МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

Е.С. САГАТОВ, П.Ю. ЯКИМОВ

LINUX В СУПЕРКОМПЬЮТЕРНЫХ СИСТЕМАХ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для студентов, обучающихся по основным образовательным программам высшего образования по направлениям подготовки 09.03.01 Информатика и вычислительная техника, 01.03.02 Прикладная математика и информатика, 01.04.02 Прикладная математика и информатика, 24.04.02 Системы управления движением и навигация, 12.04.04 Биотехнические системы и технологии

САМАРА
Издательство Самарского университета
2018

УДК 004.382.2(075), 004.451(075)
ББК 32.971я7+32.972я7
С 138

Рецензенты: д-р техн. наук, проф. С. Б. П о п о в,
д-р техн. наук, проф. С. В. В о с т о к и н

Сагатов, Евгений Собирович

С 138 **Linux в суперкомпьютерных системах**: учеб. пособие / *Е.С. Сагатов, П.Ю. Якимов*. – Самара: Изд-во Самарского университета, 2018. – 116 с. : ил.

ISBN 978-5-7883-1261-3

Учебное пособие направлено на подготовку студентов к решению задач с использованием параллельных вычислений. Для этого им даются прикладные знания по работе с операционной системой Linux, а также особенностям суперкомпьютерных или кластерных систем. Студенты получают знания по оценке эффективности и производительности параллельных алгоритмов, а также по некоторым основам параллельного программирования.

Предназначено для студентов, обучающихся по направлениям подготовки 09.03.01 Информатика и вычислительная техника, 01.03.02 Прикладная математика и информатика, 01.04.02 Прикладная математика и информатика, 24.04.02 Системы управления движением и навигация, 12.04.04 Биотехнические системы и технологии.

Подготовлено на кафедре суперкомпьютеров и общей информатики.

УДК 004.382.2(075), 004.451(075)
ББК 32.971я7+32.972я7

ISBN 978-5-7883-1261-3

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

На сегодняшний день очень многие расчётные инженерные задачи требуют применения суперкомпьютерных технологий, но основной процесс обучения в университете предполагает использование программного обеспечения ANSYS, Microsoft Visual Studio и т.п. на персональных компьютерах с операционной системой Microsoft Windows.

Но в условиях реального производственного процесса будущие специалисты столкнутся с совсем другой средой. Реальные производственные задачи невозможно рассчитывать на персональном компьютере. Тоже касается научных исследований, когда накапливаются и обрабатываются большие данные. Всё это требует применения суперкомпьютеров, которые де-факто работают на операционной системе Linux.

В данном учебном пособии авторы постарались простым языком донести для будущих инженеров, которые уже научились решать некоторые производственные задачи, используя средства персонального компьютера, основы работы с операционной системой Linux, суперкомпьютерные технологии: кластеризацию, распараллеливание, написание многопоточных программ.

В качестве примера написания параллельных программ подробно рассмотрена технология программирования массивно-многопоточных графических процессоров.

В университете, где есть возможность работать с реальным суперкомпьютером «Сергей Королёв», каждый студент должен иметь соответствующую подготовку, чему посвящено данное учебное пособие.

1. РАБОТА С LINUX

1.1. Основные сведения

Операционная система, ОС (англ. operating system) - базовый комплекс компьютерных программ, обеспечивающий управление аппаратными средствами компьютера, интерфейс с пользователем, работу с файлами, ввод и вывод данных, а также выполнение прикладных программ и утилит.

ОС позволяет абстрагироваться от деталей реализации аппаратного обеспечения, предоставляя разработчикам программного обеспечения минимально необходимый набор функций. С точки зрения обычных пользователей компьютерной техники ОС включает в себя и программы пользовательского интерфейса.

Четыре составные части операционной системы: ядро, интерфейс пользователя, файловая система и прикладные программы и утилиты.

Ядро - центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память и внешнее аппаратное обеспечение. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов.

Интерфейс пользователя, (UI - англ. user interface) - разновидность интерфейсов, в котором одна сторона представлена человеком (пользователем), другая - машиной/устройством. Представляет собой совокупность средств и методов, при помощи которых пользователь взаимодействует с различными машинами и устройствами.

Чаще всего термин применяется по отношению к компьютерным программам (приложениям). Но вообще под пользовательским интерфейсом подразумевается любая система взаимодействия с устройствами, способными к интерактивному взаимодействию с пользова-

телем: меню на экране телевизора плюс пульт дистанционного управления им же, дисплей электронного аппарата (автомагнитола, часы, проигрыватель) и набор кнопок и переключателей для его настройки и управления, и так далее.

Файловая система (англ. file system) - регламент, определяющий способ организации, хранения и именования данных на носителях информации. Она определяет формат физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имени файла, максимальный возможный размер файла, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

С точки зрения операционной системы, весь диск представляет из себя набор кластеров размером от 512 байт и выше. Драйверы файловой системы организуют кластеры в файлы и каталоги (реально являющиеся файлами, содержащими список файлов в этом каталоге). Эти же драйверы отслеживают, какие из кластеров в настоящее время используются, какие свободны, какие помечены как неисправные.

Прикладная программа или приложение - программа, предназначенная для выполнения определенных пользовательских задач, рассчитана на непосредственное взаимодействие с пользователем. В большинстве операционных систем прикладные программы не могут обращаться к ресурсам компьютера напрямую, а взаимодействуют с оборудованием и проч. посредством операционной системы.

Утилита (англ. utility или tool) - компьютерная программа, расширяющая стандартные возможности оборудования и операционных систем, выполняющая узкий круг специфических задач.

Утилиты предоставляют доступ к возможностям (параметрам, настройкам, установкам), недоступным без их применения, либо делают процесс изменения некоторых параметров проще (автоматизируют его).

1.2. Начало работы с Linux

Как известно, работать в Linux можно в графической системе X Window или в текстовой консоли. Большинство пользователей после инсталляции предпочитают работать исключительно с оконным менеджером, но есть широкий ряд задач, выполнить которые можно (или значительно проще), работая в консоли.

Для начала немного о настройке консольного входа.

Добавление нового пользователя. Утилита ***adduser*** (начало задания выполняется преподавателем под пользователем с административными правами). Переход в режим суперпользователя осуществляется командой ***su.*** Студент должен придумать имя пользователя и пароль.

```
#adduser имя_пользователя
```

Далее за консолью работает студент и заполняет все поля самостоятельно. По окончанию необходимо выйти из консоли командой exit. Далее студент должен выполнить вход в систему самостоятельно, как в графическом, так и консольном интерфейсах.

Если при инсталляции Linux настроен автоматический запуск X-ов, то необходимо сначала перейти в консоль. Для этого нажмите Ctrl+Alt+F3. Вы попадете в виртуальную текстовую консоль и после ввода имени пользователя и пароля сможете давать команды shell. Для возвращения в X Window нажмите Alt+F7. Вообще говоря, по умолчанию можно работать сразу в 6-ти виртуальных консолях, что часто бывает очень удобно (переключение между ними - Alt+F1...Alt+F6).

Опытные пользователи советуют работать root-ом как можно меньше, поскольку его ошибка может вызвать самые фатальные для системы последствия, тогда как обычный пользователь может повредить обычно лишь свои собственные файлы.

Работать в консоли довольно удобно, но для перемещения по каталогам гораздо приятнее использовать Midnight Commander. После

вызова команды mc на экране появляется Norton-подобный файловый менеджер, который по мощности почти ничем не уступает DN или FAR.

1.3. Работа с командной строкой

Эффективная профессиональная работа в Linux немыслима без использования командной строки. Пользователям, привыкшим работать в системах с графическим интерфейсом, работа с командной строкой может показаться неудобной: то, что можно сделать одним перетаскиванием мышью в командной строке потребует ввода с клавиатуры нескольких слов: команды с аргументами. Однако в Linux этот вид интерфейса всегда был основным, а поэтому и хорошо развитым. В командных оболочках, используемых в Linux, есть масса способов экономии усилий (нажатий на клавиши) при выполнении наиболее распространённых действий: автоматическое дополнение длинных названий команд или имён файлов, поиск и повторное выполнение команды, уже когда-то исполнявшейся раньше, подстановка списков имён файлов по некоторому шаблону и многое другое. Преимущества командной строки становятся особенно очевидны, когда требуется выполнять однотипные операции над множеством объектов. В системе с графическим интерфейсом потребуется столько перетаскиваний мышью, сколько есть объектов, в командной строке будет достаточно одной (пусть длинной и сложной) команды.

В этом разделе будут описаны основные инструменты, позволяющие при помощи командной строки решать любые задачи пользователя: от тривиальных операций с файлами и каталогами (копирование, переименование, поиск) до сложных задач, требующих массовых однотипных операций, которые возникают как в прикладной работе пользователя, при работе с большими массивами данных или текста, так и в системном администрировании.

1.4. Командные оболочки (shells)

Общая информация об оболочках

Командная оболочка (или интерпретатор команд) - это программа, задача которой состоит в том, чтобы передавать ваши команды операционной системе и прикладным программам, а их ответы - вам. По своим задачам ему соответствует command.com в MS-DOS илиcmd.exe в Windows, но функционально оболочки в Linux несравненно богаче. На языке командной оболочки можно писать небольшие программы для выполнения ряда последовательных операций с файлами и содержащимися в них данными - сценарии (скрипты).

Зарегистрировавшись в системе (введя имя пользователя и пароль), вы увидите приглашение командной строки - строку, оканчивающуюся символом $ (далее этот символ будет использоваться для обозначения командной строки). В случае, если при установке был настроен запуск графического интерфейса при загрузке системы, то добраться до командной строки можно на любой виртуальной текстовой консоли (нажав Ctrl-Alt-F1 - Ctrl-Alt-F6) или при помощи любой программы эмуляции терминала, например, xterm. Обычно доступны следующие командные оболочки:

bash

Самая распространённая оболочка под Linux. Она умеет дополнять имена команд и файлов, ведёт историю команд и предоставляет возможность их редактирования.

pdkdh, sash, tcsh, zsh

Оболочкой по умолчанию является ***bash*** (Bourne Again Shell). Чтобы проверить, какую оболочку вы используете, наберите команду: echo $SHELL.

Оболочки отличаются друг от друга не только возможностями, но и синтаксисом команд. Для начинающих пользователей рекомендуется использовать ***bash***, дальнейшие примеры описывают работу именно в этой оболочке.

Командная оболочка bash

Командная строка в bash составляется из имени команды, за которым могут следовать ключи (опции) - указания, модифицирующие поведение команды. Ключи начинаются с символа - или --, и зачастую состоят из одной буквы. Кроме ключей, после команды могут следовать аргументы (параметры) - названия объектов, над которыми должна быть выполнена команда (часто - имена файлов и каталогов).

```
$ команда опции аргументы
```

Ввод команды завершается нажатием клавиши Enter, после чего команда передаётся оболочке на исполнение. В результате выполнения команды на терминале пользователя могут появиться сообщения о ходе выполнения команды или об ошибках, а появление очередного приглашения командной строки (оканчивающегося символом $) - знак того, что выполнение команды завершено и можно вводить следующую.

В bash имеется несколько приёмов, облегчающих ввод и редактирование командной строки. Например, используя клавиатуру, вы можете:

- *Ctrl-A* – перейти на начало строки, это же можно сделать, нажав клавишу Home;
- *Ctrl-U* – удалить текущую строку;
- *Ctrl-C* – прервать выполнение текущей команды.

Вы можете использовать символ ; для того, чтобы ввести несколько команд в одну строку. bash записывает историю всех выполненных команд, поэтому несложно повторить или отредактировать одну из предыдущих команд. Для этого достаточно выбрать нужную команду из истории: клавиша вверх выводит предыдущую команду, вниз - последующую. Для того, чтобы найти конкретную команду среди уже выполненных, не пролистывая всю историю, наберите *Ctrl-R* и введите какое-нибудь ключевое слово, употребленное в той команде, которую вы ищете.

Команды, присутствующие в истории, отображаются в списке пронумерованными. Для того, чтобы запустить конкретную команду, наберите:

```
!номер команды
```

Если ввести *!!*, запустится последняя из набранных команд.

Иногда в Linux имена программ и команд слишком длинны. К счастью, *bash* сам может завершать имена. Нажав клавишу *Tab*, вы можете завершить имя команды, программы или каталога. Например, предположим, что вы хотите использовать программу декомпрессии bunzip2. Для этого наберите:

```
$bu
```

Затем нажмите *Tab*. Если ничего не происходит - значит, существует несколько возможных вариантов завершения команды. Нажав клавишу *Tab* ещё раз, вы получите список имён, начинающихся с *bu*.

Например, в системе есть программы *buildhash, builtin, bunzip2*:

```
$ bu
buildhash builtin bunzip2
$ bu
```

Наберите *n* > (bunzip - это единственное имя, третьей буквой которого является n), а затем нажмите *Tab*. Оболочка дополнит имя и остаётся лишь нажать *Enter*, чтобы запустить команду!

Заметим, что программу, вызываемую из командной строки, *bash* ищет в каталогах, определяемых в системной переменной *PATH*. По умолчанию в этот перечень каталогов не входит текущий каталог, обозначаемый *./* (точка слэш). Поэтому для запуска программы prog из текущего каталога надо дать команду *./prog*.

1.5. Базовые команды

Первые задачи, которые приходится решать в любой системе: работа с данными (обычно хранящимися в файлах) и управление работающими в системе программами (процессами). Ниже перечислены команды, позволяющие выполнять наиболее важные операции по работе с файлами и процессами. Только первая из них - *cd* - является составляющей частью собственно командной оболочки, остальные

распространяются отдельно, но всегда доступны в любой системе Linux. Все команды, приведённые ниже, могут быть запущены как в текстовой консоли, так и в графическом режиме (xterm, консоль KDE). Для получения более подробной информации по каждой из команд используйте команду *man*, например:

```
$man ls
```

Команда *man* (от manual) запускает чтение инструкции в редакторе *vi*. Для того, чтобы выйти из редактора необходимо набрать в командной строке *:q*

```
$cd
```

Позволяет сменить текущий каталог (перемещаться по файловой системе). Она работает как с абсолютными, так и с относительными путями. Предположим, что вы находитесь в своём домашнем каталоге и хотите перейти в его подкаталог tmp/. Для этого, введите относительный путь:

```
$cd tmp/
```

Чтобы перейти в каталог */usr/bin*, наберите (абсолютный путь):

```
$cd /usr/bin/
```

Некоторые варианты использования команды:

```
$cd ..
```

позволяет вам сделать текущим родительский каталог (обратите внимание на пробел между cd и ..).

```
$cd -
```

позволяет вам вернуться в предыдущий каталог. Команда *cd* без параметров возвращает оболочку в домашний каталог.

```
$ls
```

ls (list) выдаёт список файлов в текущем каталоге. Две основные опции: *-a* - просмотр всех файлов, включая скрытые, *-l* - отображение более подробной информации.

less позволяет вам постранично просматривать текст. Синтаксис:

```
$less имя_файла
```

Бывает полезно просмотреть файл перед тем, как его редактировать; основное же применение данной команды - конечное звено цепочки программ, выводящей существенное количество текста, которое не умещается на одном экране и в противном случае слишком быстро промелькнёт. Для выхода из *less* нажмите *q* (quit).

```
$grep
```

Данная команда позволяет найти строку символов в файле. Обратите внимание, что *grep* осуществляет поиск по регулярному выражению, то есть предоставляет возможность задавать шаблон для поиска сразу целого класса слов. На языке регулярных выражений можно составлять шаблоны, описывающие, например, такие классы строк: «четыре цифры подряд, окружённые пробелами». Очевидно, такое выражение можно использовать для поиска в тексте всех годов, записанных цифрами. Возможности поиска по регулярному выражению очень широки, за более подробными сведениями вы можете обратиться к экранной документации по *grep* (man grep). Синтаксис:

```
$grep шаблон_поискафайл
$ps
```

Отображает список текущих процессов. Колонка команд указывает имя процесса, PID (идентификатор процесса) - номер процесса (используется для операций с процессом - например, отправки сигналов командой *kill*). Синтаксис:

```
$ps аргументы
```

Аргумент *u* предоставляет вам больше информации, *ax* позволяет вам просмотреть те процессы, которые не принадлежат вам.

```
$kill
```

Если программа перестала отвечать или зависла, используйте данную команду, чтобы её завершить. Синтаксис:

```
$kill PID_номер
```

PID_номер здесь - идентификационный номер процесса, вы можете узнать номер процесса для каждой выполняемой программы при помощи команды *ps*. Обычно команда *kill* отправляет процессу сигнал нормального завершения, однако иногда это не срабатывает и

необходимо будет использовать *kill -9 PID_number* - в этом случае команда будет немедленно завершена системой без возможности сохранения данных (аварийное завершение). Список сигналов, которые команда kill может отправлять процессу можно получить, отдав команду *kill -l*.

1.6. Текстовые редакторы

В Linux часто возникает необходимость в ручном редактировании конфигурационных файлов, *nano* это единственный редактор, доступный даже на стадии инсталляции системы. Да, это не *emacs*, и даже не *joe*. Но с задачей конфигурирования справляется успешно.

Nano - немодальный редактор, и для вставки текста можно сразу начинать набор. Если вы редактируете конфигурационный файл, такой как */etc/fstab*, указывайте параметр -w, например:

```
# nano -w /etc/fstab
```

Чтобы сохранить сделанные изменения, нажмите *Ctrl+O*. Для выхода из *nano* нажмите *Ctrl+X*. Если вы выходите из редактора, а файл изменен, *nano* предложит сохранить файл. Чтобы отказаться от сохранения, просто нажмите *N*, а для подтверждения - *Y*. Редактор запросит имя файла. Просто введите имя, а затем нажмите *Enter*.

Если вы случайно подтвердили необходимость сохранения файла, который сохранять не нужно, от сохранения всегда можно отказаться нажатием *Ctrl+C* в момент запроса имени файла.

При запуске терминала nano снизу терминала появляется подсказка, представляющая набор основных управляющих клавиш, доступных в сочитании с *Ctrl* +. Клавиша (в данном случае символ ^**)** заменяет клавишу *Ctrl*. Мета клавиша используется и в других сочетаниях, например **"*M*"** это означает клавишу ***Alt.***

1.7. Midnight Commander

Если вы многие годы работали в MS-DOS/Windows, то, наверное, ощущаете себя немного «не в своей тарелке». Для того, чтобы попасть в привычную среду, запустите Midnight Commander командой mc. Midnight Commander - это свободный аналог Norton Commander и

его популярного ныне потомка - Far. Если вы в какой-то момент сочтёте, что Midnight Commander что-то не умеет, то это, скорее всего, неверно. Ознакомьтесь с его описанием в */usr/share/doc/mc*-номер_версии или дайте команду *man mc*.

1.8. Графический интерфейс

Оконная система X и XFree86

Оконная система икс (от англ. X window system, далее - X) - один из самых больших и успешных проектов в истории компьютерной техники - восходит к 1984 г., когда разработчики двух систем компьютерной графики, претендующих на универсальность - проектов Athena (Массачусетский технологический институт) и W Windowing (Стэнфордский университет) - решили объединить свои усилия. С тех пор практически каждая компания, серьезно занимающаяся графикой, считала своим долгом внести какие-либо разработки в систему, формальным «хозяином» которой в 1987 г. стал вновь созданный X Consortium (ныне Open Group, www.X.org).

