строке с этим **ftp**-соединением, появится окно для ввода пароля (пароль **ftp** = пароль для того, чтобы авторизоваться на узле Кластера).

Введите пароль и Вы окажетесь в Вашей домашней директории, куда можно будет скопировать файлы при помощи обычных команд **FAR**.

Приложение 4. Редактор vi

История создания редактора vi

Первая версия редактора **vi** была написана Биллом Джоем в 1976 году. В то время наиболее распространённым был редактор **ed**, который был создан для примитивных терминалов типа пишущих машинок и телетайпов.

Они работали в режиме командной строки. В режиме командной строки можно было ввести одну команду с несколькими параметрами. Для ускорения работы команды состояли, в основном, из одной или двух букв. Хотя с современной точки зрения это был очень примитивный редактор, но в то время это был большой прогресс по сравнению с исправлением ошибок путем перебивки перфокарт. Редактор **ed** стал составной частью операционной системы UNIX, которая создавалась в это же время.

Поскольку он был довольно сложным для «простого смертного», George Coulouris разработал редактор **em** (editor for mortals — редактор для смертных). Билл Джой модифицировал редактор **em** и назвал его **en**, а позднее — он получил название **ex**, на котором и основан **vi**. Во многих современных системах Unix он имеет еще название **vim** (VI Improved).

Режимы работы редактора vi

Командный режим

В отличие от многих привычных редакторов, **vi** имеет несколько режимов работы (модальный интерфейс). Это означает, что одни и те же клавиши в разных режимах работы выполняют разные действия.

В редакторе **vi** есть два основных режима: командный режим (normal) и режим вставки или режим командной строки (cmdline).

По умолчанию, работа начинается в командном режиме. В этом режиме вводимые команды не отображаются на экране, а сразу же выполняются. Например, команда **dd** удаляет текущую строку текста, т. е. ту строку, в которой находится курсор. Команда **x** удаляет один символ, а команда **r** заменяет один символ, на котором стоит курсор и т. д.

В командном режиме четыре алфавитные клавиши **h**, **j**, **k**, **l** перемещают курсор на одну позицию влево, вниз, вверх, вправо соответственно, то есть выполняют те же функции, что и стрелки на функциональной клавиатуре.

Информацию о командах в разных режимах можно посмотреть с помощью команды help. Для того чтобы посмотреть информацию о командах в разных режимах, нужно набрать на клавиатуре двоеточие, затем h[elp] (достаточно одной буквы **h**), и затем после пробела то, о чем мы хотим получить справку, в нашем случае слово **vim-mode**:

```
:h vim-mode
```

Откроется новое окно, в котором будет текст по данному запросу. В этом окне можно перемещаться обычным образом, с помощью стрелок, и клавиш функциональной клавиатуры. При этом пользователь может получить любую другую информацию, перемещаясь в окне help'а вверх или вниз.

Если это окно больше не нужно, то его можно закрыть, нажав одновременно клавиши Ctrl-Z-Q. Другой способ закрыть окно - набрать команду

```
:q
```

в командной строке.

Если вы не знаете, в каком режиме находится редактор, но хотите перейти в командный режим, то нужно два раз нажать клавишу **Esc**.

Режим ввода командной строки

В режим ввода командной строки (cmdline) можно перейти из командного режима (normal) следующим образом:

: (двоеточие), после которого может следовать любая команда, например, **:help** или **:quit**

/ (наклонная черта) или **?** (знак вопроса), после которых следует регулярное выражение (любая последовательность символов, которую нужно найти в тесте)

:! (двоеточие и восклицательный знак), после которого следует фильтр или любая команда Unix (к примеру **:!date**). Чтобы вернуться назад в редактор, нужно нажать клавишу **Enter**.

Для перехода из режима командной строки (cmdline) назад в командный режима (normal) нужно нажать два раза подряд клавишу **Esc** или **Ctrl-C**.

Режим ввода текста

В режим ввода текста (insert) можно перейти из командного режима, нажав клавишу **i** или **Insert**. При этом в левом нижнем углу экрана появится слово INSERT (или ВСТАВКА, если ваша система русифицирована).

Если еще раз нажать клавишу **Insert**, то в левом нижнем углу появится слово REPLACE или ЗАМЕНА. Это означает, что при наборе текста он будет затирать существующий текст. Если и дальше нажимать клавишу **Insert**, то режимы вставки и замены будут чередоваться.

Визуальный режим

Визуальный режим (visual) позволяет использовать все преимущества экранного терминала - это естественный и основной режим всех современных редакторов текста. Вы

можете выделить нужный вам участок (блок) текста путем перемещения курсора. Выделенный текст будет подсвечен. После окончания выделения, можно выполнять различные операции с выделенным блоком: копировать, перемещать, удалять и т. д.