Дальнейшее изложение ориентировано на свободную реализацию X, которая называется XFree86, поддерживается одноименным партнерством (www.xfree86.org. XFree86 - самая популярная реализация X, она поставляется в составе подавляющего большинства открытых систем (как свободных, так и несвободных) для x86-совместимых компьютеров, поддерживает беспрецедентно широкий спектр оборудования и, благодаря доступности исходных текстов и пользовательской аудитории в десятки миллионов человек, весьма устойчива и хорошо оттестирована, по крайней мере, насколько это возможно для такого разнообразия поддерживаемого оборудования. Несмотря на то, что исторически цифры «86» в названии пакета относятся к соответствующему семейству процессоров от Intel, современные версии XFree86 реализованы для большинства других популярных процессоров. XFree86 доступен и для некоторых альтернативных архитектур ОС, включая **Microsoft**.

Большинство из того, о чем будет говориться в последующих разделах, справедливо для любой реализации X на любом оборудовании и под любой ОС, список которых можно найти на www.X.org.

Цветной бутерброд

То, что пользователю, сидящему за монитором, представляется сплошной графической операционной средой, реализовано как многослойный бутерброд технологий.

Непосредственно с оборудованием (видеосистемой, устройствами ввода и динамиком) работает *X-сервер*. Эта программа захватывает оборудование и предоставляет его возможности другим программам как ресурсы (собственно, именно поэтому она и называется сервером) по особому протоколу, который так и называется, X-протокол.

Ключевой компонент графической платформы - X-сервер:

- захватывает оборудование;
- создает по запросу других программ (которые в этой терминологии называются *X-клиентами*) окна;
- предоставляет другим программам возможность работы в окнах, т.е. вывода информации в эти окна и обработки сигналов от устройств ввода (клавиатуры и мыши или другого координатного устройства), когда окно, назначенное программе, является активным. Предоставление ресурсов возможно в том числе и через сеть, когда клиент и сервер работают на разных компьютерах (узлах).

При универсальном применении компьютера характерна поочередная работа с различными программами (иногда достаточно большим их количеством), причем пользователь может отрываться, допустим, от редактирования текста, чтобы поработать с иллюстрацией при помощи другой программы, прочитать почту или заглянуть на интернет-страницу, затем возвращаться к редактированию текста и т. д. Эти возможности обеспечивает другая программа - менеджер окон, представляющая собой следующий «слой» в графической среде пользователя.

Менеджеры окон

Для одновременной и поочерёдной работы с разными программами, требуется возможность *управлять окнами* (с помощью клавиатуры или мыши), т. е. возможность изменять «на лету» их геометрию (положение и размеры), а также (обычно не относимое к геометрии) положение в воображаемой «стопке» окон - от этого зависит, какое из окон будет «верхним» (видимым полностью), если окна перекрывают друг друга на плоскости экрана.

Управление окнами и составляет основную функцию *оконного менеджера* (устоявшийся англоязычный термин window manager).

Менеджеров окон существует превеликое множество - под любой набор задач, которые может решать графическая многооконная система. Их настолько много, что выбрать какой-нибудь в качестве «типичного представителя семейства» затруднительно.

Базовая (а также расширенная) функциональность оконных менеджеров доступна пользователю прежде всего за счет введения в интерфейс так называемых *виджетов* (от англ. widgets, сокращение от window gadgets, «оконные приспособления»). Виджеты - это рамки, кнопки, меню и пр., которые служат «органами управления» окна. Технически (в терминах оконной системы X) виджеты представляют собой отдельные окна, примыкающие к окну прикладной программы и, как правило, перемещающиеся вместе с ним.

Обрамление окна обычно составляют следующие элементы:

– Рамка, окружающая окно.

При «буксировке» рамки мышью окно изменяет свой размер. Иногда для изменения размера окна предназначены только выделенные «уголки» рамки, представляющие собой отдельные виджеты.

– Полоса заголовка.

Часто совпадает с одной из (обычно, верхней) сторон рамки. В полосе заголовка может содержаться название программы или запустившая программу команда, а также другая информация, специфичная для окна. При «буксировке» полосы заголовка перемещается все

окно. Со «щелчками» различными кнопками мыши на полосе заголовка также могут быть связаны различные действия по управлению окнами.

– Кнопки управления окном.

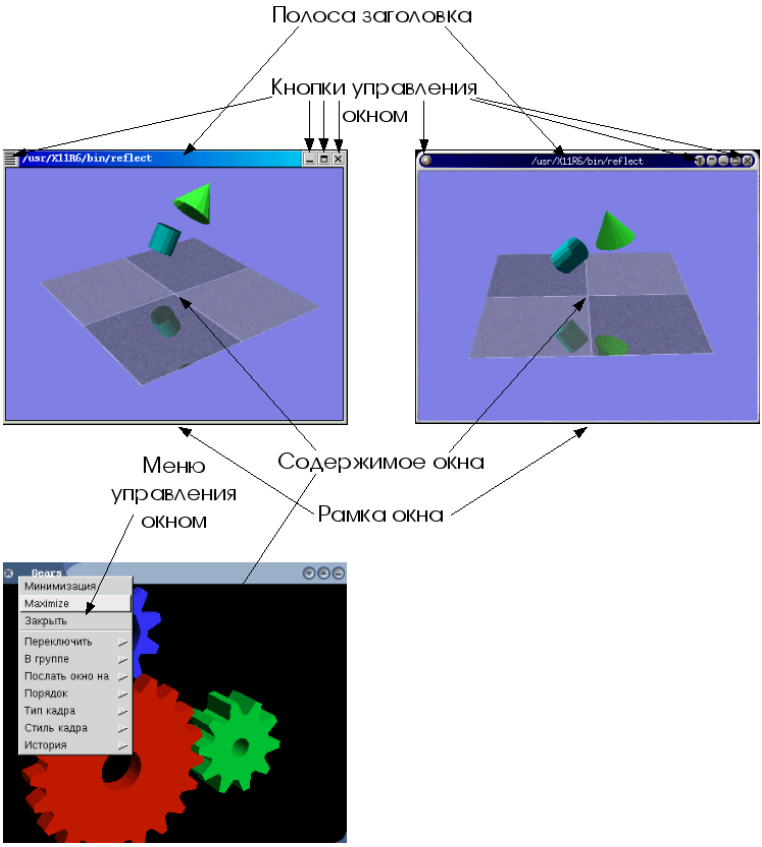


Рис. 1.1. Виджеты

Часто вынесенные на полосу заголовка или в другое место рамки кнопки позволяют выполнить с ним такие действия, как закрытие (часто сопровождающееся выходом из программы, открывшей окно), максимизация (разворачивание окна на весь экран), минимизация/сворачивание, вызов меню управления окном, которое может содержать весьма обширный репертуар других действий.

Детали реализации обрамления окна могут быть весьма различными в зависимости от конкретного оконного менеджера и его настроек.

Управление окнами - основная функция оконного менеджера, и на этом его функциональность может и заканчиваться. Однако большинство из них выполняют, по крайней мере, еще одну функцию.

При запуске оконного менеджера на экране появляется еще одно окно. Это так называемый *пейджер* (pager), на рис. 1.2 он изображен крупным планом. На пейджере представлена миниатюрная копия экрана, обновляющаяся в режиме реального времени, причем, если подвести курсор к изображению отдельного окна, оно увеличивается и рядом высвечивается название приложения, запущенного в нем. Но почему экран занимает только четверть окна пейджера? Потому что оконный менеджер позволяет оперировать «виртуальным» столом (от англ. virtual desktop, также *рабочим столом*), по размеру превышающим физический экран, а пейджер – одно из средств перемещения физического экрана по рабочему столу.

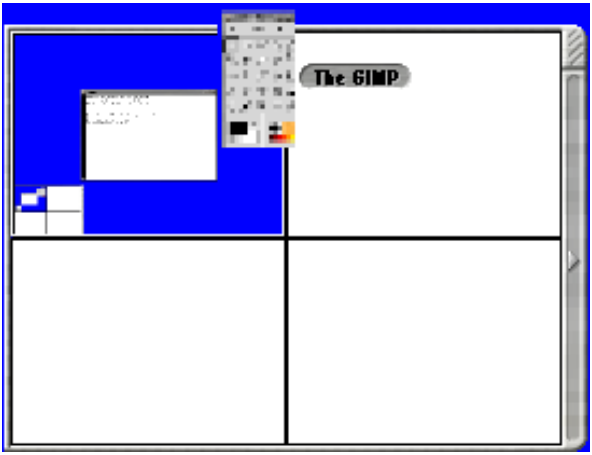


Рис. 1.2. Пейджер

1.9. Интегрированные графические среды

Существует два подхода к тому, как можно достроить оконную систему до полнофункциональной среды, позволяющей пользователю решать все (или почти все) его практические задачи. Во-первых, можно расширить функциональность менеджера окон, добавив в него недостающие возможности. В оконном менеджере до полнофункциональной среды не хватает возможности запускать программы и утилиты. Достигается это обычно при помощи организации специального меню. Во-вторых, можно добавить в «графический бутерброд» еще один слой - *менеджер рабочего стола* - работающий «поверх» менеджера окон и использующий функциональность последнего. Этим путем идут команды разработчиков ***GNOME*** и ***KDE***.

С точки зрения пользователя нет четкой границы между менеджерами окон с расширенной функциональностью и менеджерами рабочего стола, работающими «поверх» менеджера окон, поскольку они обеспечивают одну и ту же функциональность и нередко даже графически организованы сходным образом. Оба варианта предоставляют пользователю возможность работать в *графической среде* (desktop environment).

Интегрированная графическая среда предполагает не только единство оформления, но и трактовку объектов в рабочем пространстве (окон, файлов, пунктов меню и т. п.) как физических объектов, которые можно перемещать, выбрасывать в «корзину» и т.д.

На сегодня существуют и развиваются две свободные интегрированные графические среды общего назначения: KDE и GNOME. Они входят в поставку большинства стандартных (открытых) ОС, как свободных, так и несвободных.

GNOME

GNOME (GNOME, GNU Network Object Model Environment - «Среда ГНУ, основанная на модели сетевых объектов», но также и «Образцовая среда для сетевых объектов ГНУ») - один из самых амбициозных и масштабных проектов в программистском сообществе.

С пользовательской точки зрения ***GNOME*** предстает как набор базовых компонентов интерфейса и *аплетов*, утилит и прикладных программ. К базовым компонентам относятся менеджер файлов и поверхности стола *Наутилус (Nautilus),* панели управления и меню ***GNOME Panel*** и центр управления (*Gnome Control Center*).

Менеджер файлов ***Nautilus*** позволяет отображать содержимое файлов и каталогов в окнах и выполнять над файлами обычные действия (удаление, переименование, копирование и перемещение и т. п.), а также осуществлять предварительный просмотр многих типов данных. ***Nautilus*** эффектен, но работа с ним не более эффективна, чем с прочими браузерами файлов, включаемыми обычно в графические среды (менеджер файлов *CDE* или *Microsoft Windows Explorer*).

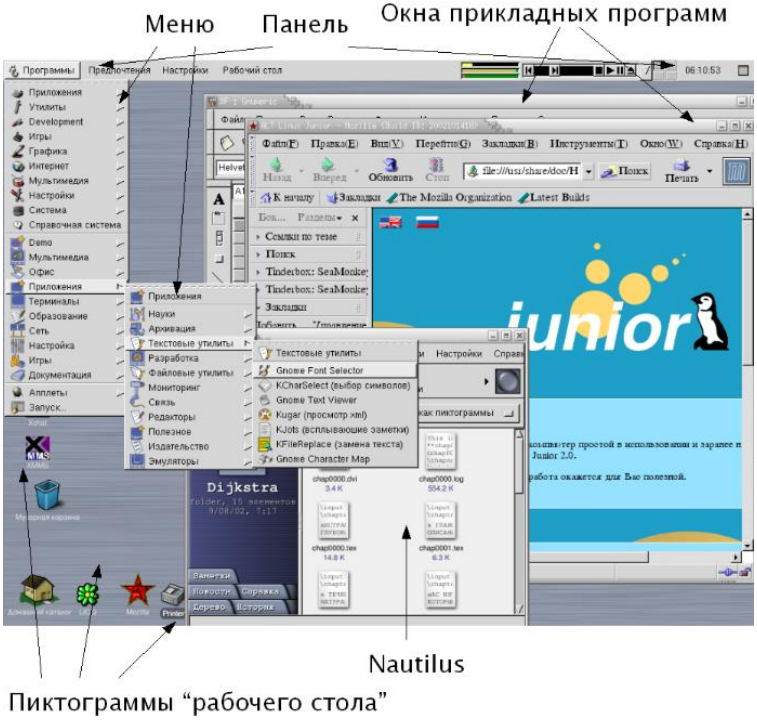


Рис. 1.3. Интегрированная графическая среда GNOME

Панели, наряду с менеджером файлов, являются важнейшей составной частью интерфейса ***GNOME***. Панелей может быть неограниченное количество. Панель может быть двух типов: панель-меню (*menu panel*) и объектная панель (*object panel*). Первая из них содержит пункты меню и может содержать пиктограммы, а вторая - только пиктограммы.

На панелях могут присутствовать объекты пяти типов:

- *Аплет* (applet, «приложеньице») - интересный тип панельного объекта, демонстрирующий то, что он не обязан быть представлен статической картинкой. Это программа, места в панели которой достаточно, чтобы отображать какую-нибудь полезную (или забавную) информацию или даже принимать клавиатурный и/или координатный ввод. Важными аплетами являются путеводитель по столу (*Desktop Guide*) и список задач (*Task List*), позволяющие переключаться между виртуальными экранами и активизировать окна запущенных программ, соответственно.
- *Пускатель* (launcher) ассоциирован с приложением или командой, которые исполняются по щелчку на его пиктограмме в панели.
- *Выдвижной ящик* (drawer) - это кнопка, открывающая другую панель, перпендикулярно первой - некий аналог подменю в меню, который можно наполнить всевозможными аплетами.
- *Специальные объекты* - это те же аплеты, но выполняющие функции, которые другими средствами «достать» почему-либо нельзя (запереть экран, выйти из ***GNOME*** или запустить программу «вручную»).
- Наконец, *объект-меню* раскрывает меню.

У ***GNOME*** нет единой иерархии меню: кроме главного, вызываемого объектом-меню с гномьей лапой (оно же, когда вызывается щелчком правой кнопки на фоне или нажатием клавиши, почему-то

называется *глобальным* (global)), пользователь может создавать *обычные* (normal) меню, связанные с объектами-меню на панелях.

Меню настраиваются примерно так же, как и панели: пользователь может добавлять, менять и удалять пункты, создавать подменю и т. п. При этом создаваемые обычные меню изначально пусты, а главное/глобальное заполняется при установке всем, что GNOME найдет в системе, и пользователю остается только убрать лишнее и переставить пункты в соответствии со своими предпочтениями.

Также постоянно расширяется набор утилит, прикладных программ и аплетов, поставляемых с ***GNOME*** - вместе с программами, входящими в большинство дистрибутивов ОС, о которых ***GNOME*** «в курсе», их число превышает сотню. Перечислить их здесь нет никакой возможности, но среди них есть интерфейсы для администрирования системы, средства звукозаписи и воспроизведения, сетевые утилиты, игры и многое другое.

KDE

Само название ***KDE*** (KDE, K Desktop Environment - «Графическая среда К») - явная пародия на CDE (Common Desktop Environment - «Общая настольная среда»). CDE была последней попыткой отрасли стандартизовать графическую среду на несвободной основе, предпринятой в конце девяностых годов. Буква «К» в KDE ничего не означает.

Несмотря на явно игривый тон, начинающийся с названия среды и продолжающийся в названии компонентов, ***KDE*** - очень серьезный проект. В ***KDE*** любят играть со словами; например, универсальный броузер, входящий в среду, называется *Konqueror* (от англ. conqueror- «завоеватель», «покоритель»), терминал - *Konsole* (от console - «консоль»), а система помощи - вообще *Kandalf* (от имени Гэндальфа, мага из произведений Дж. Р. Р. Толкиена).

Если единообразие и однородность графической среды считать достоинством, то ***KDE*** - несомненный лидер среди всех (как свободных, так и несвободных) интегрированных графических сред. Основное видимое средство интеграции - это универсальный броузер *Konqueror*. Функция *Konqueror* близка к той, которую приобрел Microsoft Windows Explorer в **Microsoft Windows** - он совмещает функции гипермедийного броузера WWW и браузера локальных ресурсов.

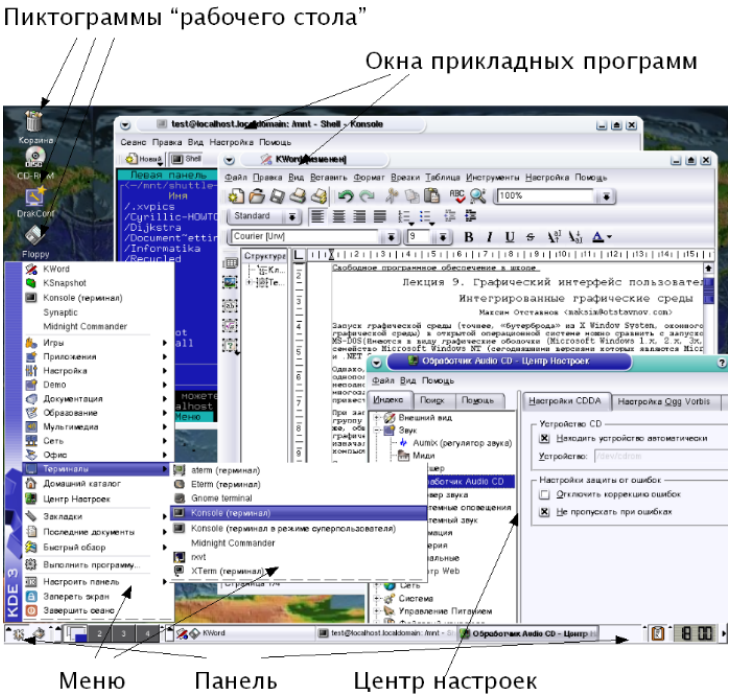


Рис. 1.4. Интегрированная графическая среда KDE

Разработчики *KDE* пошли даже дальше своих коллег из Microsoft и определили ряд дополнительных протоколов, что позволило, в частности, просматривать с помощью броузера в единообразном формате все разнообразие справочной информации, представленное в сегодняшних открытых система (традиционные страницы руководства

man, гипертекстовую систему ***Info*** из проекта ГНУ, разрозненные файлы документации в текстовом и гипертекстовом формате). В *Konqueror* интегрирована также возможность предварительного просмотра содержимого большого количества типов файлов.

KDE включает также настраиваемую систему панелей и меню и интегрированный *центр управления*, позволяющий согласованно изменять параметры среды. ***KDE*** менее гибка в настройке, чем ***GNOME***, однако ее гибкости вполне достаточно для решения любых практических задач (в том числе, имитации вида и поведения других сред). ***KDE*** работает только с собственным оконным менеджером *KWin*.

В поставку ***KDE*** входит множество «аксессуаров» и прикладных программ, к тому же рядом с проектом выросла целая группа сопутствующих, ориентированных на те или иные предметные приложения, из которых самым развитым является офисный пакет *KOffice*.

Файловая система - это структура, с помощью которой ядро операционной системы предоставляет пользователям (и процессам) ресурсы долговременной памяти системы, т. е. памяти на различного вида долговременных носителях информации - жестких дисках, магнитных лентах. CD-ROM и т. п. Для того, чтобы действительно понять *файловую систему* UNIX (а это также касается и файловых систем *Linux*), необходимо заново определить понятие «Что такое файл». С точки зрения UNIX все является файлом. Здесь «всё» действительно означает всё. Жесткий диск, раздел на жестком диске, параллельный порт, подключение к веб-сайту, карта Ethernet - всё это файлы. Даже каталоги являются файлами.