В визуальный режим из обычного режима можно перейти тремя способами, которые соответствуют трем различным видам выделяемых блоков. Для выделения блоков текста используются следующие три команды:

v - начало выделения посимвольного блока (непрерывная цепочка символов) подсвечивается при дальнейшем перемещении курсора

Shift-V - выделение строчного блока. При дальнейшем перемещении курсора вверх или вниз, выделяется и подсвечивается соответствующий строчный блок

Ctrl-V — начало выделение вертикального блока. При перемещении курсора влево/вправо и вверх/вниз выделяется соответствующий вертикальный блок.

Чтобы выделенный блок попал в буфер редактора, используйте команды **y** или **x**. Если нажать клавишу **y**, выделенный блок попадает в буфер, но не удаляется из текста. Если нажать клавишу **v**, то подсвеченный блок гаснет, и произойдет отмена визуального режима.

Если нажать клавишу **x**, выделенный блок попадает в буфер и удаляется из текста. Он будет оставаться в буфере до тех пор, пока его не заменит какая-либо другая команда, которая использует для своей работы буфер обмена.

Находящийся в буфере блок, можно многократно вставлять в нужные места текста с помощью команды **p** или **P**. Команда **P** вставляет блок перед курсором, а команда **p** — после курсора.

Вызов редактора

Редактор **vi** можно вызвать из командно строки терминала различными способами. Общая форма вызова следующая:

```
vi [options] [file ...]
```

Квадратные скобки означают необязательные опции или файлы. Можно начинать работу с редактором с явным указанием имени редактируемого файла:

```
vi file_name
```

где file_name — имя файла, с которым вы хотите работать. Если такого файла нет в текущей директории, то он будет создан при выходе, а при входе вы увидите в открывшемся окне пустой экран.

Если же файл с указанным именем есть в текущей директории, то вы увидите в открывшемся окне текст этого файла, и после этого его можно начать редактировать.

Довольно часто возникает необходимость перенести информацию из одного файла в другой, или сравнить содержимое двух файлов. Для этого нужно вызвать редактор следующей командой:

```
vi -o2 file1 file2
```

Одновременно откроются два окна. Перейти из одного окна в другое можно при помощи команды **Ctrl-W-W**. Аналогично можно работать с тремя и более файлами.

Если текст в файле имеет короткие строки (текст программы на ассемблере), то в этом случае можно открыть два вертикальных окна. Для этого вместо малой буквы **o** нужно набрать большую букву **O**:

```
vi -O2 file1 file2
```

Закрыть окно, в котором находится курсор, можно при помощи команды **:q** или **Ctrl-Z-Z**. Открыть новое окно можно с помощью команды **Ctrl-W-N**.

Выход из редактора

Чтобы выйти из редактора с сохранением результатов, используйте команду **:wq** или **Ctrl-Z-Z**. Чтобы выйти из редактора без сохранения результатов редактирования, используйте команду **:q!** или **Ctrl-Z-Q**. Если файл был открыт только для посмотра, то выйти из редактора можно набрав команду **:q** или **Ctrl-Z-Q**.

Поиск и замена текста

Наиболее частая и важная операция при редактировании больших файлов - это контекстный поиск и замена одного фрагмента текста другим. Для этого у редактора **vi** имеется большая группа команд с использованием регулярных выражений. Для поиска текста можно использовать команду

```
/string
```

где **string** может содержать произвольный текст, в том числе и знаки препинания. После того, как вы набрали команду и нажали клавишу **Enter**, курсор остановится на начале указанного фрагмента текста **string**. Если же такого фрагмента нет в тексте, то в левом нижнем углу будет сообщение:

```
E486: Шаблон не найден: string
```

Дальнейший поиск заданного фрагмента текста можно выполнить с помощью команды **n**. Если будет достигнут конец файла, то в командной строке будет сообщение:

```
поиск будет продолжен с КОНЦА документа
```

Если вместо малой буквы **n** использовать большую букву **N**, то поиск будет произведен в обратном направлении, т. е. по направлению к началу файла. Замена текста может быть выполнена вручную.

Существует мощная команда автоматическоой замены текста:

```
:[range]s/old_text/new_text/[g]
```

Заменить **old_text** на **new_text** в указанном диапазоне строк **range**. Параметры **new_text** и **old_text** могут быть регулярными выражениями, а **range** задает диапазон строк текста, для которых будет выполнены эта команда. Например, если **range** задать в виде **10,20**, то замена будет выполнена только в строках **10-20**.

Если **range** задать в виде **%**, то замена будет выполнена для всех строк. Если **range** задать в виде **20,$**, то замена будет выполнена от 20-й строки до конца файла.

Необязательный параметр **g** нужен, чтобы заменить все фрагменты текста в текущей строке, иначе будет замена только первого фрагмента в строке.

Использование регулярных выражений

Регуля́рные выраже́ния (англ. regular expressions, сокр. RegExp) — это формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. wildcard characters). По сути это строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

Например, при помощи регулярных выражений можно задать шаблоны, позволяющие:

- найти все последовательности символов *«кот»*: *«кот», «котлета», «терракотовый»*;
- найти отдельно стоящее слово *«кот»* и заменить его на *«кошка»*;
- найти слово *«кот»*, которому предшествует слово *«персидский»* или *«чеширский»*;
- убрать из текста все предложения, в которых упоминается слово *кот* или *кошка*.

Регулярные выражения позволяют задавать и гораздо более сложные шаблоны поиска или замены.

/^$/ — пустая строка, т.е. только конец строки
/./ — непустая строка, по крайней мере, один символ
/^/ — все строки
/thing/ — thing где-либо в строке
/^thing/ — thing в начале строки
/thing$/ — thing в конце строки
/^thing$/ — строка, состоящая лишь из thing
/thing.$/ — thing плюс любой символ в конце строки
/\/thing\// — /thing/ где-либо в строке
/[tT]hing/ — thing или Thing где-либо в строке
/thing[0-9]/ — стр. thing, за которой идет одна цифра
/thing[^0-9]/ — стр. thing, за которой идет не цифра
/thing1.*thing2/ —thing1, затем любая строка, thing2
/^thing1.*thing2$/ — thing1 в начале, thing2 в конце

Основные команды vi

Команды удаления

x удалить символ, на котором стоит курсор
X удалить символ перед курсором
D удалить символ от курсора до конца строки
d0 удалить символ от курсора до начала строки
dw удалить слово
dd удалить строку
5dd удалить 5 строк

Команды вставки

p вставить удаленный текст из буфера после курсора
P вставить удаленный текст из буфера перед курсором

Команды замены

r замена одного символа, на котором стоит курсор. Символ для замены нужно ввести сразу после команды **r**. При этом редактор остается в командном режиме.

R перевод редактора в режим постоянной замены. При этом появляется индикатор (ЗАМЕНА) в левом нижнем углу экрана. Переход в командный режим – **Esc**.

Команды добавления

a добавить текст после курсора

A добавить текст в конец строки

i добавить текст перед курсором

I добавить текст в начало строки

o добавить новую строку после текущей строки

O добавить новую строку перед текущей строкой

Во время выполнения любой из этих команд редактор переходит из командного режима в режим вставки текста (ВСТАВКА). Для возврата в командный режим нужно нажать клавишу **Esc**.

Команды перемещения по тексту

G на последнюю строку

GG на первую строку

10G на десятую строку

:33 переход на 33 строку

Отмена предыдущей команды (откат)

u отмена предыдущей команды (будет отменять ранее выполненные команды, до тех пор, пока не будет исчерпан буфер с историей команд).

U восстановить текущую строку в прежнем состоянии (отмена отката).

Команды чтения и записи файлов

Команды чтения и записи файлов выполняются в режиме командной строки (cmdline), нажмите клавишу **:** (двоеточие), а затем наберите нужную команду.

:r file_name чтение и вставка в текущую позицию курсора текста из файла file_name

:w file_name запись текущего состояния редактируемого файла в файл file_name

:w! file_name повторная запись редактируемого файла под тем же именем. Если не использовать символ **!** (восклицательный знак), то появляется сообщение, что такой файл уже существует (предупреждение о возможном затирании файла), и записи файла не произойдет.

:wq запись редактируемого файла под именем, которое было в самом начале редактирования, следующая за **w** команда **q** означает выход из редактора.

Замена строчных букв на прописные

[Visual]U замена малых букв большими

[Visual]u замена больших букв на малые

Команды выхода из редактора

:q выход, если файл был открыт только для просмотра;

:q! выход без сохранения результатов редактирования;

:wq выход с сохранением результатов редактирования;

Ctrl-Z-Q выход без сохранения в командном режиме;

Ctrl-Z-Z выход с сохранением в командном режиме.

Приложение 5. Структура параллельной программы

Любая исполняемая на кластерном компьютере программа должна состоять из стандартного набора алгоритмических блоков [10].

```
1.   program test
2.   <инициализация параллельной среды>
3.   N=<номер текущего процессора>
4.   <загрузка массивов данных для процесса N>
5.   while(<задача не решена>) do
6.       <получение и передача граничных данных>
7.       <выполнение очередной итерации>
8.       if(<контрольная точка>) then
9.           <сохранение данных на диск>
10.  done
11.  <деинициализация параллельной среды>
12.  end
13.  stop
```

Строка 1. Начало программы.