Linux различает много типов файлов в дополнение к стандартным файлам и каталогам. Обратите внимание, что здесь под типом файла мы подразумеваем не *содержимое* файла: в GNU/Linux, как и в любой другой системе UNIX, файл, будь то изображение PNG, двоичный файл или что-либо еще, - это просто поток байтов. Разделение файлов согласно их содержимому оставляется приложениям. Файл - один из базовых элементов любой операционной системы, и Linux

здесь не исключение. Но в этой ОС файлу придается особое значение, ведь им описывается любой объект - от текстового документа до устройства. А технологии разграничения прав доступа к файлам являются основой концепции безопасности Linux.

1.10. Файловая система Linux

Операционные системы хранят данные на диске при помощи ***файловых систем***. Классическая файловая система представляет данные в виде вложенных друг в друга ***каталогов*** (их ещё называют папками), в которых содержатся ***файлы***. Один из каталогов является «вершиной» файловой системы (а выражаясь технически - «корнем»), в нём содержатся (или, если угодно, из него растут) все остальные каталоги и файлы.

Имена файлов в Linux могут иметь длину до 255 символов и состоять из любых символов, кроме символа с кодом *0* и символа */* (слэша). Однако имеется еще ряд символов, которые имеют в оболочке shell специальное значение и которые поэтому не рекомендуется включать в имена. Это следующие символы:

`! @ # $ & ~ % * ( ) [ ] { } ' " \ : ; > < ` пробел.`

Если имя файла содержит один из этих символов (это не рекомендуется, но возможно), то вы должны перед этим символом поставить символ обратного слэша "\" (в том числе и перед самим этим слэшем, т. е. повторить его дважды). В Linux различаются символы верхнего и нижнего регистра в именах файлов. Поэтому FILENAME.tar.gz и filename.tar.gz вполне могут существовать одновременно и являться именами разных файлов.

Если жёсткий диск разбит на разделы, то на *каждом* разделе организуется отдельная файловая система с собственным корнем и структурой каталогов (ведь разделы полностью изолированы друг от друга).

В Linux корневой каталог называется весьма лаконично - “/”. Полные имена (пути) всех остальных каталогов получаются из “/”, к которому дописываются справа имена последовательно вложенных

друг в друга каталогов. Имена каталогов в пути также разделяются символом “/” («слэш»). Например, запись */home* обозначает каталог “home” в корневом каталоге (“/”), а */home/user* - каталог “user” в каталоге “home” (который, в свою очередь, в корневом каталоге)[3]. Перечисленные таким образом каталоги, завершающиеся именем файла, составляют **полный путь** к файлу.

Относительный путь строится точно так же, как и полный - перечислением через “/” всех названий каталогов, встретившихся при движении к искомому каталогу или файлу. Между полным путём и относительным есть только одно существенное различие: относительный путь начинается *от текущего каталога*, в то время как полный путь всегда начинается *от корневого каталога*. Относительный путь любого файла или каталога в файловой системе может иметь любую конфигурацию: чтобы добраться до искомого файла можно двигаться как по направлению к корневому каталогу, так и от него. Linux различает полный и относительный пути очень просто: если имя объекта *начинается* на “/” - это полный путь, в любом другом случае - относительный.

Стандартные каталоги

В корневом каталоге Linux-системы обычно находятся только подкаталоги со *стандартными* именами. Более того, не только имена, но и *тип данных*, которые могут попасть в тот или иной каталог, также регламентированы стандартом. Этот стандарт довольно последовательно соблюдается во всех Linux-системах: так, в любой Linux вы всегда найдёте каталоги */etc, /home, /usr/bin* и т. п. и сможете довольно точно предсказать, что именно в них находится.

Стандартное размещение файлов позволяет и человеку, и даже программе предсказать, где находится тот или иной компонент системы. Для человека это означает, что он сможет быстро сориентироваться в любой системе Linux (где файловая система организована в соответствии со стандартом) и найти то, что ему нужно. Для про-

грамм стандартное расположение файлов - это возможность организации автоматического взаимодействия между разными компонентами системы.

Операции с файлами

Для создания файлов проще всего обратиться к команде cat, используя перенаправление вывода:

```
$cat > [имя файла].
```

В этом случае в объект будет помещено всё, что вводится с клавиатуры (окончание операции - одновременное нажатие клавиш Ctrl и D). Разумеется, на практике данный метод используется редко - разве что при необходимости создать небольшой текстовый файл, состоящий из одной-двух строк.

Просмотреть только что созданный файл можно с помощью той же самой команды. Только при этом никакого перенаправления не будет, поскольку задействуется стандартный вывод: *cat [имя файла]*. Обратите внимание, как элегантно и экономно здесь работают консольные команды.

Впрочем, на практике чаще применяются другие программы: *more* и *less*. Синтаксис их довольно прост, в чем вы убедитесь, набрав в консоли команду

```
$man [название программы]
```

Для копирования, переименования или перемещения файлов вы можете использовать любой файловый менеджер. Однако тем, кто успел оценить достоинства командной строки, предлагаются другие решения.

Для копирования файлов в Linux существует команда *cp*. Набирать ее следует так:

```
$cp [параметры] [источник] [приемник]
```

В роли приемника выступает либо имя файла, либо название каталога, в котором объект будет продублирован с тем же наименованием.

Для сокращения набора как в этой, так и в других командах можно использовать специальные символы. Например, точка обозначает

ссылку на текущий каталог, тильда указывает на домашнюю директорию.

За перемещение или переименование файлов отвечает команда *mv*. Скажем, если надо перенести несколько объектов из одного каталога в другой, то следует набрать в консоли

```
$mv ~/*.[расширение] /[каталог назначения]
```

С этой командой надо обращаться особенно внимательно. Если в каталоге назначения имеется файл с точно таким же именем, то он будет изменен без предупреждения. С непривычки это кажется не совсем удобным, однако это всего-навсего иная концепция интерфейса: здесь программа является не столько партнером, сколько безмолвным слугой. Таким образом, от пользователя требуется более высокий уровень ответственности.

Удаление объектов системы осуществляется командой *rm*. Если набрать ее без параметров, то никакого предупреждения выдаваться не будет. Учитывая, что использование этой команды (особенно от имени суперпользователя) потенциально опасно, лучше ввести в консоли следующее:

```
$rm -i [файл ли группа файлов].
```

В этом случае у вас будет шанс передумать, поскольку система потребует подтверждения.

Для поиска файла предназначена команду *locate*. При этом вместо полного имени можно указывать его часть. Весьма полезная возможность для рассеянных людей, которые не могут удержать всего в памяти.

Если речь идет о команде (программе), то в командной строке нужно набрать *which [имя]*. Этот подход удобен не только для нахождения файла. В частности, таким методом можно быстро узнать, установлено или нет какое-либо приложение. Впрочем, той же цели можно достичь более коротким путем.

Поскольку система Linux поддерживает функцию автозаполнения командной строки, то пользователю достаточно ввести несколько

первых символов и нажать на клавишу *Tab*. Вам будет предложено несколько вариантов названия - вспомнить слово, напечатанное на дисплее, всегда проще, чем извлечь его из памяти (естественно, не из оперативной, а из своей собственной).

Поиск

Поиск информации в файловой системе можно условно разделить на поиск по атрибутам файла (понимая их расширительно, то есть включая имя, путь и т. п.) и поиск по содержимому.

Существуют не менее пяти способов поиска файлов в Linux. Здесь мы изложим лишь некоторые из них.

Начнем с простой команды – *find*, которая может выглядеть как:

```
$find / -name filename.txt
```

Эта команда задает поиск по всем каталогам от корневого /

Другие примеры использования команды *find*:

```
$ find /home -user serhiy
```

Найти все файлы в директории */home* и всех поддиректориях принадлежащие пользователю *serhiy*

```
$find ~ -name *.c
```

В вашей домашней директории найдет все файлы с расширением .c. Например *helloworld.c*

```
$ find . -name "[A-Z]*"
```

В текущем каталоге и его подкаталогах найдет файлы, начинающиеся с большой буквы. Заметьте что выражение для поиска задано в "..."

```
$ find /var/www/ -mtime -10
```

Найти файлы в каталоге */var/www/* и его подкаталогах, которые были изменены менее чем 10 дней назад

```
$ find /var/www/ -mtime +30 -name "*.php"
```

Найти все .php файлы в каталоге /var/www/ и его подкаталогах, которые были изменены более чем 30 дней назад

```
$ find . -perm 777
```

Найти все файлы в текущем каталоге, которые имеют права доступа 777.

Команда *locate*

Команда *locate* это альтернатива команде *find -name*. Команда *find* ищет файлы в выбранной части файловой системы и процесс может быть не очень быстрым. С другой стороны, команда *locate* ищет файлы в базе данных, созданной специально для этих целей */var/lib/locatedb*, что происходит намного быстрее. Для обновления базы используется команда ***updatedb***.

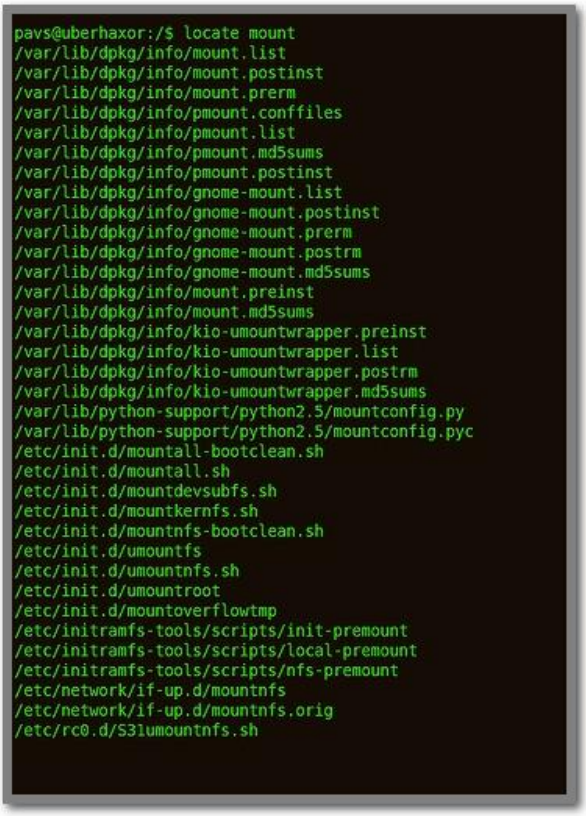


Рис. 1.5. Пример использования команды ***locate***

Команда ***whereis***

whereis возвращает место расположения кода (опция -b), манстраниц (опция -m), и исходные файлы (опция -s) для указанной команды. Если опции не указываются, выводится вся доступная информация. Эта команда быстрее чем *find,* но менее полная.

Рис. 1.6. Пример использования команды ***whereis***

Команда *which*

Команда *which* ищет все пути, перечисленные в переменной PATH для указанной команды.

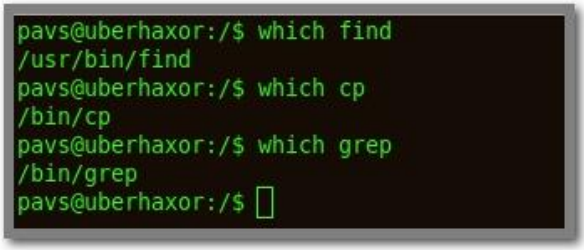


Рис. 1.7. Пример использования команды ***which***

Команда ***type***

При вызове без опций показывает, как имена будут интерпретироваться при использовании в качестве имени команды. Если использована опция ***-a***, команда ***type*** выдает список всех каталогов, где есть выполняемый файл с соответствующим именем. В список включаются также псевдонимы и функции, если только не указана опция ***-p***. К хэшу команд не обращаются, если указана опция ***-a***. Команда ***type*** возвращает 0, если хоть один из аргументов найден, и 1 в противном случае.

Рис. 1.8. Пример использования команды ***type***

Монтирование

Корневой каталог в Linux всегда только *один*, а все остальные каталоги в него вложены, т. е. для пользователя файловая система представляет собой единое целое. В действительности, разные части файловой системы могут находиться на совершенно разных устройствах: разных разделах жёсткого диска, на разнообразных съёмных носителях (лазерных дисках, дискетах, флэш-картах), даже на других компьютерах (с доступом через сеть). Для того, чтобы соорудить из этого хозяйства единое дерево с одним корнем, используется процедура **монтирования**.

Монтирование - это подключение в один из каталогов целой файловой системы, находящейся где-то на другом устройстве. Эту операцию можно представить как «прививание» ветки к дереву. Для монтирования необходим пустой каталог - он называется **точкой монтирования**. Точкой монтирования может служить любой каталог, никаких ограничений на этот счёт в Linux нет. При помощи специальной команды (*mount*) мы объявляем, что в данном *каталоге* (пока пустом) нужно отображать файловую систему, доступную на таком-то *устройстве* или же по сети. После этой операции в каталоге (точке монтирования) появятся все те файлы и каталоги, которые находятся на соответствующем устройстве. В результате пользователь может даже и не знать, на каком устройстве какие файлы располагаются.

Подключённую таким образом («смонтированную») файловую систему можно в любой момент отключить - **размонтировать** (для

этого имеется специальная команда *umount*), после чего тот каталог, куда она была смонтирована, снова окажется пустым.

Для Linux самой важной является **корневая файловая система** (root filesystem). Именно к ней затем будут подключаться (монтироваться) все остальные файловые системы на других устройствах. Обратите внимание, что корневая файловая система тоже монтируется, но только не к другой файловой системе, а к «самой Linux», причём точкой монтирования служит "/" (корневой каталог). Поэтому при загрузке системы прежде всего монтируется корневая файловая система, а при останове она размонтируется (в последнюю очередь).

Пользователю обычно не требуется выполнять монтирование и размонтирование вручную: при загрузке системы будут смонтированы все устройства, на которых хранятся части файловой системы, а при останове (перед выключением) системы все они будут размонтированы. Файловые системы на съёмных носителях (лазерных дисках, дискетах и пр.) также монтируются и размонтируются автоматически - либо при подключении носителя, либо при обращении к соответствующему каталогу.

1.11. Права доступа в системе Linux

Пользователи и группы

Поскольку система **Linux** с самого начала разрабатывалась как многопользовательская система, в ней предусмотрен такой механизм, как права доступа к файлам и каталогам. Он позволяет разграничить полномочия пользователей, работающих в системе. В частности, права доступа позволяют отдельным пользователям иметь «личные» файлы и каталоги. Например, если пользователь **user** создал в своём домашнем каталоге файлы, то он является владельцем этих файлов и может определить права доступа к ним для себя и остальных пользователей. Он может, например, полностью закрыть доступ к своим файлам для остальных пользователей, или разрешить им читать свои файлы, запретив изменять и исполнять их.

Правильная настройка прав доступа позволяет повысить надёжность системы, защитив от изменения или удаления важные системные файлы. Наконец, поскольку внешние устройства с точки зрения **Linux** также являются объектами файловой системы, механизм прав доступа можно применять и для управления доступом к устройствам.

У любого файла в системе есть *владелец* - один из пользователей. Однако каждый файл одновременно принадлежит и некоторой *группе* пользователей системы. Каждый пользователь может входить в любое количество групп, и в каждую группу может входить любое количество пользователей из числа определённых в системе.

Кроме учётных записей, которые используют люди для работы с системой, в **Linux** предусмотрены учётные записи для т. н. *системных пользователей*: с точки зрения системы это такие же пользователи, которые могут быть владельцами файлов, однако эти учётные записи используются только для работы некоторых программ-серверов. Например, стандартный системный пользователь **mail** используется программами доставки почты.

Когда в системе создаётся новый пользователь, он добавляется по крайней мере в одну группу. В дальнейшем администратор может добавить пользователя к другим группам.

Механизм групп может применяться для организации совместного доступа нескольких пользователей к определённым ресурсам. Например, на сервере организации для каждого проекта может быть создана отдельная группа, в которую войдут учётные записи (имена пользователей) сотрудников, работающих над этим проектом. При этом файлы, относящиеся к проекту, могут принадлежать этой группе и быть доступными для её членов.

Виды прав доступа

Права доступа определяются по отношению к трём типам действий: чтение **(r)**, запись **(w)** и исполнение **(x).** Эти права доступа могут быть предоставлены трём классам пользователей: владельцу файла (пользователю), группе, которой принадлежит файл, а также

всем остальным пользователям, не входящим в эту группу. Право на чтение даёт пользователю возможность читать содержимое файла или, если такой доступ разрешён к каталогам, просматривать содержимое каталога (используя команду *ls*). Право на запись даёт пользователю возможность записывать или изменять файл, а право на запись для каталога - возможность создавать новые файлы или удалять файлы из этого каталога. Наконец, право на исполнение позволяет пользователю запускать файл как программу или сценарий командной оболочки (разумеется, это действие имеет смысл лишь в том случае, если файл является программой или сценарием). Владение правами на исполнение для каталога позволяет перейти (командой *cd*) в этот каталог.

Чтобы получить информацию о правах доступа, используйте команду ***ls*** с ключом ***-l***. При этом будет выведена подробная информация о файлах и каталогах, в которой будут, среди прочего, отражены права доступа. Рассмотрим следующий пример:

```
$ls –l > ls.txt
$ls -l
-rw-r-r--   user user 430 Ноя 9 11:11 ls.txt
```

Первое поле в этой строке *(-rw-r--r--)* отражает права доступа к файлу. Третье поле указывает на владельца файла, четвёртое поле указывает на группу, которая владеет этим файлом. Последнее поле - это имя файла (**ls.txt**). Другие поля описаны в документации к команде *ls.*

Первый символ из этого ряда (**-**) обозначает тип файла. Символ - означает, что это - обычный файл, который не является каталогом (в этом случае первым символом было бы **d**) или псевдофайлом устройства. Следующие три символа (**rw-**) представляют собой права доступа, предоставленные владельцу **user**. Символ **r** - сокращение от read (англ. читать), а **w** - сокращение от write (англ. писать). Таким образом, **user** имеет право на чтение и запись (изменение) файла.

После символа **w** мог бы стоять символ **x**, означающий наличие прав на исполнение (англ. execute, исполнять) файла. Однако символ

-, стоящий здесь вместо **x**, указывает, что **user** не имеет права на исполнение этого файла. Это разумно, так как файл не является программой. В то же время, пользователь, зарегистрировавшийся в системе как **user**, при желании может предоставить себе право на исполнение данного файла, поскольку является его владельцем. Для изменения прав доступа к файлу или каталогу используется команда **chmod**.

Следующие три символа (**r--**) отражают права доступа группы к файлу. Наконец, последние три символа (это опять **r--**) отражают собой права доступа к этому файлу всех других пользователей, помимо собственника файла и пользователей из группы. Так как здесь указан только символ **r**, эти пользователи тоже могут читать файл.

Вот ещё несколько примеров:

```
-rwxr-x--x
```

Пользователь-владелец файла может читать файл, изменять и исполнять его; пользователи, члены группы-владельца могут читать и исполнять файл, но не изменять его; все остальные пользователи могут лишь запускать файл на выполнение.

```
-rw-------
```

Только владелец файла может читать и изменять его.

```
-rwxrwxrwx
```

Все пользователи могут читать файл, изменять его и запускать на выполнение.

```
----------
```

Никто, включая самого владельца файла, не имеет прав на его чтение, запись или выполнение. Хотя такая ситуация вряд ли имеет практический смысл, с точки зрения системы она является вполне корректной. Разумеется, владелец файла может в любой момент изменить права доступа к нему.