Строка 2. Операции для подготовки к работе параллельной вычислительной среды.

Строка 3. Перед выполнением основного кода нужно понять, на каком процессоре выполняется данная программа, поскольку от этого зависит, что будет дальше. Каждый отдельный процесс параллельной программы выполняет либо собственные специфические действия, либо обрабатывает собственную порцию общего массива данных, либо и то, и другое.

Строка 4. На этом этапе программа должна подготовить для дальнейшей обработки массивы данных, загрузив в них информацию для процесса, номер которого мы получили в предыдущем шаге.

Строка 5. Начало итерационного процесса. Здесь по каким-то критериям определяется, достигнут ли желаемый результат, и не пора ли завершить программу.

Строка 6. Перед выполнением собственно итерации, необходимо получить данные от соседних процессов для граничных областей локальных массивов.

Строка 7. Собственно обработка массивов данных в соответствии с задачами, возложенными на процесс номер **N**, то есть очередной шаг итерации.

Строка 8. После завершения очередной порции вычислений нужно результаты на диск. То есть, не достигли ли мы контрольной точки. Как часто это делать - решать программисту. Однако разумным будет сохранять данные не на каждой итерации, а гораздо реже. Так, чтобы время, необходимое для записи на диск, было много меньше, чем длительность итераций между контрольными точками.

Строка 9. Если достигнута контрольная точка, то начинается процедура сохранения данных. Причем, выполнять запись данных на диск необходимо (по мере возможности) в асинхронном режиме.

Строка 10. Конец итерационного цикла.

Строка 11. В конце работы программы мы должны корректно завершить работу параллельной среды. Здесь же можно сохранить на диск финальные данные.

Приложение 6. MPI

MPI (Message passing interface)--интерфейс передачи сообщений, обеспечивающий единый механизм взаимодействия процессов внутри параллельно исполняемой задачи независимо от машинной архитектуры (однопроцессорные, многопроцессорные с общей или раздельной памятью), взаимного расположения процессов (на одном физическом процессоре или на разных) и **API** операционной системы [10].

Программа, использующая **MPI**, легко отлаживается и переносится на другие платформы, часто для этого достаточно простой перекомпиляции исходного текста программы.

Несмотря на то, что **MPI** представляет собой значительный шаг вперед по сравнению с предшествующим поколением библиотек передачи сообщений, а, возможно и вследствие этого, программировать на **MPI** достаточно сложно, причиной тому является не недостаток стандарта, а в самой идеологии передачи сообщений. **MPI** можно рассматривать как уровень ассемблера для параллельных программ.

Основное отличие стандарта MPI от его предшественников - понятие коммуникатора. Все операции синхронизации и передачи сообщений локализуются внутри коммуникатора. С коммуникатором связывается группа процессов. В частности, все коллективные операции вызываются одновременно на всех процессах, входящих в эту группу. Поскольку взаимодействие между процессами инкапсулируется внутри коммуникатора, на базе **MPI** можно создавать библиотеки параллельных программ.

В настоящее время разными коллективами разработчиков написано несколько программных пакетов, удовлетворяющих спецификации **MPI**, в частности: MPICH, LAM, HPVM, OpenMPI (в нашем случае OpenMPI) и так далее.

MPI -- это хорошо стандартизованный механизм для построения программ по модели обмена сообщениями, существуют стандартные «привязки» **MPI** к языкам C/C++, Fortran 77/90, за стандартизацию **MPI** отвечает MPI Forum (http://www.mpi-forum.org).

Основные понятия MPI. Парадигма SPMD

При запуске задачи создается группа из **P** процессов. Группа идентифицируется целочисленным дескриптором (коммуникатором). Внутри группы процессы нумеруются от **0** до **P-1**. В ходе решения задачи исходная группа (ей присвоено имя **MPI_COMM_WORLD**) может делиться на подгруппы, подгруппы могут объединяться в новую группу, имеющую свой коммуникатор. Таким образом, процесс имеет возможность одновременно принадлежать нескольким группам процессов. Каждому процессу доступен свой номер **myProc** внутри любой группы, членом которой он является.

Поведение всех процессов описывается одной и той же программой. Коммуникации между процессами в ней программируются явно с использованием библиотеки **MPI**, которая и диктует стандарт программирования.

Квазиодновременный запуск исходной группы процессов производится средствами операционной системы. При этом **P** определяется желанием пользователя, а отнюдь не количеством доступных процессоров!