Возможность доступа к файлу зависит также от прав доступа к каталогу, в котором находится файл. Например, даже если права доступа к файлу установлены как **-rwxrwxrwx**, другие пользователи не

могут получить доступ к файлу, пока они не имеют прав на исполнение для каталога, в котором находится файл. Другими словами, чтобы воспользоваться имеющимися у вас правами доступа к файлу, вы должны иметь право на исполнение для всех каталогов вдоль пути к файлу.

«Пользователями» системы **Linux**, выполняющими различные действия с файлами и каталогами, являются на самом деле вовсе не люди, а программы, выполняемые в системе - *процессы*. Одна из таких программ - командная оболочка, которая считывает команды пользователя из командной строки и передаёт их системе на выполнение. Каждая программа (процесс) выполняется от имени определённого пользователя. Её возможности работы с файлами и каталогами определяются правами доступа, определёнными для этого пользователя.

С целью оптимальной настройки прав доступа для ряда программ-серверов в системе созданы системные пользователи (учётные записи), от имени которых работают эти программы. Например, веб-сервер (Apache) выполняется от имени пользователя **apache**, а ftp-сервер - от имени пользователя **ftp**. Такие учётные записи не предназначены для работы людей-пользователей.

Права доступа и администрирование системы

Установка и поддержание оптимальных прав доступа является одной из важнейших задач системного администратора. Права должны быть достаточными для нормальной работы пользователей и программ, но не большими, чем необходимо для такой работы.

Поскольку программы, исполняемые от имени суперпользователя (**root**), могут совершать любые действия с любыми файлами и каталогами, их выполнение может нанести системе серьёзный ущерб. Это может быть как следствием уязвимостей или ошибок в программах, так и результатом ошибочных действий самого пользователя. Поэтому работа с правами суперпользователя требует особой осторожности. Чтобы уменьшить связанные с этим риски рекомендуется для

выполнения задач, требующих таких прав, использовать утилиту ***sudo***.

Основные команды.

Ниже перечислены важнейшие команды для решения задач, связанных с правами доступа. Обратитесь к документации по системе для получения информации об использовании этих команд.

Изменение прав доступа к файлу или каталогу:

```
$chmod
```

Изменение владельца файла:

```
$chown
```

Изменение группы, которой принадлежит файл:

```
$chgroup
```

Определение прав доступа по умолчанию для файлов, создаваемых пользователем:

```
$umask
```

1.12. Установка приложений под Linux

Для Linux приложения поставляются в основном в виде *rpm*-пакетов или **.tar.gz*-архивов. Устанавливается *rpm*-пакет командой

```
$rpm -i имя пакета
```

Программа *rpm* сама создаст все необходимые для работы приложения каталоги и положит туда файлы. Если у вас уже установлена предыдущая версия приложения, то в командной строке надо дать ключ *–force* для замены старой версии. *rpm*-пакеты содержатся на CD с большинством дистрибутивов Linux, а также на многих ftp-серверах в Интернете. Если расширение *rpm*-файла выглядит как **.src.rpm*, то это - исходный код приложения, и перед запуском его необходимо самостоятельно откомпилировать (обычно такие пакеты содержат инструкцию о том, как это сделать). Для удаления пакета из системы дайте команду

```
$rpm -e имя пакета.
```

Если вам досталось приложение в виде упакованного файла с расширением ***.tar.gz**, то для его распаковки надо дать команду

```
$tar xzvf имя архив.
```

Далее необходимо найти файл с инструкциями по установке приложения, каковые в каждом конкретном случае могут различаться.

Кстати, чтобы просмотреть содержимое архивов, не распаковывая их вручную, удобно пользоваться Midnight Commander. При нажатии *Enter* на имени архива вы входите в него, как в обычный каталог.

Хотелось отметить одну полезную программу - *fsck*. Если, к примеру, во время работы в Linux у вас отключилось питание или вы случайно нажали *reset* :-), то при загрузке ОС спросит пароль root, и вы попадете в однопользовательский режим. В нем файловая система смонтирована только для чтения и исполнения. Для того, чтобы восстановить поврежденную файловую систему, дайте команду

```
$fsck -Aa
```

После ее окончания дайте команду *reboot*, и после перезагрузки все должно заработать, как раньше.

Для нас также представляет интерес приложение *kpackage - Менеджер пакетов*. Он позволяет устанавливать и удалять приложения Linux, не прибегая к консольной утилите *rpm*, описанной выше. В левой части экрана находится список установленных пакетов, разделенный по категориям. При выборе одного из них в правой части экрана показывается его описание.

Менеджер пакетов Synaptic

Synaptic - это графическая программа, позволяющая управлять пакетами. Она совмещает в себе все возможности консольной системы управления пакетами *apt* и удобство графического интерфейса. С помощью *Synaptic* можно устанавливать, удалять, настраивать и обновлять пакеты в системе, просматривать списки доступных и установленных пакетов, управлять репозиториями и обновлять систему до новой версии.

Перед запуском программы вы увидите окно, в которое вам нужно будет ввести свой пароль, для дальнейшей работы с приложением.

Для запуска Synaptic откройте:

Система→Администрирование→Менеджер пакетов Synaptic или введите в терминале:

```
$sudo synaptec
```

При запуске вы увидите главное окно программы:

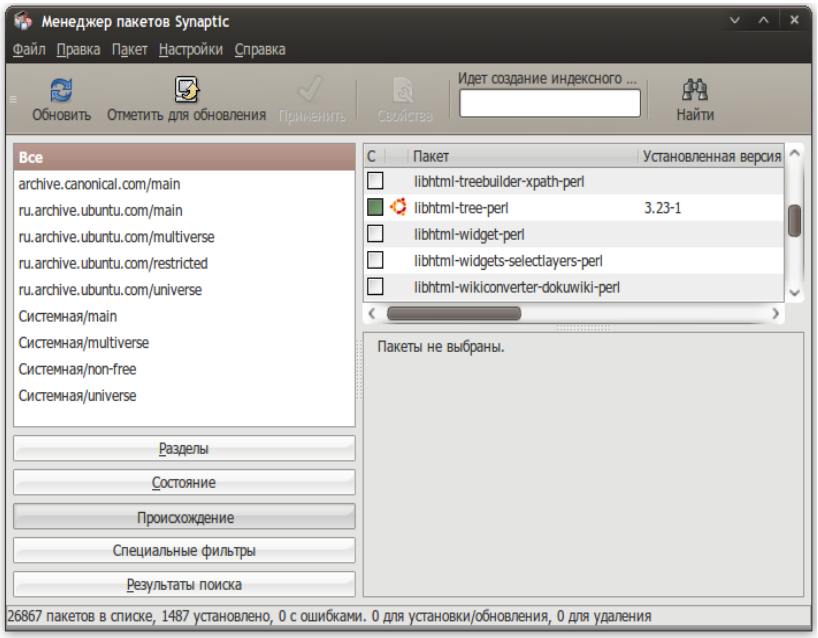


Рис. 1.9. Менеджер пакетов Synaptic

Главное меню вверху, панель с несколькими кнопками, роль которых станет ясна чуть позже.

В левой части экрана внизу есть пять кнопок, которые определяют, что будет показываться в списке над ними, так что вы можете выбирать пакеты в списке, группируя их по статусу.

Если вы выберет «Все», вы увидите полный список доступных и установленных пакетов. При нажатии «Установленные» будут показываться лишь установленные пакеты и так далее. Правая часть окна разделена на верхнюю и нижнюю части. В верхней части выводится список пакетов, и при выбора пакета из этого списка в нижней части отображаются сведения о нем и его описание.

Пакеты могут быть сгруппированы по функциональности (текстовые редакторы, документация, почтовые клиенты и т.д.). Для этого

используется кнопка «Разделы». После нажатия на нее вы сможете выбирать пакеты из различных секций.

Для получения подробной информации о пакете, кликните по нему правой кнопкой мыши и в появившемся меню выберите «Свойства».

Установка и удаление ПО

Установка.

- Щелкните по кнопке «Обновить» или нажмите *Ctrl + r* для того чтобы скачать список самых последних версий ПО.
- Правый клик на нужном пакете и выберите в появившемся меню «Отметить для установки», или нажмите *Ctrl + I*. Если пакет требует установки другого пакета, то появиться диалоговое окно с изменениями которые будут сделаны, если вы действительно хотите продолжить установку, то щелкните по клавише «Применить» или нажмите *Ctrl + P*.
- Для установки, нажмите кнопку «*Применить*» на главной панели Менеджера пакетов *Synaptic*.

Удаление.

- Правый клик на нужном пакете и выберите в появившемся меню выберите «Отметить для удаления».
- Появиться диалоговое окно с изменениями которые будут сделаны, если вы действительно хотите продолжить удаление, то щелкните по клавише «Применить» или нажмите *Ctrl + P*.
- Для удаления, нажмите кнопку «*Применить*» на главной панели Менеджера пакетов *Synaptic*.

Если вы отметите пакет маркером «Отметить для полного удаления» то удалится не только выбранный вами пакет, но и все зависимости.

Как исправить сломанные пакеты.

«Сломанные пакеты» - это пакеты которые имеют неудовлетворённые зависимости. Если сломанные пакеты обнаружены, то

Synaptic не позволит проводить ни каких изменений в системе с пакетами до тех пор пока все сломанные пакеты не будут исправлены.

Для исправления сломанных пакетов.

1. Выберите Правка→Исправить пакеты с ошибками в главном меню.
2. Выберите «Внести отмеченные изменения» в меню «Правка» или нажмите *Ctrl + P*
3. Подтвердите изменения, щелкнув по кнопке «*Применить*».

Таблица 1.1 Горячие клавиши в Synaptic

Команда	Сочетание клавиш
Обновить список доступных пакетов	Ctrl + R
Открыть диалоговое окно поиска	Ctrl + F
Показать окно с свойствами выбранного пакета	Ctrl + O
Отметить выбранный(е) пакет(ы) для установки	Ctrl + I
Отметить выбранный(е) пакет(ы) для обновления	Ctrl + U
Отметить выбранный(е) пакет(ы) для удаления	Delete
Отметить выбранный(е) пакет(ы) для полного удаления	Shift + Delete
Снять какие-либо изменения в пакетах	Ctrl + N
Отметить все возможные обновления	Ctrl + G
Быстрая установка специфической версии для пакета	Ctrl + E
Отменить последнее изменение	Ctrl + Z
Повторить последнее изменение	Ctrl + Shift + Z
Применить все выбранные действия	Ctrl + P
Выйти из Synaptic	Ctrl + Q

1.13. Лабораторные работы

Лабораторная работа 1 – Команды для работы с файлами

1) Создать директорию lab1 в домашнем каталоге.

2) Создать файл first.txt с текстом “Мама мыла”.

3) Создать файл second.txt с текстом “раму.”

4) Объединить содержимое файлов first и second в файле third.txt.

5) Удалить файлы first.txt и second.txt.

6) Переименовать файл third.txt в first.txt.

7) Скопировать файл first.txt в second.txt.

8) Вывести на экран содержимое получившихся файлов.

Часто используемые команды:

ls, cd, cp, mv, rm, rmdir, mkdir, touch, echo, cat, chown, chmod

Лабораторная работа 2 – Поиск файлов, работа со ссылками

1) Найти на диске все файлы типа сокет и вывести постранично в консоль

2) Создайте каталог с любым именем в домашней директории и далее выполняйте в нём

3) Создайте файл с текстовыми данными, путем перенаправления результата команды cat в файл (cat > myfile).

4) Далее скопируйте файл, создайте на него жесткую и символьную ссылки. Все объекты оставьте в том же каталоге, что и файл-оригинал.

5) Выполните команду ls -l, затем сделайте выводы о том, какие имена указывают на один и тот же объект, а какие на разные.

6) Создайте жёсткую ссылку на myfile в директории /tmp.

7) Поиском по всему диску найдите файлы, указывающие на те же данные, что и /tmp/myfile.

8) Повторите тоже самое с символической ссылкой.

Лабораторная работа 3 – Ввод/вывод, выборка, скрипты

Написать скрипт, выводящий информацию о компьютере в следующем виде:

```
Имя компьютера: Stud1
IP адреса: 192.168.0.10, 127.0.0.1
Процессор: Pentium 4 (2.2 MHz), 4 ядра
Объём ОЗУ: 4Гб (20% свободно)
Объём ЖД: 80Гб (60% свободно)
```

Лабораторная работа 4 – Настройка сети

1) Сохранить IP-адрес, маску, шлюз, DNS-сервера из подключения в отдельном файле.

2) Удалить настройки из NetworkManager и перезапустить сеть. Убедиться, что IP-адрес не присвоен.

3) Задать все настройки сетевому интерфейсу командами ip, route и в файле resolv.conf. Проверить работоспособность сети.

4) Очистить настройки интерфейса.

5) Задать настройки интерфейса в файле interfaces. Проверить работу сети.

6) Удалить сделанные изменения и вернуть первоначальные настройки сети на компьютере.

Лабораторная работа 5 – Установка/удаление пакетов, вывод информации о пакетах

1) Установить пакет mc с помощью командной строки. Какие дополнительные пакеты требуются для удовлетворения зависимостей?

2) Установить пакеты apache2 и php5 с помощью aptitude. От каких пакетов зависит apache2?

3) Вывести список пакетов, установленных не автоматически через запятую.

Лабораторная работа 6 – Управление процессами

1) Создать скрипт, который будет запускать команду sleep 3600 в фоновом режиме и сохранять информацию о процессе в /run в общепринятом формате. Опция запуска start.

2) Скрипт не должен повторно запускать команду, если процесс уже запущен. Проверить действительно ли процесс запущен.

3) Добавить в созданный скрипт возможность завершения фонового процесса опцией stop. Корректно обработать данную команду для случая, когда фоновый процесс не был запущен.

4) Если скрипт запускается без опций или с неверными аргументами, то выводить подсказку по командам.

5) Добавить опцию перезапуска restart, которая будет делать при необходимости stop и затем start.

Лабораторная работа 7 – Загрузка ОС, службы, автозапуск

1) Добавить при загрузке Linux команду синхронизации времени по протоколу NTP в фоновом режиме.

2) Доработать лабораторную работу №6 таким образом, чтобы создать демон ОС Linux.

3) Создать скрипт запуска и остановки процесса sleep 360 000 и соответствующую описательную часть таким образом, чтобы демон запускался при старте ОС после запуска службы сети.

Лабораторная работа 8 – Основы конфигурирования: установка веб-сервера

1) Установить apache2, PHP с модулями и MySQL. Проверить, что apache отвечает на localhost.

2) Внести коррективы в настройки apache2 и файла hosts таким образом, чтобы apache начал отвечать на запросы браузера с именем вашего домена вида familia.ru.

3) Создать БД и пользователя в MySQL для установки Joomla.

4) Установить Joomla на своём домене. Проверить работоспособность.

5) Русифицировать Joomla. Активировать в Joomla человекопонятные ссылки. Необходимо включить модуль rewrite в apache.

Список литературы

1. Баркакати, Н. Red Hat Linux. Секреты профессионала / Наба Баркакати. – Вильямс, 2004. – 1056 с.
2. Бендел, Д. Использование Linux / Дэвид Бендел, Роберт Нейпи. – 6-е изд., спец. изд. – Вильямс, 2002. – 784 с.
3. Блам, Р. Система электронной почты на основе Linux. Руководство администратора / Ричард Блам. – Вильямс, 2001. – 448 с.
4. Боковой, А. ALT Linux снаружи. ALT Linux изнутри / [А. Боковой и др.]. – ДМК пресс, 2006.
5. Визерспун, К. Освой самостоятельно Linux за 24 часа / Крэг Визерспун, Колетта Визерспун. – 3-е изд. – Вильямс, 2001. – 352 с.
6. Джонсон, М. К. Разработка приложений в среде Linux. Программирование для linux / Майкл К. Джонсон, Эрик В. Троан. – 2-е изд. – Вильямс, 2007. – 544 с.
7. Камер, Д. Сети TCP/IP. Т. 3. Разработка приложений типа клиент: сервер для Linux / POSIX / Дуглас Камер, Дэвид Л. Стивенс. – Вильямс, 2002. – 592 с.
8. Карлинг, М. Системное администрирование Linux / М. Карлинг, Стефан Деглер, Джеймс Деннис. – Вильямс, 2000. – 320 с.
9. Лав, Р. Разработка ядра Linux / Роберт Лав. – 2-е изд. – Вильямс, 2008. – 448 с.
10. Лебланк, Д.-А. Linux для чайников / Ди-Анн Лебланк. – 6-е изд. – Диалектика, 2008. – 464 с.
11. Ли, Д. Использование Linux, Apache, MySQL и PHP для разработки Web-приложений / Джеймс Ли, Брент Уэр. – Вильямс, 2004. – 432 с.

12. Мак-Клар, С. Секреты хакеров. Безопасность сетей – готовые решения / Стюарт Мак-Клар, Джоэл Скембрей, Джордж Курц. – 4-е изд. – Вильямс, 2004. – 656 с.

13. Манн, С. Безопасность Linux / Скотт Манн, Эллен Л. Митчелл, Митчелл Крелл. – 2-е изд. – Вильямс, 2003. – 624 с.

14. Митчелл, М. Программирование для Linux. Профессиональный подход / Марк Митчелл, Джеффри Оулдем, Алекс Самьюэл. – Вильямс, 2004. – 288 с.

15. Негус, К. Red Hat Linux 8. Библия пользователя / Кристофер Негус. – Диалектика, 2004. – 1056 с.

16. Негус, К. Linux. Библия пользователя / Кристофер Негус. – Диалектика, 2005. – 704 с.

17. Немет, Э. Руководство администратора Linux. Установка и настройка / Эви Немет, Гарт Снайдер, Трент Хейн. – 2-е изд. – Вильямс, 2011. – 1072 с.

18. Птицын, К.А. Серверы Linux. Самоучитель / Птицын Константин Александрович. – Диалектика, 2003. – 208 с.

19. Реймонд, Э.С. Искусство программирования для Unix / Эрик С. Реймонд. – Вильямс, 2005. – 544 с.

20. Смит, Р. FreeBSD: полный справочник / Родерик Смит. – Вильямс, 2005. – 672 с.

21. Смит, Р. Сетевые средства Linux / Родерик Смит. – Вильямс, 2003. – 672 с.

22. Собелл, М. Г. Практическое руководство по Red Hat Linux: Fedora Core и Red Hat Enterprise Linux. – 2-е издание (Practical Guide to Red Hat Linux: Fedora Core and Red Hat Enterprise Linux). – Вильямс, 2005. – 1072 с.

23. Уолтон, Ш. Создание сетевых приложений в среде Linux / Шон Уолтон. – Вильямс, 2001. – 464 с.

24. Уэлш, М. Руководство по установке и использованию системы Linux / [М. Уэлш и др.]. – М.: ИЛКиРЛ, 1999.

25. Хадсон, П. Red Hat Linux Fedora 4. Полное руководство / Пол Хадсон, Эндрю Хадсон, Билл Болл, Хойт Дафф. – Вильямс, 2006. – 1104 с.

26. Хатч, Б. Секреты хакеров. Безопасность Linux – готовые решения / Брайан Хатч, Джеймс Ли, Джордж Курц. – 2-е изд. – Вильямс, 2004. – 704 с.