Итак, все **P** процессов асинхронно выполняют одну и ту же программу. Но у каждого из них свой номер **myProc**. Поэтому в программе, естественно, будут такие фрагменты:

```
if (myProc.eq.0) then
   < делать что-то одно >
else if (myProc.eq.1) then
   < делать что-то другое >
         . . .
```

```
else
   < делать что-то P-е >
endif
```

Таким образом, в программе «под одной крышей» закодировано поведение всех процессов. В этом и заключена парадигма программирования Single Program - Multiple Data (**SPMD**).

Обычно поведение процессов одинаково для всех, кроме одного, который выполняет координирующие функции, не отказываясь, впрочем, взять на себя и часть общей работы. В качестве координатора обычно выбирают процесс с номером **0**.

Общая организация MPI

MPI -- это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений, состоящая примерно из 130 функций, в число которых входят следующие функции:

- функции инициализации и закрытия MPI процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Каждая из **MPI** функций характеризуется способом выполнения:

1. Локальная функция - выполняется внутри вызывающего процесса, ее завершение не требует коммуникаций.
2. Нелокальная функция - для ее завершения требуется выполнение **MPI**-процедуры другим процессом.
3. Глобальная функция - процедуру должны выполнять все процессы группы, несоблюдение этого условия может приводить к зависанию задачи.

4. Блокирующая функция - возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.

5. Неблокирующая функция - возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

Базовые функции MPI

Любая прикладная **MPI**-программа должна начинаться с вызова функции инициализации **MPI** - функции **MPI_Init**. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения.

Процессы в группе упорядочены и пронумерованы от **0** до **groupsize-1**, где **groupsize** равно числу процессов в группе. Кроме этого, создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса. Синтаксис функции инициализации **MPI_Init**:

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

В программах на языке FORTRAN параметр IERROR является выходным и возвращает код ошибки.

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

```
MPI_FINALIZE(IERROR)
INTEGER IERROR
```

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

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

MPI_COMM_SIZE(COMM, SIZE, IERROR)

INTEGER COMM, SIZE, IERROR

Функция возвращает количество процессов в области связи коммуникатора **comm**.

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

Функция определения номера процесса MPI_Comm_rank

MPI_COMM_RANK(COMM, RANK, IERROR)

INTEGER COMM, RANK, IERROR

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

В минимальный набор следует включить также две функции передачи и приема сообщений.

Функция передачи сообщения **MPI_Send**

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

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

Функция приема сообщения **MPI_Recv**

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

<type> BUF(*)

INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,

STATUS(MPI_STATUS_SIZE), IERROR

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

Приложение 7. Программная реализация решения уравнения Лапласа

В данном приложении приведен пример параллельной программы решения дифференциального уравнения Лапласа в двумерном случае [11].

```
/*-----------------------------------------------------
--------------------
parallel.laplace.c
Finite Difference (parallel) program
Author: Robb Newman
Converted to MPI: 11/12/94 by Xianneng Shen
Last revised: 3/14/97 RYL

        This   program   uses   a   finite   difference
scheme to solve Laplace's equation for a square ma-
trix distributed over a square (logical) processor
topology.  A complete description of the algorithm
is found in Fox, et al "Solving problems on concur-
rent Processors, Volume 1: General Techniques and
Regular Problems, Prentice Hall,
Englewood Cliffs, New Jersey.

        This program works on the SPMD (single pro-
gram, multiple data) paradigm.  It illustrates 2-d
block decomposition, nodes exchanging edge values,
and convergence checking.
        Each matrix element is updated based on the
values  of  the  four  neighboring  matrix  elements.
This process is repeated until the data converges -
- until  the  average  change  in  any  matrix  element
(compared to the value 20 iterations previous) is
smaller than a specified value.
        To   ensure   reproducible   results   between
runs, a red/black checkerboard algorithm is used.
Each  process  exchanges  edge  values  with  its  four
neighbors.  Then new values are calculated for the
upper left and lower right corners (the "red" cor-
ners) of each node's matrix.  The processes ex-
change  edge  values  again.   The  upper  right  and
```

```
lower left corners (the "black" corners) are then
calculated.

        The program is currently configured for a
48x48 matrix distributed over four processors.  It
can be edited to handle different matrix sizes or
number of processors, as long as the matrix can be
divided evenly between the processors.
---------------------------------------------------
--------------------*/

#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define n 48           /* matrix is nxn, excluding
boundary values     */
#define nodeedge 24    /* a task works on a
nodeedge x nodeedge matrix */
#define nblock n/nodeedge   /* number of tasks per
row of matrix            */
#define nproc nblock*nblock /* total number of
tasks (processors)           */
#define w (double)1.2      /* convergence factor */

void setmatrix(double M[][nodeedge+2]);
void setcomm(int rank, int comm[]);
void iterate(double M[][nodeedge+2],double re-
sult[][n],int rank,int comm[]);
void setex(double ex[], double M[][nodeedge+2], int
which);
void unpack(double M[][nodeedge+2],int where,double
in[]);
void exchange(double M[][nodeedge+2], int comm[],
int rank);
void dored(double M[][nodeedge+2]);
void doblack(double M[][nodeedge+2]);

main(int argc, char **argv)
{
  double M[nodeedge+2][nodeedge+2], /* my block of
the array                */
```

```
result[n][n], /* will store all final values    */
start, etime; /* initial, elapsed wallclock time
(sec)*/
int    ntasks, /* number of tasks started     */
rank, /* rank (process ID)                    */
comm[4], /* directions requiring communication   */
i,j;
FILE   *fp; /* file pointer to the output file  */

  MPI_Init(&argc, &argv);
  start = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
  if (ntasks != nproc) {
    if (rank == 0) printf("error: MP_PROCS should
be set to %i!\n",nproc);
    exit(1);
  }
  if (rank==0) printf("Everything's okay so
far\n");

  /* set initial values of M */
  setmatrix(M);
  if (rank==0) printf("Matrix is set\n");

  /* figure out who I communicate with */
  setcomm(rank, comm);
  if (rank==0) {
    printf("Communications are set up\n");
    printf("Beginning to iterate\n");
  }

  /* update M until convergence */
  iterate(M, result, rank, comm);

  /* report results and timing */
  etime = (MPI_Wtime() - start);
  printf("task %i took %6.3f seconds\n", rank,
etime);
  if (rank == 0) {
    fp=fopen("parallel.laplace.out", "w");
```

```
    for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
        fprintf(fp,"%f \n", result[i][j]);
      }
      fprintf(fp, "\n");
    }
    fclose(fp);
  }
  MPI_Finalize();
}


/*-------------------------------------------------
------------------------
  setmatrix sets the initial values of my piece of
the matrix
  -------------------------------------------------
------------------------*/
void setmatrix(double M[][nodeedge+2])

{
  int i,j;
  double bv[4], avg;

  bv[0]=100.0;
  bv[1]=0.0;
  bv[2]=0.0;
  bv[3]=0.0;

  /* put boundary values in edge vectors of M */
  for (i=1;i<=nodeedge;i++) {
    M[0][i]=         bv[0];
    M[i][nodeedge+1]=bv[1];
    M[nodeedge+1][i]=bv[2];
    M[i][0]=         bv[3];
  }