27. Чеботарев, А.В. Библиотека Qt 4. Программирование прикладных приложений в среде Linux / Арсений Викторович Чеботарев. – Диалектика, 2006. – 256 с.

2. СУПЕРКОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ Классификация алгоритмов по типу параллелизма

В настоящее время ощущается, что дальнейшее повышение производительности компьютеров только за счет улучшения характеристик элементов электроники, в рамках традиционных технологий, ограничивается физическими пределами свойств материалов. Поэтому дальнейшее повышение производительности возможно лишь за счет распараллеливания процессов обработки информации.

Идея сокращения времени выполнения большого объема работ, путем разбиения на отдельные работы, которые могут выполняться *независимо* и *одновременно,* не нова и используется во многих отраслях (строительстве, машиностроении и др.). В вычислительных машинах это достигается путем увеличения числа одновременно работающих процессоров. При этом возникает естественное желание обычную последовательную программу, которая ранее уже запускалась на однопроцессорном компьютере, перенести на многопроцессорную вычислительную систему. Как ни странно, это может не дать выигрыша в производительности, более того, часто в результате такого переноса программа работает медленнее.

При переходе к использованию многопроцессорных систем необходимо провести структурный анализ алгоритма и выявить потенциальные возможности его распараллеливания. Способность алгоритма к распараллеливанию потенциально связана с одним из двух (или одновременно с обоими) внутренних свойств, которые характеризуются как *параллелизм задач* (message passing) и *параллелизм данных* (data parallel).

Если алгоритм основан на параллелизме задач, вычислительная задача разбивается на несколько, относительно самостоятельных подзадач, каждая из которых загружается в «свой» процессор. Каждая подзадача реализуется независимо, но использует общие данные и/или обменивается результатами своей работы с другими подзада-

чами. Для реализации такого алгоритма на многопроцессорной системе необходимо выявлять независимые подзадачи, которые могут выполняться параллельно. Часто это оказывается далеко не очевидной и весьма трудной проблемой.

При наличии в алгоритме свойства параллелизма данных, одна операция может выполняться сразу над всеми элементами исходного массива данных. В этом случае различные фрагменты массива могут обрабатываться независимо на разных процессорах. Для алгоритмов этого типа распределение данных между процессорами обычно осуществляется до выполнения задачи на ЭВМ. Построение алгоритма, обладающего свойством параллелизма данных, и подбор подходящей архитектуры компьютера для него могут выполняться с использованием достаточно простых методик, не требующих применения сложного математического аппарата.

Для того чтобы в полной мере использовать структурные свойства алгоритма, необходимо выявить, к какому типу он относится. Ниже приводится общая классификация алгоритмов, с точки зрения типа параллелизма [1].

1. *Алгоритмы с распределением данных (Data Partitioning).* Это алгоритмы решения задач, в которых пространство данных может быть разделено на непересекающиеся области, с каждой из которых связаны независимые процессы, оперирующие каждый со своими данными.

2. *Алгоритмы, использующие параллелизм данных (Data Parallelism).* Этот тип параллелизма характерен для численных алгоритмов обработки данных, которые также могут быть разделены на непересекающиеся области, обработка которых может осуществляться независимо, но требуется обмен данными между параллельными процессами.

3. *Алгоритмы с синхронизацией итераций (Synchronous Iteration).* Синхронизация в конце каждой итерации заключается в том, что разрешение на начало следующей итерации дается после

того, как все процессоры завершили предыдущую итерацию. Обычно к таким алгоритмам приводятся задачи, в которых исходные данные могут быть разделены на области с некоторым перекрытием, при этом возникает необходимость обмена данными на границах.

4. Релаксационные алгоритмы (Relaxed Algorithm). В отличие от предыдущего алгоритм может быть представлен в виде независимых процессов без синхронизации связи между ними, но процессоры должны иметь доступ к общим данным.

5. Самовоспроизводящиеся задачи (Replicated Workers). Для задач этого класса создается и поддерживается центральный пул (хранилище) похожих вычислительных задач. Параллельно реализуемые процессы осуществляют выбор задач из пула, выполнение вычислений и добавление новых задач к пулу. Вычисления заканчиваются, когда пул пуст. Эта технология характерна для исследований графа или дерева.

6. Конвейерные вычисления (Pipelined Computation). Этот тип вычислений характерен для процессов, которые могут быть представлены в виде некоторой регулярной структуры, например, в виде кольца или двумерной сети. Каждый процесс, находящийся в узле этой структуры, реализует определенную фазу вычислений.

Нетрудно заметить, что некоторые алгоритмы из приведенного списка обладают явно выраженными свойствами параллелизма задач или параллелизма по данным. Вместе с тем ряд алгоритмов в той или иной мере обладают *обоими* указанными свойствами. В ходе структурного анализа должны быть выявлены типы параллелизма и выбрана схема формирования параллельного алгоритма.

Общий подход к построению параллельных алгоритмов заключается в их представлении в виде одного или нескольких взаимодействующих процессов. Процессы – это логически завершенные элементарные работы, на которые можно и целесообразно разбить выполняемый алгоритм. Структура общего алгоритма должна устанав-

ливать правила взаимодействия процессов. При формировании процессов основная проблема – выделение элементарных работ, выполнение которых в вычислительной системе подлежит распараллеливанию.

С точки зрения простоты организации параллельных вычислений, конечно, удобнее делить задачу на крупные подзадачи, каждая из которых решается на «своем» процессоре. Однако при этом будет задействовано небольшое число процессоров и добиться высокого ускорения не удастся. Выбор степени «измельчения» задачи и обоснование числа параллельных процессоров является трудно формализуемым, в значительной степени творческим, этапом, успешность которого часто зависит от наличия у пользователя опыта решения подобных задач.

Часто существенное повышение эффективности параллельной реализации алгоритма достигается путем коренного пересмотра математической формулировки задачи. Следует иметь в виду, что наибольший эффект достигается, если при формулировке задачи сразу принимается во внимание конкретная архитектура многопроцессорной системы, на которой предполагается ее решение. Такая формулировка не всегда очевидна, однако усилия в этом направлении всегда оправданы, т.к. параллельные алгоритмы, построенные с учетом конкретной архитектуры параллельной ЭВМ, как правило, дают наибольшее ускорение и эффективность.

Поэтому пользователю, приступающему к решению сложной в вычислительном отношении задачи, прежде всего, целесообразно познакомиться с архитектурой вычислительной системы, на которой предполагается реализация параллельной программы. Ниже приводится краткий обзор типовых архитектур вычислительных систем.

2.2. Архитектуры вычислительных систем

Для того, чтобы подобрать вычислительную систему, на которой может быть эффективно реализован конкретный параллельный алгоритм, или построить вычислительную схему, наиболее подходящую

для имеющейся в распоряжении пользователя вычислительной системы, необходимо ясно представлять потенциальные возможности различных архитектур. Рассмотрим некоторые наиболее известные классификации и особенности построения компьютерных систем.

При разработке параллельного алгоритма наиболее важным фактором является тип оперативной памяти. В зависимости от организации подсистем оперативной памяти параллельные компьютеры можно разделить на следующие два класса.

Системы с разделяемой памятью (*мультипроцессоры*), у которых имеется одна виртуальная память, а все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти (uniform memory access или UMA). По этому принципу строятся векторные параллельные процессоры (parallel vector processor или PVP) и симметричные мультипроцессоры (symmetric multiprocessor или SMP), которые состоят из совокупности процессоров, имеющих разделяемую общую память с единым адресным пространством и функционирующих под управлением одной операционной системы (рис. 2.1).

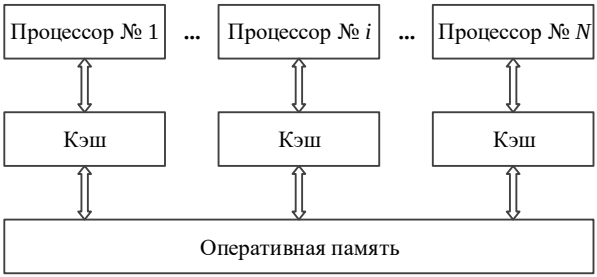


Рис. 2.1. Архитектура многопроцессорных систем
с общей (разделяемой) памятью

Конечно, самый простой способ коммутации процессоров – использование общей шины. В данном случае пользователю не нужно заботиться о распределении данных, достаточно лишь один раз задать соответствующую структуру данных в оперативной памяти и построить процедуру выбора необходимых данных из этой памяти в про-

цессе решения задачи. Однако в таких системах даже небольшое увеличение числа процессоров, подключаемых к общей шине, делает ее узким местом. Поэтому компьютеры этого класса обычно имеют небольшое число процессоров, т.к. наращивание числа процессоров ведет к быстрому росту потерь на межпроцессорный обмен данными.

Системы с распределенной памятью (*мультикомпьютеры*), у которых каждый процессор имеет свою локальную оперативную память, а у других процессоров доступ к этой памяти отсутствует. Этот класс систем часто называют также *массово-параллельные* (или *массивно-параллельные*) компьютеры.

На рис. 2.2 показана общая схема связей основных элементов системы в архитектуре многопроцессорных систем с распределенной памятью. Преимущество этой архитектуры – возможность практически неограниченного наращивания числа процессоров. Для систем этого класса характерно также низкое значение отношения цена/производительность. При работе на компьютере с распределенной памятью необходимо создавать копии исходных данных на каждом процессоре. Поэтому наиболее подходящими для реализации на системах данного класса являются алгоритмы *с распределением данных*, в которых пространство данных декомпозируется без пересечений, а каждая область может обрабатываться отдельно без обмена.

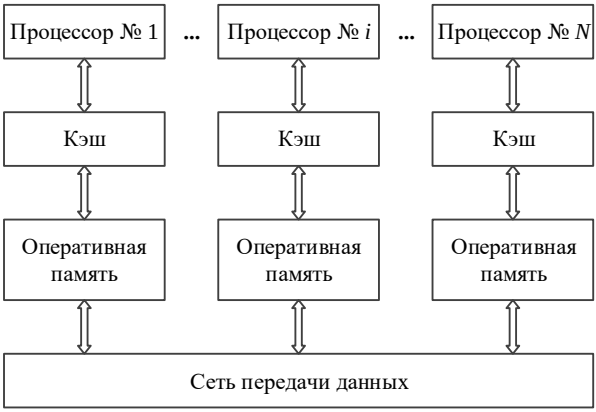


Рис. 2.2. Архитектура многопроцессорных систем с распределенной памятью

Однако часто это вызывает серьезные трудности, связанные с недостаточным объемом памяти в узлах. Поэтому при реализации параллельного алгоритма на компьютерах с распределенной памятью важно построить рациональную, с точки зрения потерь на обмен и допустимых объемов памяти в узлах, схему размещения данных.

Компромисс между системами с *разделяемой* и *распределенной* памятью достигается с использованием программно-аппаратных решений, реализующих неоднородный доступ к памяти (non-uniform memory access или NUMA). При такой реализации общая память является физически распределенной, однако все процессоры имеют доступ к памяти любого процессора (*распределенная общая память*). Это позволяет увеличить число процессоров, но сохранить возможность работы в рамках единого адресного пространства при приемлемых потерях на межпроцессорный обмен. Основная проблема, которую решают в рамках этого подхода – обеспечение когерентности кэш-памяти отдельных процессоров. Эта трудность преодолевается применением специальных программно-аппаратных средств. На рис. 2.3 приведен пример схемы связей элементов в мультипроцессорных системах с неоднородным доступом к разделяемой памяти.

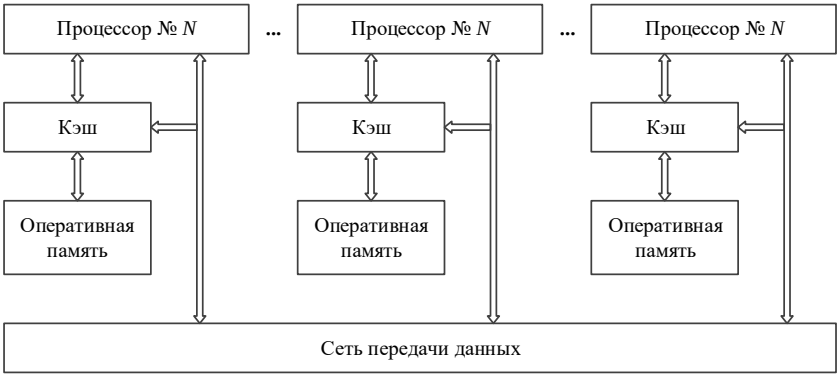


Рис. 2.3. Архитектура мультипроцессорной системы с неоднородным доступом к разделяемой памяти

Использование распределенной общей памяти (distributed shared memory или DSM) упрощает проблемы создания параллельных программ. При этом в системе может функционировать огромное число параллельных процессоров, однако время доступа к локальной и удаленной памяти может существенно различаться. В данном случае необходимо планировать распределение процессоров и схему обмена данными так, чтобы минимизировать число обращений к удаленной памяти. Это может быть учтено на этапе составления расписания параллельного алгоритма. Соответствующая методика будет описана в следующих разделах. Заметим также, что системы с разделяемой памятью обычно имеют высокую стоимость.

Известна также другая классификация компьютеров, основанная на понятии *потока*, под которым понимается последовательность команд или данных, обрабатываемых процессором:

- SISD (Single Instruction stream/Single Data stream) – один поток команд и один поток данных;
- SIMD (Single Instruction stream/Multiple Data stream) – один поток команд и множество потоков данных;
- MISD (Multiple Instruction stream/Single Data stream) – множество потоков команд и один поток данных;
- MIMD (Multiple Instruction stream/Multiple Data stream) – множество потоков команд и множество потоков данных.

Класс SISD являлся характерным для архитектуры ранних компьютеров. Современные компьютеры чаще строятся на основе MIMD-архитектуры или с использованием комбинации нескольких архитектур в одной системе.

Векторно-конвейерные компьютеры, в которых используются векторные команды, обеспечивающие выполнение операций с массивами независимых данных за один такт (SIMD). Типичным представителем данного направления является линия «классических» век-

торно-конвейерных компьютеров CRAY. Собственно, появление термина *суперкомпьютер* связано именно с созданием высокопроизводительного компьютера Cray-1 (1976) с *векторной* архитектурой.

Эффективность использования векторной архитектуры зависит от наличия в алгоритме большого числа одинаковых и независимых операций. Если же вычисление некоторого элемента массива не может начаться, пока не будет вычислен предыдущий элемент, такой алгоритм не векторизуется. Другими словами, *зависимость между операциями* всегда препятствует векторизации. В векторной программе операции выполняются над всеми элементами регистра. В параллельной программе каждый из процессоров выполняет команды, оперируя со своими собственными регистрами. В обоих случаях действия выполняются одновременно, однако каждый из процессоров параллельной ЭВМ может реализовывать алгоритм, отличающийся от алгоритмов других процессоров. Другими словами, алгоритм, который векторизуется, всегда можно и распараллелить. Обратное утверждение не всегда верно.

Кластеры образуются из вычислительных узлов, объединенных системой связи или посредством разделяемой внешней памяти. Кластерные проекты связаны с появлением на рынке недорогих микропроцессоров и коммуникационных решений. В результате появилась реальная возможность создавать установки «суперкомпьютерного» класса из составных частей массового производства. При этом могут использоваться как специализированные, так и универсальные сетевые технологии, которые обычно и определяют эффективность кластерных решений.

Для характеристики сетей в кластерных системах используют два параметра: латентность и пропускную способность. *Латентность* – это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если алгоритм содержит много коротких сообщений, то наиболее критической характеристикой является латентность.

Если передача сообщений организована большими пакетами данных, более важной является пропускная способность.

Для повышения эффективности реализации параллельной программы на кластере необходимо обеспечить *равномерную загрузку всех процессоров*. Если вычислительная система неоднородна, а объемы вычислительных работ распределены между процессорами равномерно, то часть процессоров может простаивать.

2.3. Инженерный подход к построению параллельных алгоритмов

Разработку параллельного алгоритма обычно осуществляет (по крайней мере, начинает) специалист, работающий в некоторой предметной области. При этом чаще всего он вынужден ориентироваться на доступную ему вычислительную систему с некоторой сложившейся архитектурой. Поэтому естественно стремление находить некоторое приемлемое решение, руководствуясь не вполне строгими, но проверенными на практике, приемами и правилами. В частности, если решаемая задача обладает свойством *параллелизма по данным*, то эти правила обычно очевидны и позволяют строить весьма эффективные параллельные алгоритмы.

Совокупность выработанных на основе опыта и здравого смысла методов и приемов распараллеливания задач, обладающих свойством декомпозиируемости по данным, будем называть *инженерным* подходом. В данном случае, существо дела сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Основной целью построения параллельной программы в этом случае является обеспечение заданной степени загрузки всех процессоров. Для этого необходимо, с одной стороны необходимо обеспечить равномерную загрузку процессоров, возможно, с учетом их различной производительности. С другой стороны, необходимо обеспечить допустимое соотношение временных затрат на пересылку данных (накладные расходы) и проведение вычислений на фрагментах исходных данных [2, 3].

Для реализации указанных целей при построении параллельного алгоритма в рамках инженерного подхода обычно решаются следующие задачи [3]:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.
2. Определение для этого набора подзадач информационных взаимодействий.
3. Масштабирование подзадач, определение количества процессоров.
4. Закрепление подзадач за процессорами, составление расписания.

Характер и последовательность указанных действий могут различаться в зависимости от архитектуры системы, числа доступных процессоров и др. Если число подзадач (областей данных) меньше числа доступных процессоров, выполняют *масштабирование* параллельного алгоритма, которое сводится к декомпозиции первоначально заданных областей данных. Для сокращения количества подзадач укрупняют области исходных данных, притом в первую очередь объединяют области, для которых соответствующие подзадачи обладают высокой степенью информационной взаимозависимости.

Распределение подзадач между процессорами очевидно, если количество областей данных совпадает с числом доступных процессоров, а топология сети передачи данных – полный граф, т.е. все процессоры связаны между собой. Если это не так, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных. Требование минимизации информационных обменов между процессорами может вступить в противоречие с условием равномерной загрузки. Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений изменяется в ходе решения задачи. При этом целесообразна динамическая балансировка в ходе выполнения программы.

Способ разделения вычислений на независимые части зависит от того, в какой степени решаемая задача обладает свойством декомпозируемости по данным, другими словами, какое место занимает алгоритм в классификации, приведенной в разделе 2.1.

Простейший и наиболее благоприятный с точки зрения организации параллельных вычислений случай, когда вся область исходных данных задачи может быть разделена на непересекающиеся области любых размеров, а вычисления в каждой области могут вестись независимо (*Алгоритмы с распределением данных*). В этом случае задача построения параллельного алгоритма проста: необходимо всю область разбить на подобласти, число которых равно числу доступных процессоров, а размеры подобластей подобрать так, чтобы обеспечить их равномерную загруженность, с учетом производительности каждого.

Можно рекомендовать следующий простой способ построения эффективной параллельной программы, совмещенный с этапом ее отладки. Размеры фрагментов массива исходных данных уменьшают (соответственно увеличивают число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном запуске программы с увеличенным числом процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров. Этот подход обсуждался в работе [4].