  /* set all interior values to be the average of
the boundary values */
  avg=(bv[0]+bv[1]+bv[2]+bv[3])/4.0;
  for (i=1;i<=nodeedge;i++)
```

```
    for (j=1;j<=nodeedge;j++)
      M[i][j]=avg;
}


/*-------------------------------------------------------
------------------------
  setcomm figures out what directions I must commu-
nicate with (1=yes, 0=no)
  comm[0]=up, comm[1]=right, comm[2]=down,
comm[3]=left
  -----------------------------------------------------
------------------------*/
void setcomm(int rank, int comm[])

{
  int i;

  for (i=0; i<4; i++) comm[i] = 1;
  if (rank < nblock) comm[0] = 0;
  if (fmod((double)(rank+1), (double)nblock) ==
0.0) comm[1] = 0;
  if (rank > (nblock*(nblock-1)-1)) comm[2] = 0;
  if (fmod((double)rank, (double)(nblock)) == 0.0)
comm[3] = 0;
}


/*-------------------------------------------------------
------------------------
  iterate takes care of everything to do with it-
eration, including the convergence checking
  -----------------------------------------------------
------------------------*/
void iterate(double M[][nodeedge+2], double re-
sult[][n],
             int rank, int comm[])

{
  int    it, /* current iteration         */
  i,j,k,l,index, /* index variables       */
```

```
  done,          /* loop control variable */
  count;  /* length (in elements) of messages  */
  double mold[nodeedge+2][nodeedge+2],  /* my ma-
trix, 20 iterations ago      */
/*   used to check convergence        */
  diff,  /* the average absolute difference in ele-
ments   */
/*   of M and mold                                */
  in;  /* sum of diff from each task          */
  double MM[n*n], /* initially holds results from
all tasks        */
  ediff,              /* the difference in two ele-
ments of M and mold  */
  send[nodeedge][nodeedge];  /* the result I will
send to task 0      */

  it=0;
  done=0;
  for (i=1;i<=nodeedge;i++)
    for (j=1;j<=nodeedge;j++)
      mold[i][j]=M[i][j];

  while (done==0) {
    it++;
    /* exchange values with neighbors, update red
squares, exchange values */
    /* with neighbors, update black squares */
    exchange(M, comm, rank);
    dored(M);
    exchange(M, comm, rank);
    doblack(M);

    /* check for convergence every 20 iterations */
    /* find the average absolute change in elements
of M */
    /* maximum iterations is 5000 */
    if (it>5000) done=1;
    if ((fmod((double)it, (double)20.0) == 0.0) &&
(done != 1)) {
      diff = 0.0;
      for (i=1; i<=nodeedge; i++)
```

```
        for (j=1; j<=nodeedge; j++) {
          ediff = M[i][j] - mold[i][j];
          if (ediff < 0.0) ediff = -ediff;
          diff += ediff;
          mold[i][j] = M[i][j];
        }
      diff = diff/((double)(nodeedge*nodeedge));
      MPI_Allreduce(&diff, &in, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
      if (in < (double)nproc*.001) done = 1;
    }
  }

  /* send results to task 0 */
  /* don't need to include boundary points */
  for (i=0;i<nodeedge;i++)
    for (j=0;j<nodeedge;j++)
      send[i][j] = M[i+1][j+1];
  count = nodeedge*nodeedge;
  MPI_Gather(&send, count, MPI_DOUBLE, &MM,
             count, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  printf("I finished gather\n");

  /* storage on task 0 has to be consistent with a
nblock x nblock */
  /* decomposition */
  if (rank ==0) {
    printf("did %i iterations\n",it);
    index=0;
    for (k=0;k<nblock;k++)
      for (l=0;l<nblock;l++)
        for (i=k*nodeedge;i<(k+1)*nodeedge;i++)
          for (j=l*nodeedge;j<(l+1)*nodeedge;j++) {
            result[i][j]=MM[index];
            index++;
          }
  }
}
```

```
/*------------------------------------------------------
------------------------
  pulls off the edge values of M to send to another
task
  ------------------------------------------------------
------------------------*/
void setex(double ex[], double M[][nodeedge+2], int
which)

{
  int i;
  switch (which)
  {
    case 0: {
      for (i=1; i<=nodeedge; i++) ex[i-1] =
M[1][i];
      break; }
    case 1: {
      for (i=1; i<=nodeedge; i++) ex[i-1] =
M[i][nodeedge];
      break; }
    case 2: {
      for (i=1; i<=nodeedge; i++) ex[i-1] =
M[nodeedge][i];
      break; }
    case 3: {
      for (i=1; i<=nodeedge; i++) ex[i-1] =
M[i][1];
      break; }
  }
}


/*------------------------------------------------------
------------------------
  unpack puts the vector of new edge values I just
received into the edges of M
  ------------------------------------------------------
------------------------*/
void unpack(double M[][nodeedge+2],int where,double
in[])
```

```
{
  int i;
  switch (where)
  {
    case 0: {
      for (i=0; i<nodeedge; i++) M[0][i+1] = in[i];
      break; }
    case 1: {
      for (i=0; i<nodeedge; i++) M[i+1][nodeedge+1]
= in[i];
      break; }
    case 2: {
      for (i=0; i<nodeedge; i++) M[nodeedge+1][i+1]
= in[i];
      break; }
    case 3: {
      for (i=0; i<nodeedge; i++) M[i+1][0] = in[i];
      break; }
  }
}


/*---------------------------------------------------
------------------------
  makes the communication calls for each task to
exchange edge values with up  to four neighbors
  ---------------------------------------------------
------------------------*/
void exchange(double M[][nodeedge+2], int comm[],
int rank)

{
  double ex0[nodeedge], ex1[nodeedge],  /* vectors
being sent            */
         ex2[nodeedge], ex3[nodeedge];
  double in0[nodeedge], in1[nodeedge],  /* vectors
being received        */
  in2[nodeedge], in3[nodeedge];
  int tag, /* message tag                   */
           /* 0=up, 1=right, 2=down, 3=left */
```

```
  partner, /* who I am exchanging with      */
  i;
  MPI_Request requests[8]; /* communication identi-
fier     */
  MPI_Status status[8]; /* status of completed com-
munication */

  /* initialize requests */
  for (i=0; i<8; i++) requests[i] =
MPI_REQUEST_NULL;

  /* receive incoming messages */
  if (comm[0]==1) {                          /*
receive from above */
    partner=rank-nblock;
    tag=0;
    MPI_Irecv(&in0, nodeedge, MPI_DOUBLE, partner,
tag, MPI_COMM_WORLD,
              &requests[0]);
  }
  if (comm[1]==1) {                          /*
receive from right */
    partner=rank+1;
    tag=1;
    MPI_Irecv(&in1, nodeedge, MPI_DOUBLE, partner,
tag, MPI_COMM_WORLD,
              &requests[1]);
  }
  if (comm[2]==1) {                          /*
receive from below */
    partner=rank+nblock;
    tag=2;
    MPI_Irecv(&in2, nodeedge, MPI_DOUBLE, partner,
tag, MPI_COMM_WORLD,
              &requests[2]);
  }
  if (comm[3]==1) {                          /*
receive from left */
    partner=rank-1;
    tag=3;
```

```
    MPI_Irecv(&in3, nodeedge, MPI_DOUBLE, partner,
tag, MPI_COMM_WORLD,
              &requests[3]);
  }

  /* send messages to my partners */
  if (comm[0]==1) {                          /*
send up */
    partner=rank-nblock;
    setex(ex0,M,0);
    MPI_Isend(&ex0, nodeedge, MPI_DOUBLE, partner,
2, MPI_COMM_WORLD,
              &requests[4]);
  }
  if (comm[1]==1) {                          /*
send right */
    partner=rank+1;
    setex(ex1,M,1);
    MPI_Isend(&ex1, nodeedge, MPI_DOUBLE, partner,
3, MPI_COMM_WORLD,
              &requests[5]);

  }
  if (comm[2]==1) {                          /*
send down */
    partner=rank+nblock;
    setex(ex2,M,2);
    MPI_Isend(&ex2, nodeedge, MPI_DOUBLE, partner,
0, MPI_COMM_WORLD,
              &requests[6]);
  }
  if (comm[3]==1) {                          /*
send left */
    partner=rank-1;
    setex(ex3,M,3);
    MPI_Isend(&ex3, nodeedge, MPI_DOUBLE, partner,
1,  MPI_COMM_WORLD,
              &requests[7]);
  }

  /* wait for all communication to complete */
```

```
  MPI_Waitall(8, requests, status);

  if (comm[0] == 1) unpack(M, 0, in0);   /* put msg
from above into matrix */
  if (comm[1] == 1) unpack(M, 1, in1);   /* put msg
from right into matrix */
  if (comm[2] == 1) unpack(M, 2, in2);   /* put msg
from below into matrix */
  if (comm[3] == 1) unpack(M, 3, in3);   /* put msg
from left  into matrix */
}
/*--------------------------------------------------
-----------------------
  iterates on the upper left and lower right cor-
ners of my matrix
  -------------------------------------------------
------------------------*/
void dored (double M[][nodeedge+2])

{
  int i,j;

  for (i=1; i<=nodeedge/2; i++)                  /*
upper left */
    for (j=1; j<=nodeedge/2; j++)
      M[i][j]=w/4.0*(M[i-1][j]+M[i][j-
1]+M[i+1][j]+M[i][j+1])+(1.0-w)*M[i][j];
  for (i=nodeedge/2+1; i<=nodeedge; i++)        /*
lower right */
    for (j=nodeedge/2+1; j<=nodeedge; j++)
      M[i][j]=w/4.0*(M[i-1][j]+M[i][j-
1]+M[i+1][j]+M[i][j+1])+(1.0-w)*M[i][j];
}
/*--------------------------------------------------
-----------------------
  iterates on the upper right and lower left cor-
ners of my matrix
  -------------------------------------------------
------------------------*/
void doblack(double M[][nodeedge+2])
```

```
{
  int i,j;
  for (i=1; i<=nodeedge/2; i++)
/* upper right */
    for (j=nodeedge/2+1;j<=nodeedge;j++)
      M[i][j]=w/4.0*(M[i-1][j]+M[i][j-
1]+M[i+1][j]+M[i][j+1])+(1.0-w)*M[i][j];
  for (i=nodeedge/2+1; i<=nodeedge; i++)        /*
lower left */
    for (j=1;j<=nodeedge/2;j++)
      M[i][j]=w/4.0*(M[i-1][j]+M[i][j-
1]+M[i+1][j]+M[i][j+1])+(1.0-w)*M[i][j];
}
```

```
makefile
mpcc -g parallel.laplace.c -o parallel.laplace
${CFLAGS} -lm
```

Учебное издание

Составители: Михаил Аркадьевич Клочков,
Константин Юрьевич Марков,
Юрий Степанович Митрохин,
Любовь Сергеевна Чиркова

Организация параллельных вычислений для решения дифференциальных уравнений на blade-сервере

Учебно-методическое пособие

Отпечатано с оригинал-макета заказчика
Подписано в печать 11.05.11. Формат 60х84 $^{1/16}$.
Печать офсетная. Усл. печ. л. 4,6 Уч.-изд. л. 2,6
Заказ №   Тираж 30 экз.
Издательство «Удмуртский университет»
426034, Ижевск, Университетская, 1, корп. 4.