Если задача не допускает распараллеливания по данным, т.е. возможен лишь *параллелизм задач*, трудности существенно возрастают. Подход к программированию, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач, для каждой из которых пишется собственная программа и загружается в «свой» процессор. Эти подзадачи могут взаимодействовать сложным образом, при этом распределение и закрепление подзадач, обеспечивающее равномерную

загрузку процессоров, становится не столь очевидным, как это мы видели при декомпозиции по данным. Для построения эффективного кода в данном случае необходимо вскрыть более тонкую структуру параллелизма алгоритма. Наиболее подходящим математическим аппаратом в этом случае являются модели в виде графов.

2.4. Понятие и построение модели параллельных вычислений

Модель параллельных вычислений будем описывать графом

$$G_s = (V, P, E, I_s), \tag{2.1}$$

где V – множество вершин графа, представляющих операции алгоритма; $P = \{t_i\}, i = \overline{1,n}$ – множество весов вершин; E – множество дуг графа, устанавливающих зависимости между операциями (для простоты мы не рассматриваем случай, когда дугам также приписаны веса); I_s – множество натуральных чисел $i \in I_s, i = 1, 2, ..., s$, используемых для нумерации s доступных для реализации алгоритма процессоров.

В ходе составления расписания необходимо за каждым процессором (целым числом) закрепить оператор, т.е. каждой операции $v_i \in V$ из множества V поставить в соответствие номер процессора $i \in I_s$. Еще раз подчеркнем, что в этой модели мы отказываемся от использования параметра t_i – время начала выполнения операции. В модели (2.1) при заданном множестве параметров $P = \{t_i\}, i = \overline{1,n}$, определяющих время выполнения каждой операции, множество связей E однозначно определяет время начала выполнения операции, притом так, что к началу каждой операции на любом процессоре все необходимые данные уже вычислены.

Приведённую модель (2.1) удобно представлять в виде временной диаграммы выполнения операторов. В диаграмме (рис. 2.4*б*) операторы обозначаются прямоугольниками с длиной, равной времени выполнения соответствующего оператора. Номера операторов на диа-

грамме обозначаются целыми числами в соответствии с их нумерацией в графе алгоритма. Для определенности условимся обозначать их цифрой в кружке, расположенном в центре прямоугольников. Ось ординат диаграммы разбивается на интервалы, каждый из которых соответствует одному из параллельно работающих процессоров. В каждом интервале размещаются только те прямоугольники-операторы, которые закреплены за соответствующим этому интервалу процессором.

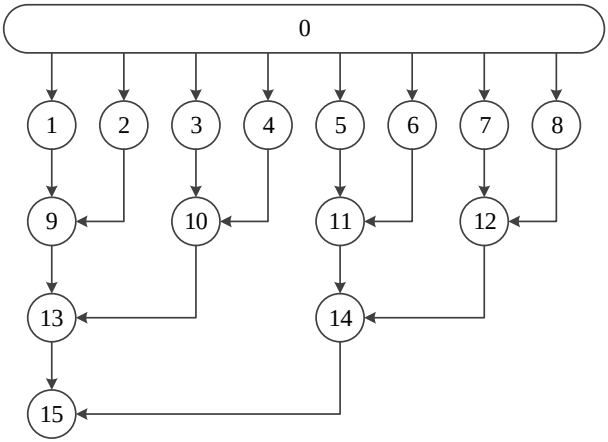
Связи между прямоугольниками-операторами обозначаются цифрами слева и справа от номера оператора, указанного в кружочке в центре. В частности, в левой части прямоугольника-оператора записываются номера предшествующих по информационной связи операторов, а в правой части – номера операторов, следующих за данным оператором. Поскольку каждый прямоугольник диаграммы размещается в интервале «своего» процессора, описанный способ их нумерации отражает также связь между процессорами.

Описанный способ представления модели параллельных вычислений является исчерпывающим описанием расписания параллельного алгоритма и может быть удобным программисту для написания параллельной программы. При этом информация о номерах связанных операторов может использоваться для оценки объема передаваемых данных, как между процессами, так и между процессорами. На рис. 2.4*б* в виде диаграммы, составленной в соответствии с описанными выше правилами нумерации операторов, приведен пример модели параллельных вычислений алгоритма, представленного графом на рис. 2.4*а*.

Построение модели параллельных вычислений в случае, когда задача декомпозируется по данным (в рамках инженерного подхода), обычно не представляет трудностей. В этом случае задача сводится лишь к закреплению процессоров за фрагментами, на которые разбита вся область исходных данных. Если при этом после каждой итерации обработки фрагментов требуется обмен данными, необходимо

так распределять работы между процессорами, чтобы на каждой итерации длины прямоугольников-операторов были одинаковыми.

а)



б)

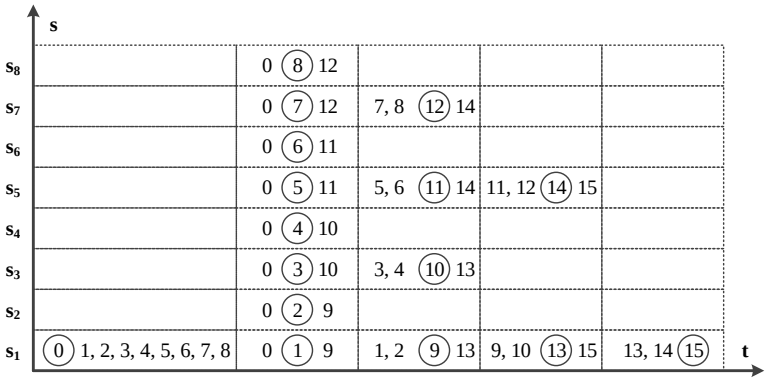


Рис. 2.4. Граф (*а*) и временная диаграмма (*б*) алгоритма умножения матрицы на вектор при разделении матрицы по строкам

В качестве примера построим модель параллельных вычислений в виде временной диаграммы для задачи умножения матрицы на вектор. Выбор этого примера обусловлен тем, что матричные операции являются классическими примерами для демонстрации многих приемов и методов распараллеливания задач. Далее для общности будем

полагать, что матрица является плотной, т.е. число нулевых элементов в них мало по сравнению с общим количеством элементов матриц.

Ограничимся рассмотрением ленточного разбиения задачи умножения матрицы на вектор по строкам, при котором каждому процессору выделяется некоторое количество, обычно подряд идущих, строк.

На рис. 2.4*а* приведен граф алгоритма параллельного умножения матрицы на вектор на 8 процессорах при ленточном разбиении по строкам. Здесь цифрой 0 обозначена операция распределения данных (разделения матрицы на ленточные фрагменты и рассылка этих фрагментов и умножаемого вектора процессорам); цифрами 1-8 обозначены операции умножения выделенных каждому процессору полос на вектор. В результате выполнения этих операций на каждом процессоре будет получена 1/8 часть искомого вектора. Цифрами 9-15 обозначены операции «сборки». Сборка осуществляется за три шага. После первого шага (выполнения операций 9-12) на 4 процессорах окажется по 1/4 части искомого вектора. На следующем шаге (операции 13-14) на двух процессорах будет сформировано по 1/2 части искомого вектора. Наконец, на завершающем 3-м шаге в результате выполнения операции 15 на процессоре № 1 будет получен результирующий вектор.

Соответствующая этому графу временная диаграмма параллельных вычислений приведена на рис. 2.4*б*. Если подготовлены фрагменты программы, реализующие фигурирующие на графе алгоритма операции, то по этой диаграмме нетрудно «собрать» параллельную программу решения всей задачи. Мы продолжим этот пример в разделе 2.6, в частности, приведем MPI-программу, реализующую эту модель параллельных вычислений.

При построении модели параллельных вычислений очень важно сформулировать цель. Например, если мы хотим построить реализацию некоторого информационного графа с максимально возможным

быстродействием, то ясно, что вычислительный процесс должен быть организован в соответствии с канонической строгой параллельной формой. При этом максимальное число одновременно работающих процессоров должно быть равным максимальной ширине яруса. Однако если число доступных процессоров меньше максимальной ширины яруса или число ярусов максимальной ширины мало, а их ширина велика, то целесообразно минимизировать число используемых процессоров, чтобы исключить простаивание большей их части.

2.5. Построение оценок производительности и эффективности параллельных алгоритмов

Для построения оценок производительности и эффективности мы будем использовать следующую модель вычислительной системы [2]. Система состоит из набора *простых функциональных устройств* (ФУ), не имеющих *собственной памяти*, т. е. никакая последующая операция не может начаться раньше, чем предыдущая, а результат предыдущего срабатывания ФУ может сохраняться в нем только до момента очередного срабатывания. ФУ.

Обозначим τ – время выполнения одной операции (*стоимость операции*), а T – время выполнения работы (*стоимость работы*). За это время может быть выполнено приблизительно T/τ операций (в действительности следует брать только целую часть результата деления). *Загруженностью устройства – pE* называют отношение стоимости реально выполненной работы к максимально возможной стоимости.

Максимальный объем работ, который может быть выполнен системой, оценивается *пиковой производительностью*, определяемой как максимальное количество операций, которое может быть выполнено системой за единицу времени при отсутствии потерь времени на связи между ФУ. Пиковая производительность определяется как количество операций с плавающей точкой (простых ФУ), выполняемых за один такт, умноженное на частоту работы процессора и на число

процессоров. Единица измерения производительности – Flops (одна вещественная операция в секунду).

Показатель эффективности одного процессора – количество операций, запускаемых за один такт процессора – IPC (instructions per cycle). *Реальная производительность* вычислительной системы – это количество операций, реально выполняемых в среднем в единицу времени. Отношение реальной производительности к пиковой характеризует *эффективность* реализации задачи на данном конкретном компьютере. Превышение пиковой производительности над реальной характеризует, насколько данная архитектура приспособлена к решению конкретной задачи.

Если s устройств системы имеют пиковые производительности $\pi_1,\ldots,\pi_s$ и работают с загруженностями $p_1,\ldots,p_s$, то реальная производительность системы выражается формулой [2]

$$r = p\pi\,, \tag{2.2}$$

где $\pi = \pi_1 + \ldots + \pi_s$, а p – *загруженность системы*, определяемая как

$$p = \sum_{i=1}^{s} \alpha_i p_i, \quad \alpha_i = \frac{\pi_i}{\sum_{j=1}^{s} \pi_j}\,. \tag{2.3}$$

Из (3.2) видно, что для достижения наибольшей реальной производительности системы при фиксированном числе устройств необходимо обеспечить наиболее полную ее загруженность. Дальнейшее повышение производительности достигается увеличением числа устройств.

Ускорение реализации алгоритма на вычислительной системе из s устройств определяется как [2]

$$R_s = r/\pi_s\,, \tag{2.4}$$

где π_s – пиковая производительность самого быстродействующего устройства системы. Это означает, что наибольшее ускорение – s системы из s устройств может достигаться только в случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены.

Реальное ускорение для однородных вычислительных систем, имеющих одинаковую производительность устройств, часто определяют также как отношение времени решения задачи на одном процессоре – T_1 к времени T_s решения той же задачи на системе из *s* таких же процессоров:

$$R = T_1 / T_s . \tag{2.5}$$

Это соотношение можно получить также из форумы (2.4) с учетом форумы (2.2), т.к. при одинаковой производительности устройств $p = T_1 / T_s \cdot s$, а $\pi = \pi_s \cdot s$.

Отношение *реального ускорения* к числу используемых процессоров *s*:

$$E_s = R/s = T_1/\left(T_s \cdot s\right), \tag{2.6}$$

называют *эффективностью* системы. Второе равенство в (2.6) показывает, что при одинаковой производительности устройств эффективность системы совпадает со значением загруженности системы. Наилучшие показатели ускорения и эффективности – соответственно $R=s$, $E_s = 1$.

Для анализа производительности вычислительных систем, в которых имеют место направленные связи между устройствами, используют модель ориентированного *графа* [2]. В частности, с использованием этой модели в показано, что для системы из *s* устройств с пиковыми производительностями $\pi_1,\ldots,\pi_s$, описываемой связным графом, максимальная производительность $r_{\max}$ определяется как

$$r_{\max} = s \min_{1 \le i \le s} \pi_i .$$

Отсюда вытекает 1-й *закон Амдала*: Производительность вычислительной системы, состоящей из связанных между собой устройств, определяется самым непроизводительным устройством.

Для количественной оценки производительности многопроцессорных вычислительных систем используется 2-й *закон Амдала* [2]. В

соответствии с этим законом максимально возможное ускорение для системы, состоящей из s одинаковых устройств, определяется как

$$R = s \Big/ \left(\beta \cdot s + \left(1 - \beta\right) \right), \tag{2.7}$$

где $\beta = n/N$, N – общее число операций алгоритма, а n – число операций, которые могут выполняться только последовательно. Формула Амдала используется для прогноза достижимого ускорения на этапе анализа параллелизма задачи. Например, если по описанию алгоритма установлено, что половина операций не поддаются распараллеливанию, максимально достижимое ускорение в случае использования 2 процессоров в соответствии с (2.7) составит около 1,33, для 10 процессоров – менее 1,82, а для 100 процессоров – около 1,98.

Наряду с ускорением и эффективностью важным показателем является также масштабируемость вычислительной системы. Параллельный алгоритм называют масштабируемым (scalable), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении эффективности использования процессоров.

Для характеристики свойств масштабируемости оценивают накладные расходы (время T_0) на организацию взаимодействия процессоров, синхронизацию параллельных вычислений и т.п.:

$$T_0 = sT_s - T_1, \tag{2.8}$$

где T_s , T_1 – те же, что и в (2.5). Используя эти обозначения, соотношения для времени параллельного решения задачи и соответствующего ускорения можно представить в виде

$$T_s = \left(T_1 + T_0\right) \Big/ s, \tag{2.9}$$

$$R_s = T_1 / T_s = sT_1 \Big/ \left(T_1 + T_0\right). \tag{2.10}$$

Соответственно эффективность использования s процессоров

$$E_s = R_s / s = T_1 \Big/ \left(T_1 + T_0\right) = 1 \Big/ \left(1 + T_0 / T_1\right). \tag{2.11}$$

Из (2.11) следует, что если время решения последовательной задачи фиксировано (T_1=const), то при росте числа процессоров эффективность может убывать лишь за счет роста накладных расходов T_0.

В заключение этого раздела, посвященного анализу производительности, заметим, что для начинающего пользователя может показаться удивительным тот факт, что, запустив программу на компьютере с огромной пиковой производительностью, можно не получить ускорения или даже получить замедление счета по сравнению с обычным персональным компьютером. Для правильной оценки ситуации в этом случае прежде всего необходимо сопоставить граф алгоритма и граф вычислительной системы. Наряду с общими, влияющими на производительность, факторами, которые рассматривались выше, часто недооценивается влияние кэш-памяти. Эффективность кэш-памяти существенно выше, если в задаче большой объем *локальных вычислений* и *локального использования данных*. В моменты времени, когда объем локальных данных превышает размеры кэш-памяти, наблюдается резкое падение производительности вычислительной системы.

При решении задач на кластере одним из основных факторов являются также *коммутационные сети.* Они определяют накладные расходы – время задержки передачи сообщения. Это время зависит от латентности (начальной задержки при посылке сообщений) и длины передаваемого сообщения. Для правильной трактовки временных затрат надо сопоставлять характер и множество обменов по графу алгоритма с характеристиками используемой сети. Конечно, на производительность параллельного компьютера огромное влияние оказывают также операционная система, драйвера сетевых устройств, программы, обеспечивающие сетевой интерфейс нижнего уровня, компиляторы и др. Однако этими факторами программист уже не может управлять в полной мере, поэтому мы их не обсуждаем. Достаточно подробное рассмотрение этих вопросов можно найти в учебных пособиях, посвященных параллельному программированию [3, 4].

2.6. Подготовка и запуск параллельных программ

Цель настоящего раздела – дать начальные сведения по подготовке, отладке и запуску параллельных программ на кластере. При

этом имея в виду скорейшее освоение технологии кластерных вычислений, мы ограничимся рассмотрением минимального набора программных средств для подготовки параллельных приложений. Для быстрого знакомства с этими инструментами, мы рассмотрим их применение в простой программе ввода-вывода данных, а также продолжим рассмотренный в разделе 3.4 пример умножения матрицы на вектор с ленточной декомпозицией по строкам.

При реализации параллельного алгоритма на многопроцессорной вычислительной системе (например, кластерной архитектуры) запускаются программы, написанные на языке последовательного программирования Си или Фортран. Каждая такая программа выполняет некоторую операцию обработки «своей» порции данных. Взаимодействие этих программ обеспечивает библиотека MPI (Message Passing Interface – Интерфейс Передачи Сообщений) путем вызова функций синхронизации, посылки и получения сообщений. Работа с библиотекой MPI несколько различается при программировании на языках Си и Фортран. Здесь излагается вариант библиотеки при программировании на Си.

MPI-программа запускается одновременно на параллельно работающих процессорах. Каждая копия программы реализует отдельный процесс. Процессы изолированы друг от друга в том смысле, что у них разные области кода, стека и данных. Процессы обмениваются друг с другом данными в виде сообщений. Сообщения обозначаются идентификаторами. Каждый процесс может получать информацию о количестве одновременно запущенных процессов и свой номер. Процессы могут объединяться в группы.

Важной особенностью MPI является понятие так называемой области связи (*communication domains*). В программе для описания области связи используются коммуникаторы. При запуске программы для описания стартовой области связи библиотека автоматически создает коммуникатор с использованием идентификатора MPI_COMM_WORLD.

В области связи размещаются все процессы [4], а коммуникатор ограничивает многие функции MPI, прикрепляя их к этой области связи. Если для одной области связи используется несколько коммуникаторов, программа работает с ними как с несколькими разными областями.

Все идентификаторы начинаются с префикса «MPI_». Поэтому нельзя использовать собственные идентификаторы, начинающиеся с этого префикса, а также с префиксов «MPID_», «MPIR_» и «PMPI_», которые применяются в служебных целях. Имена констант (и неизменяемых пользователем переменных) записываются полностью заглавными буквами: MPI_COMM_WORLD, MPI_FLOAT.

В именах функций первая за префиксом буква заглавная, остальные маленькие: MPI_Send, MPI_Comm_size.

Подчеркнем, что состояние регистра символов при указании имен функций (процедур) и именованных констант при программировании на языке Си существенен (это не имеет значения только при программировании на Фортране).

Определение всех именованных констант, прототипов функций и определение типов выполняется в языке Си подключением файла mpi.h.

В операциях пересылки и преобразования данных указываются собственные типы MPI. Это обеспечивает переносимость программ между различными компиляторами и платформами, а также возможность автоматического преобразования типов данных при пересылках, когда программа реализуется на неоднородной вычислительной системе.

В табл. 2.1, заимствованной из [6], приведен список типов библиотеки MPI и соответствующих типов языка Си. Здесь тип MPI_BYTE используется для передачи двоичной информации без ее преобразования. Тип MPI_PACKED используется для передачи в одном сооб-

щении предварительно упакованных разнотипных данных. Программисту предоставляются также средства создания собственных типов на базе стандартных.

Рассмотрим 4 функции MPI на примере простой программы [4], в результате выполнения которой каждый запускаемый процесс должен выдать следующее сообщение:

«Hello world from process i of n»

Здесь i – номер процесса, а n – количество процессов. Файл с текстом этой программы назовем «helloworld.c». Текст программы на языке Си имеет вид:

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

Таблица 2.1 Соответствие между MPI-типами и типами языка Си

Тип MPI	Тип языка Си
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

В приведенном тексте программы использованы следующие 4 функции MPI: MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Finalize. MPI-программа начинается с вызова функции инициализации MPI: MPI_Init.

В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы, и создается область связи, описываемая коммуникатором MPI_COMM_WORLD.

Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. В данном случае величина groupsize равна числу процессоров, выделенных задаче. При инициализации каждому процессу передаются аргументы функции main, полученные из командной строки.

Функция определения числа процессов в области связи MPI_Comm_size:

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

Здесь коммуникатор comm – является входным параметром, a size – выходной параметр, указывающий число процессов в области связи коммуникатора comm, т.е. эта функция возвращает количество процессов в области связи коммуникатора comm. Если явным образом не создаются группы и связанные с ними коммуникаторы, то значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI.

Функция определения номера процесса MPI_Comm_rank:

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

Здесь коммуникатор comm также является входным параметром, a rank – выходной параметр, указывающий номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size определяется с помощью предыдущей функции).

Функция завершения MPI программ – MPI_Finalize:

```
int MPI_Finalize(void).
```

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

Для запуска MPI-программы используется команда mpirun .

Это специальный командный файл – скрипт, в котором указывается количество запускаемых процессов и имя файла исполняемого модуля (программы). Можно также задать конфигурацию (список компьютеров) на которой запускается программа. Количество процессов не обязательно равно количеству процессоров. Возможен запуск MPI-программы с количеством процессов большим, чем число доступных процессоров, при этом несколько процессов должны одновременно выполняться на одном процессоре. Выполнение программы с указанием запуска нескольких процессов на однопроцессорном компьютере позволяет производить первоначальную отладку параллельного приложения. Рассмотрим основные соглашения по использованию библиотеки MPI.

Откомпилируем программу helloworld.c. Для этого сформируем в командной строке команду

```
% mpicc -o helloworld helloworld.c
```

Запуститим программу на 4-х процессорах, задав в той же строке команду:

```
% mpirun -np 4 helloworld
```

Поскольку порядок появления сообщений от различных процессов не определен, в результате выполнения программы мы можем получить следующее сообщение:

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

Если появляется какое-либо сообщение об ошибке, например:

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

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

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

Как видно из этого примера рассмотренные четыре функции MPI используются в параллельной программе всегда, даже если программа содержит лишь ввод и вывод данных. Теперь рассмотрим еще две функции MPI, осуществляющие собственно распараллеливание процессов: MPI_Send, MPI_Recv.

Эти функции мы рассмотрим на примере программы (далее называемой Multiply.c) из прил. 2 для параллельного умножения матрицы на вектор при ленточном разбиении матрицы по строкам. Модель параллельных вычислений в виде временной диаграммы этого алгоритма была нами рассмотрена в разделе 2.4 (рис. 2.4б). Текст программы Multiply.c с комментариями приведен в приложении А.

Рассмотрим более детально функции передачи (MPI_Send) и приема (MPI_Recv) сообщений. Функция

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

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

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

Здесь входной параметр buf – адрес начала расположения принимаемого сообщения, выходные параметры: count – максимальное число принимаемых элементов; datatype – тип элементов принимаемого сообщения; source – номер процесса-отправителя; tag – идентификатор сообщения; comm - коммуникатор области связи, а также выходной модифицируемый параметр status – атрибуты принятого сообщения. Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

В функции MPI_Recv при указании номера процесса-отправителя возможно использование специального параметра MPI_ANY_SOURCE ("принимай от кого угодно"), а в качестве идентификатора получаемого сообщения – MPI_ANY_TAG ("принимай что угодно"). Это так называемые параметры-джокеры, MPI резервирует для них отрицательные целые числа, в то время как реальные идентификаторы процессов и сообщений лежат всегда в диапазоне от 0 до 32767. Пользоваться джокерами следует с осторожностью, потому что по ошибке таким вызовом MPI_Recv может быть захвачено сообщение, которое должно приниматься в другой части процесса-получателя.

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

В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов – передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. В дополнение к стандартному режиму возможно использование синхронной, буферизованной или согласованной передачи, как с блокировкой, так и без блокировки. Вместе с тем, с помощью только пары операций MPI_Send/MPI_Recv можно реализовать практически любой параллельный алгоритм.

В настоящем пособии мы намеренно ограничились изучением лишь 6 функций MPI, достаточных для написания простейших параллельных программ. Это отвечает нашей цели: помочь пользователям в максимально короткие сроки освоить подготовку и реализацию простейших параллельных MPI-приложений на кластере. Пользователи,

освоившие работу с описанными в настоящем пособии шестью функциями MPI, могут попытаться построить более эффективный параллельный код с использованием, так называемых, коллективных функций MPI. Подробное описание коллективных функций и примеры параллельных программ, в которых они используются, можно найти в учебном пособии [7].

В заключение приведем краткую инструкцию по компиляции и запуску MPI-программ. Процесс запуска MPI-программы включает в себя следующие шаги: выбор реализации MPI, компиляцию MPI-программы и запуск MPI-программы.

Для выбора MPI среды выполните следующие команды:

```
[user@mgt1 ~] $ module avail
/etc/modulefiles
impi/3  impi/4   openmpi
[user@mgt1 ~] $ module load impi/4
[user@mgt1 ~] $
```

В результате выполнения этих команд модуль impi/4 загружает переменные среды для работы с Intel MPI-библиотекой версии 4. Загруженные переменные сохраняются в течении текущей сессии. После выбора реализации MPI можно использовать программы для компиляции и запуска параллельных приложений (mpicc, mpicxx, mpif77, mpif90 и т.д.).

Для компиляции С-программы используется команда

```
mpicc – o <имя файла с объектным кодом> <имя файла с исходным кодом>
```

В данном случае такая команда имеет вид:

```
[user@mgt1 ~] $ mpicxx – o  Multiply.mpi Multiply.c
```

Завершающий этап – запуск программы. Для постановки задания в очередь и передачи всех переменных среды на вычислительные ноды (узлы), используется команда qsub с ключом –V.

Например, для командного файла задания run.pbs:

```
[user@mgt1 ~] $ qsub –V run.pbs
```

Помимо специфических инструкций, отличающихся для каждого суперкомьютера, в файле задания должна присутствовать строка запуска программы:

```
mpirun -np 4 Multiply.mpi a.txt x.txt y.txt
```

Для того, чтобы программа успешно выполнилась и выдала верный результат, необходимо, чтобы в директории, из которой запускается программа присутствовали два файла: a.txt и x.txt. Файлы должны иметь следующую структуру:

a.txt:

```
<Число строк>
<Число столбцов>
<Матрица, в которой значения элементов разделены пробелами, а строки – знаком переноса строки>
```

x.txt:

```
<Вектор, на который необходимо умножить матрицу>
```

Результат выдается в файл y.txt, в котором будет содержаться вектор значений, полученных после умножения, разделенных пробелом.

Для получения информации о статусе запущенной задачи (списка очередей и их параметры; списка задач с расширенной информацией; полной информации о задаче; информации на каких узлах запущена задача) можно использовать команду qstat.

Список литературы

1. Lester, Bruce P. The Art of Parallel Programming / Bruce P. Lester. Prentice Hall, Englewood Cliffs, New Jersey, 1993. – 376 p.

2. Воеводин, В.В. Параллельные вычисления / В.В. Воеводин, Вл.В. Воеводин. – СПб.: БХВ-Петербург, 2002.

3. Гергель, В.П. Лекции по параллельным вычислениям: учеб. пособие / В.П. Гергель, В.А Фурсов. – Самара: Изд-во СГАУ, 2009. – 164 с.

4. Введение в программирование для параллельных ЭВМ и кластеров: учеб. пособие / В.В. Кравчук, С.Б. Попов, А.Ю. Привалов [и др.]. – Самара: Самар. науч. центр РАН; СГАУ, 2000. – 87 с.

5. Барский, А.Б. Параллельные процессы в вычислительных системах. Планирование и организация / А.Б. Барский. – М.: Радио и связь, 1990. – 256 с.

6. Букатов, А.А. Программирование многопроцессорных вычислительных систем / А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. – Изд-во ООО "ЦВВР", 2003.

5. Головашкин, Д.Л. Методы параллельных вычислений: учеб. пособие / Д.Л. Головашкин. – Самара: Изд-во СГАУ, 2002. Ч. I. – 92 с.

3. ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

3.1. Эволюция графических процессоров. Введение в CUDA

Развитие архитектуры вычислительных систем – это история постоянного поиска баланса свойств, оптимального для множества целевых приложений. Пока не был исчерпан ресурс основных факторов роста, массовое производство и экономическая выгода сдерживали сколько-нибудь значительную специализацию основных вычислительных архитектур. Однако каждое новое инженерное решение в своем развитии со временем обнаруживало соответствующий противовес: частота и тепловыделение, многоядерность и когерентность кэшей, общая память и неоднородный доступ, конвейерность и ветвления и т.д. В условиях недостатка новых идей фактором роста в настоящее время становятся специализированные вычислители. Наибольший успех графических ускорителей (GPU) в этом качестве связан с их устойчивым положением в основной сфере применения.

Устройство архитектуры GPU можно кратко охарактеризовать как «макроархитектуру вычислительного кластера, реализованную в микромасштабе». GPU состоит из однородных вычислительных элементов с общей памятью. Каждый вычислительный элемент способен исполнять тысячи потоков, переключение между которыми не имеет накладных расходов. Потоки могут быть сгруппированы в блоки, имеющие общий кэш и быструю разделяемую память, явно контролируемую пользователем. Данная реализация в сочетании с расширениями для процедурных языков программирования носит название *Compute Unified Device Architecture* (CUDA).

Одной из важнейших характеристик любого вычислительного устройства является производительность (*performance*). Для математических расчетов она обычно измеряется в количестве операций над вещественными данными в секунду (*floating-point operations per second,* FLOPS). В зависимости от того, учитывается ли только скорость расчета или также влияние других факторов, таких как скорость

обмена данными, различают максимальную теоретически возможную пиковую и реальную производительность.

Производительность сильно зависит от тактовых частот центрального процессора (CPU) и памяти. Процессоры с высокой частотой и большим количеством интегрированной памяти могли бы обладать превосходной производительностью, но не могут быть массовыми из-за слишком высокой цены. По этой причине основной объем памяти находится в отделенных от процессора внешних модулях, и частота их работы в несколько раз ниже частоты процессора. Таким образом, реальная производительность системы при обработке больших массивов данных в значительной мере характеризуется именно скоростью работы внешней памяти и поэтому может быть значительно ниже пиковой.

Процессоры архитектуры x86 с момента своего появления в 1978 году увеличили свою тактовую частоту с 4,77 МГц до 4 ГГц, т.е. более чем в 800 раз, однако в последние несколько лет рост частоты более не наблюдается. Это связано как с ограничениями технологии производства микросхем, так и с тем, что энергопотребление (а значит и выделение тепла) пропорционально четвертой степени частоты. Таким образом, увеличение тактовой частоты всего в 2 раза приводит к увеличению тепловыделения в 16 раз. До сих пор с этим удавалось справляться за счет уменьшения размеров отдельных элементов микросхем. Дальнейшая миниатюризация связана со значительными трудностями, поэтому в настоящее время рост производительности идет в основном за счет увеличения числа параллельно работающих ядер, т.е. за счет параллелизма.

Максимальное ускорение, которое можно получить, распределив выполнение программы на N параллельно работающих элементов (процессоров, ядер), определяется законом Амдала (2.7). Согласно этому закону при увеличении числа процессоров ускорение стремится к $1/(1-P)$, где P – это доля вычислений, которые возможно распараллелить. Таким образом, если возможно распараллелить 3/4 всех

вычислений в программе, то при сколь-угодно большом числе доступных процессоров ускорение никогда не превысит 4 раз. Закон Амдала показывает, что возможная выгода от использования параллельных вычислений во многом предопределена свойствами применяемых в программе методов или алгоритмов. Поэтому в разработанном курсе уделяется внимание не только архитектуре многопроцессорных систем, но и способам распараллеливания алгоритмов.

Термин Graphics Processing Unit (GPU) был впервые использован корпорацией NVIDIA для обозначения того факта, что графический ускоритель, первоначально используемый только для ускорения трехмерной графики, стал мощным программируемым устройством (процессором), пригодным для решения значительно более широкого класса вычислительных задач (General Purpose computations on GPU – GPGPU).

Современные GPU представляют собой массивно-паралельные вычислительные устройства с производительностью порядка десятков терафлопс и большим объемом (до 24 Гб) собственной памяти (DRAM).

История GPU начиналась более чем скромно: первые графические ускорители Voodoo компании 3DFx лишь выполняли растеризацию (перевод треугольников в массивы пикселов) с поддержкой буфера глубины, наложение текстур и альфаблендинг. При этом вся обработка вершин проводилась центральным процессором, и ускоритель получал на вход уже отображенные на экран (т.е. спроектированные) вершины. Однако именно эти очень простые задачи Voodoo умел решать намного быстрее, чем универсальный центральный процессор, что привело к широкому распространению графических ускорителей трехмерной графики.

Традиционные задачи рендеринга имеют значительный ресурс параллелизма: все вершины и фрагменты, полученные при растеризации треугольников, можно обрабатывать независимо друг от друга.

Данное свойство типовых задач графических ускорителей определило их архитектурные принципы.

Ускорители трехмерной графики быстро эволюционировали, при этом помимо увеличения производительности, также росла и их функциональность. Так, графические ускорители следующего поколения (например, Riva TNT), уже могли обрабатывать вершины без участия CPU и одновременно накладывать несколько текстур. Важно отметить постепенное расширение возможностей программируемой обработки отдельных элементов с целью реализации ряда сложных эффектов, таких как попиксельное освещение. В GeForce 256 были впервые добавлены блоки register combiners, каждый из которых позволял выполнять простые вычислительные операции, например, скалярное произведение. Построение сложного эффекта сводилось к настройке и соединению входов и выходов множества таких блоков.

Следующим шагом стала поддержка в GeForce 2 вершинных программ на специальном ассемблере. Такие программы выполнялись параллельно для каждой вершины в 32-битной вещественной арифметике. Впоследствии подобная функциональность стала доступна уже на уровне отдельных фрагментов в серии GeForce FX. Благодаря тому, что графический ускоритель содержал как вершинные, так и фрагментные процессоры, соответствующие программы для вершин и фрагментов также выполнялись параллельно, что приводило к еще большему быстродействию. В целом графические ускорители на тот момент представляли собой мощные SIMD-процессоры.

Термином SIMD (Single Instruction Multiple Data) называют метод обработки, при котором одна и та же операция применяется одновременно ко множеству независимых данных. SIMD-процессор получает на вход поток однородных данных и параллельно обрабатывает их, порождая выходной поток. Программный модуль, описывающий подобное преобразование, называют вычислительным ядром (kernel). Отдельные ядра могут быть соединены между собой, образуя сложные составные схемы.

С введением поддержки текстур 16- и 32-битных вещественных элементов появился простой и универсальный метод обмена большими массивами данных между GPU и памятью основной системы. Высокоуровневые языки, такие как Cg, GLSL и HLSL, значительно упростили процесс написания программ для GPU. Функциями OpenGL или Direct3D входные данные загружались в текстуры. Затем на графическом процессоре через операцию рендеринга (обычно – прямоугольника) запускалась программа обработки этих данных. Результат получался также в виде текстур, которые оставалось выгрузить обратно в системную память. Таким образом, программа состояла из двух частей, написанных на разных языках: C/C++ – для подготовки и передачи данных и язык шейдеров – для вычислений на GPU.

Тем не менее, API, ориентированные на работу с графикой, имеют ряд ограничений, затрудняющих реализацию вычислительных алгоритмов. Так, отсутствует возможность какого-либо взаимодействия между параллельно обрабатываемыми пикселами, что для вычислительных задач является желательным. В частности, это приводит к отсутствию поддержки операции scatter, применяемой, например, при построении гистограмм: очередной элемент входных данных может приводить к изменению заранее неизвестного элемента (или элементов) гистограммы.

В последние годы на смену графическим API в GPGPU пришли программные системы, предназначенные именно для вычислений – CUDA, DirectCompute, OpenCL. Примечательно, что теперь они оказывают сильное обратное влияние и в сфере своих прародителей – графических приложений. Так, визуальные эффекты во многих современных играх основаны на численном решении дифференциальных уравнений в реальном времени с помощью GPU.

Далее рассмотрены основные отличия CPU и GPU. Во-первых, CPU имеет лишь небольшое число ядер, работающих на высокой тактовой частоте независимо друг от друга. GPU же напротив работает

на низкой тактовой частоте и имеет сотни сильно упрощенных вычислительных элементов (например, отсутствует предсказатель ветвлений и суперскалярное исполнение команд). Во-вторых, значительная доля площади кристалла CPU занята кэшем, в то время как практически весь GPU состоит из арифметико-логических блоков. В архитектуре GPU кэш имеет меньшее значение, поскольку используется принципиально иная стратегия покрытия латентности памяти. За счет этих отличий производительность каждого нового поколения GPU быстро растет как в пиковом значении, так и на реальных приложениях, например, Linpack.

GPU наиболее эффективны при решении задач, обладающих параллелизмом по данным, число арифметических операций в которых велико по сравнению с операциями над памятью. Например, в 3D-рендеринге параллелизм по данным выражается в распределении по потокам обработки отдельных вершин. Аналогично, обработка изображений, кодирование и декодирование видео и распознавание образов легко делятся на подзадачи над блоками изображений и пикселов. Более того, множество задач, не связанных с графикой, также обладают параллелизмом по данным: обработка сигналов, физика, финансовый анализ, вычислительная биология и т.д.

3.2. Программная модель CUDA. Среда выполнения CUDA

Compute Unified Device Architecture (CUDA) – это программная модель, включающая описание вычислительного параллелизма и иерархичной структуры памяти непосредственно в язык программирования. С точки зрения программного обеспечения, реализация CUDA представляет собой кроссплатформенную систему компиляции и исполнения программ, части которых работают на CPU и GPU. CUDA предназначена для разработки GPGPU-приложений без привязки к графическим API и поддерживается всеми GPU NVIDIA, начиная с серии GeForce 8.

Основные принципы

Концепция CUDA отводит GPU роль массивно-параллельного сопроцессора. В литературе о CUDA основная система, к которой подключен GPU, для краткости называется термином хост (host), аналогично сам GPU по отношению к хосту часто называется просто устройством (device). CUDA-программа задействует как CPU, так и GPU, на CPU выполняется последовательная часть кода и подготовительные стадии для GPU-вычислений. Параллельные участки кода могут быть перенесены на GPU, где будут одновременно выполняться большим множеством нитей (threads). Важно отметить ряд принципиальных различий между обычными потоками CPU и нитями GPU:

- нить GPU чрезвычайно легковесна, ее контекст минимален, регистры распределены заранее;
- для эффективного использования ресурсов GPU программе необходимо задействовать тысячи отдельных нитей, в то время как на многоядерном CPU максимальная эффективность обычно достигается при числе потоков, равном или в несколько раз большем количества ядер.

В целом работа нитей на GPU соответствует принципу SIMD, однако есть существенное различие. Только нити в пределах одной группы (для GPU архитектуры Fermi – 32 нити), называемой варпом (warp) выполняются физически одновременно. Нити различных варпов могут находиться на разных стадиях выполнения программы. Такой метод обработки данных обозначается термином SIMT (Single Instruction – Multiple Threads). Управление работой варпов производится на аппаратном уровне.

По ряду возможностей новых версий CUDA прослеживается тенденция к постепенному превращению GPU в самодостаточное устройство, полностью замещающее обычный CPU за счет реализации некоторых системных вызовов (в терминологии GPU системными вызовами являются, например, malloc и free, реализованные в

CUDA 3.2) и добавления облегченного энергоэффективного CPU-ядра в сам GPU (архитектура Maxwell).

Важным преимуществом CUDA является использование для программирования GPU языков высокого уровня. В настоящее время существуют компиляторы C++ и Fortran. Эти языки расширяются небольшим множеством новых конструкций: атрибуты функций и переменных, встроенные переменные и типы данных, оператор запуска ядра.

Нити и блоки

Рассмотрим пример CUDA-программы из прил. 1, использующей GPU для поэлементного сложения двух одномерных массивов**.**

Функция (или процедура в случае Fortran) sum kernel является ядром (атрибут global или global) и будет выполнять на GPU по одной независимой нити для каждого набора элементов a[i], b[i] и c[i]. Нить GPU имеет координаты во вложенных трехмерных декартовых равномерных сетках «индексы блоков» и «индексы потоков внутри каждого блока», двумерный случай показан на рис. 3.1 [1]. В контексте каждой нити значения координат и размерностей доступны через встроенные переменные threadIdx, blockIdx и blockDim, gridDim соответственно. Если проводить аналогию с Message Passing Interface (MPI), то значения этих переменных по смыслу аналогичны результатам функций MPI Comm rank и MPI Comm size.

Код ядра sum kernel начинается с определения глобального индекса массива idx, зависящего от координат нити. В общем случае соответствие нитей и частей задачи может быть любым, например, одна нить может обрабатывать не один элемент массива, а определенный диапазон. В течение времени работы ядра декартовы сетки нитей и блоков зафиксированы для отображения на аппаратный уровень вычислительных элементов GPU. Каждая нить производит сложение элементов массивов a и b и помещает результат в элемент массива c. После этого нить завершает работу.

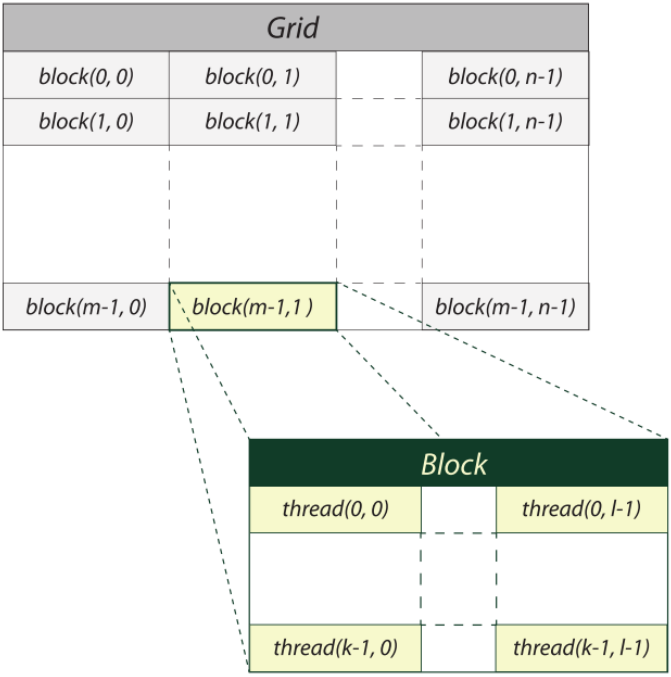


Рис. 3.1. Иерархия нитей в CUDA

При использовании глобальной памяти, расположенной физически на самом GPU, управлять ею можно с хоста. В функции sum host память, выделяемая на GPU, заполняется копией данных из памяти хоста, затем производится запуск ядра sum kernel, синхронизация и копирование результатов обратно в память хоста. В конце производится высвобождение ранее выделенной глобальной памяти GPU.

Такая последовательность действий характерна для любого CUDA-приложения. Общим приемом программирования для CUDA является группировка множества нитей в блоки. На это есть две причины. Во-первых, далеко не для каждого параллельного алгоритма существует эффективная реализация на полностью независимых нитях: результат одной нити может зависеть от результата некоторых других, исходные данные нити могут частично совпадать с данными соседних. Во-вторых, размер одной выборки данных из глобальной

памяти намного больше размера вещественного или целочисленного типа, т.е. одна выборка может покрыть запросы группы из нескольких нитей, работающих с подряд идущими в памяти элементами. В результате группировки нитей исходная задача распадается на независимые друг от друга подзадачи (блоки нитей) с возможностью взаимодействия нитей в рамках одного блока и объединения запросов в память в рамках одного варпа (рис. 3.2). Разбиение нитей на варпы также происходит отдельно для каждого блока. Объединение в блоки является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и возможностью сделать соответствующую аппаратную логику эффективной и дешевой.

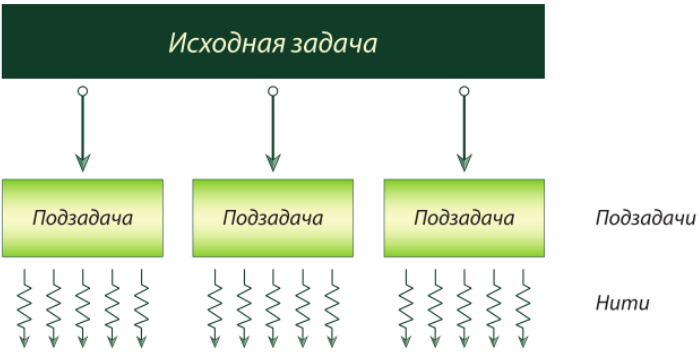


Рис. 3.2. Разбиение исходной задачи на набор независимо управляемых подзадач

На время выполнения ядра каждый блок получает в распоряжение часть быстрой разделяемой памяти, которую могут совместно использовать все нити блока. Поскольку не все нити блока выполняются физически одновременно, то для их взаимного согласования необходим механизм синхронизации. Для этой цели в CUDA предусмотрен вызов syncthreads(), который блокирует дальнейшее исполнение кода ядра до тех пор, пока все нити блока не войдут в эту функцию (рис. 3.3).

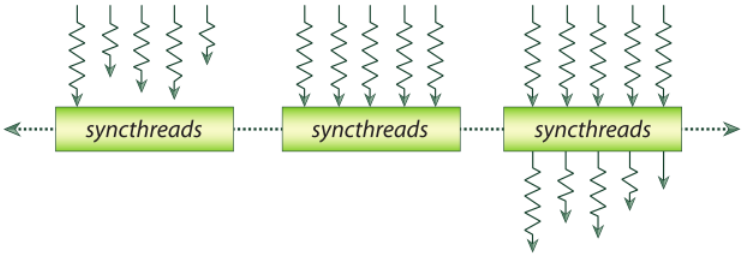


Рис. 3.3. Барьерная синхронизация

Расширение языка

Расширение языка:

- атрибуты функций – показывают, где будет выполняться функция и откуда она может быть вызвана;
- атрибуты переменных – задают тип используемой памяти;
- оператор запуска ядра – определяет иерархию нитей, очередь команд (CUDA Stream) и размер разделяемой памяти;
- встроенные переменные – содержат контекстную информацию относительно текущей нити;
- дополнительные типы данных – определяют несколько новых векторных типов.

В настоящее время существуют два компилятора с поддержкой CUDA: реализация CUDA для языка C++ компании NVIDIA, основанная на open-source компиляторе Open64 и реализация CUDA для Fortran компании Portland Group Inc. (PGI) с закрытой лицензией. Стандартное расширение имен исходных файлов – .cu или .cuf (.CUF – с проходом через препроцессор) соответственно. В случае если CPU-программа также написана на C++ или Fortran, части кода для CPU и GPU могут объединяться в общие модули компиляции.

Атрибуты функций и переменных

В CUDA используются следующие атрибуты функций:

Атрибуты host (host) и device (device) могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и на CPU – соответствующий код для обеих

платформ будет автоматически сгенерирован компилятором). Атрибуты global и host не могут быть использованы вместе. Атрибут global обозначает ядро, и соответствующая функция CUDA C должна возвращать значение типа void, CUDA Fortran – быть процедурой. На функции, выполняемые на GPU (device и global), накладываются следующие ограничения:

- не поддерживаются static-переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Таблица 3.1. Атрибуты функций и переменных в CUDA

Атрибут C	Функция выполняется на	Функция может вызываться из
__device__	device(GPU)	device(GPU)
__global__	device(GPU)	host(CPU)
__host__	host(CPU)	host(CPU)

Для задания способа размещения переменных в памяти GPU используются следующие атрибуты: device, constant и shared. На их использование также накладывается ряд ограничений:

- атрибуты не могут быть применены к полям структуры (struct или union);
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;
- запись в переменные типа constant может осуществляться только CPU при помощи специальных функций;
- shared переменные не могут инициализироваться при объявлении.

Пока существующие компиляторы для CUDA не реализуют поддержку модульной сборки GPU-кода, каждая global -функция должна находиться в одном исходном файле вместе со всеми device -функциями и переменными, которая она использует.

Встроенные типы

В язык добавлены 1/2/3/4-мерные векторные типы на основе char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long,

longlong, float и double. Имя векторного типа формируется из базового имени и числа элементов, например, float4.

Компоненты векторных типов имеют имена x, y, z и w. Для создания значений векторов заданного типа служит конструкция вида make <typeName>:

```
int2 a=make_int2(1,7); // Создает вектор (1,7).
float3 u=make_float3(1,2,3.4f); //Создает вектор
(1.0f,2.0f,3.4f).
```

В отличие от шейдерных языков GLSL, Cg и HLSL, для этих типов не поддерживаются векторные покомпонентные операции, т.е. нельзя просто сложить два вектора при помощи оператора «+», это необходимо делать отдельно для каждой компоненты.

Также добавлен тип dim3, используемый для задания размерностей сеток нитей и блоков. Этот тип основан на типе uint3, и обладает нормальным конструктором, по умолчанию инициализирующим компоненты единицами.

```
dim3 blocks(16,16);// Эквивалентно blocks(16,16,1)
dim3 grid (256);// Эквивалентно grid(256,1,1).
```

Встроенные переменные

В язык добавлены следующие специальные переменные:

- gridDim – размер сетки (имеет тип dim3);
- blockDim – размер блока (имеет тип dim3);
- blockIdx – индекс текущего блока в сетке (имеет тип uint3);
- threadIdx – индекс текущей нити в блоке (имеет тип uint3);
- warpSize – размер варпа (имеет тип int).

Оператор вызова GPU-ядра

Для запуска ядра на GPU используется следующая конструкция:

```
kernel_name <<<Dg , Db , Ns , S>>> ( args ) ;
```

Здесь *kernel name* – это имя или адрес соответствующей global - функции. Параметр *Dg* типа *dim3* задает размерности сетки блоков (число блоков в сетке блоков), параметр *Db* типа *dim3* задает размерности блока нитей (число нитей в блоке). Необязательный параметр Ns типа size t задает дополнительный объем разделяемой памяти в

байтах (по умолчанию – 0), которая должна быть динамически выделена каждому блоку (в дополнение к статически выделенной). Параметр S типа cudaStream t ставит вызов ядра в определенную очередь команд (CUDA Stream), по умолчанию – 0. Вызов функции-ядра также может иметь произвольное фиксированное число параметров, суммарный размер которых не превышает 4 Кбайт. Следующий пример запускает ядро my kernel параллельно на n нитях, используя одномерный массив из двумерных блоков нитей 16 x 16, и передает на вход ядру два параметра – a и n. При этом каждому блоку дополнительно выделяется 512 байт разделяемой памяти и запуск ядра производится в очереди команд my stream.

```
my_kernel<<<dim3(n/256),dim3(16,16),512,
my_stream>>>(a,n);
```

Встроенные функции

Для CUDA реализованы математические функции, совместимые с ISO C. Также имеются соответствующие аналоги, вычисляющие результат с пониженной точностью, например, *sinf* для *sinf*.

Приложение 1 содержит пример программы CUDA.

Подробнее с расширениями языка C можно ознакомиться в книгах [1] и [2].

3.3. Иерархия памяти CUDA

GPU и CPU существенно отличаются организацией систем памяти и методами работы с ней [3-4]. В CPU большую долю площади схемы обычно занимают кэши различных уровней. Основная же часть GPU занята вычислительными блоками. Кроме того, CPU обычно не предоставляет прямой доступ к управлению кэшами, ограничиваясь в основном лишь инструкцией prefetch. В GPU также присутствует не контролируемый явно кэш, однако существует еще несколько видов памяти, которые всегда должны управляться программно (таблица 3.2). Одни виды памяти расположены непосредственно

в каждом из потоковых мультипроцессоров, другие – размещены в DRAM. В умелом использовании различных видов памяти состоит одновременно и сложность программирования для CUDA, и значительный потенциал эффективности.

Таблица 3.2. Типы памяти в CUDA

Тип памяти	Расположение	Кэш	Доступ	Видимость	Время жизни
Регистры	Мультипроц.	Нет	R/W	Нить	Нить
Локальная	DRAM GPU	Нет	R/W	Нить	Нить
Разделяемая	Мультипроц.	Нет	R/W	Блок	Блок
Глобальная	DRAM GPU	Есть	R/W		
Константная	DRAM GPU	Есть	R/O		
Текстурная	DRAM GPU	Есть	R/O		
UVA	Host DRAM	Нет	R/W		

Наиболее простым видом памяти является регистровый файл (или просто регистры). Каждый потоковый мультипроцессор в GPU в различных версиях графических процессоров (compute capability от 1.0 до 7.0), содержит от 8 192 до 131 072 32-битных регистров соответственно. Регистры распределяются между нитями блока на этапе компиляции, что влияет на максимальное количество блоков, которые может выполнять один мультипроцессор. Каждая нить получает в монопольное использование для чтения и записи некоторое количество регистров на все время исполнения ядра. Доступ к регистрам других нитей запрещен. Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, они обладают минимальной латентностью.

Если регистров не хватает, то для размещения локальных данных нити дополнительно используется локальная память. Так как она размещена в DRAM, то латентность доступа к ней велика: 400–800 тактов. Также в локальную память всегда попадают союзы (unions), а также большие или нетривиально индексируемые автоматические массивы.

Важным типом памяти в CUDA является разделяемая память (shared memory). Эта память расположена непосредственно в потоковом мультипроцессоре, но в отличие от регистров, выделяется не на уровне нитей, а на уровне блоков. Каждый блок получает в свое распоряжение одно и то же количество разделяемой памяти. Всего каждый мультипроцессор устройств поколений содержит от 16 до 112 Кб разделяемой памяти. От размеров разделяемой памяти, требуемой каждому блоку, зависит максимальное количество блоков, которое может быть запущено на одном мультипроцессоре. Разделяемая память обладает очень небольшой латентностью, сравнимой со временем доступа к регистрам и может быть использована всеми нитями блока для чтения и записи.

Глобальная память – это обычная память DRAM, расположенная на плате GPU или разделяемая с памятью мобильного устройства. Начиная с архитектуры Fermi, глобальная память кэшируется. Влияние присутствия кэша тем не менее не стоит переоценивать: в отличие от CPU, его размер в пересчете на одну нить очень мал. Глобальная память может выделяться как с CPU функциями CUDA API, так и нитями CUDA-ядра с помощью вызова malloc. Все нити ядра могут читать и писать в глобальную память. Как и локальная память, глобальная обладает

высокой латентностью. Минимизация доступа к глобальной памяти – это основной метод получения высокоэффективных CUDA-приложений.

Константная и текстурная память также расположены в DRAM, но обладают независимым кэшем, обеспечивающим высокую скорость доступа. Оба типа памяти доступны всем нитям GPU, но только на чтение. Запись в них может производить CPU при помощи функций CUDA API. Общий объем константной памяти ограничен 64 Кб. Текстурная память также предоставляет специальные возможности по работе с текстурами.

Следует обратить внимание на то, что любой CUDA-вызов, связанный с выделением и освобождением памяти на GPU, всегда является синхронным и поэтому будет выполнен только после завершения всех активных асинхронных вызовов.

Глобальная память

Основную часть DRAM GPU занимает глобальная память. При корректной работе ядер динамически выделенная глобальная память сохраняет целостность данных на протяжении всего времени жизни приложения, что, в частности, позволяет использовать ее как основное хранилище для передачи данных между несколькими ядрами.

Глобальная память может быть выделена как статически, так и динамически. Динамическое выделение и освобождение глобальной памяти в коде хоста производится при помощи следующих вызовов:

```
cudaError_t cudaMalloc (void ** devPtr , size_t size ) ;
cudaError_t cudaFree (void * devPtr ) ;
cudaError_t cudaMallocPitch (void ** devPtr , size_t * pitch
, size_t width , size_t height );
```

С появлением device-функций malloc и free в CUDA 3.2, на GPU с compute capability 2.0 и выше динамическое выделение и освобождение глобальной памяти возможно не только в хост-коде но и в коде CUDA-ядра. Однако при этом динамическое выделение происходит лишь относительно нитей ядра, тогда как общий пул памяти под эти аллокации выделяется заранее. Размер пула может быть установлен вызовом функции cudaLimitMallocHeapSize или по умолчанию равен 8 Мб. В следующем примере два CUDA-ядра выделяют и освобождают глобальную память GPU:

```
#include <stdio.h>
#include <stdlib.h>
__global__ void mass_malloc(void** ptrs, size_t size)
{
  ptrs[threadIdx.x] = malloc(size);
}

__global__ void mass_free(void** ptrs)
{
  free(ptrs[threadIdx.x]);
}

int main(int argc, char* argv[])
{
  if (argc != 3)
  {
    printf("Usage: %s <nthreads> <size>\n", argv[0]);
    return 0;
  }

  int nthreads = atoi(argv[1]);
  size_t size = atoi(argv[2]);

  // Set a heap size of 16 megabytes. Note that this must
  // be done before any kernel is launched.
  cudaDeviceSetLimit(cudaLimitMallocHeapSize, 16 * 1024 *
1024);

  void** ptrs; cudaMallocHost(&ptrs, sizeof(void*) *
nthreads);
  memset(ptrs, 0, sizeof(void*) * nthreads);
  mass_malloc<<<1, nthreads>>>(ptrs, size);
  cudaDeviceSynchronize();
```

```
  for (int i = 0; i < nthreads; i++)
    printf("Thread %d got pointer: %p\n", i, ptrs[i]);

  mass_free<<<1, nthreads>>>(ptrs);
  cudaDeviceSynchronize();
  cudaFreeHost(ptrs);

  return 0;
}
```

```
$ ./device_malloc_test 8 24
Thread 0 got pointer: 0x2013ffb20
Thread 1 got pointer: 0x2013ffb70
Thread 2 got pointer: 0x2013ffbc0
Thread 3 got pointer: 0x2013ffc10
Thread 4 got pointer: 0x2013ffc60
Thread 5 got pointer: 0x2013ffcb0
Thread 6 got pointer: 0x2013ffd00
Thread 7 got pointer: 0x2013ffd50
```

Глобальная переменная может быть объявлена статически при помощи атрибута device . В этом случае память будет выделена при инициализации модуля с CUDA-ядрами и данными (в зависимости от способа загрузки, в момент старта приложения или при вызове cuModuleLoad). В следующем примере утилита nm показывает, что в объектном представлении cubin (двоичный файл CUDA-модуля), глобальные данные размещаются таким же образом как на CPU: “B” – неинициализированная переменная, “D” – инициализированная переменная, “R” – только для чтения. Отличие же состоит в том, что в cubin статические переменные не становятся приватными (соответствующие литеры в нижнем регистре) и не декорируются. Правила видимости не имеют смысла, поскольку в CUDA отсутствует линковка, и все CUDA-объекты должны быть самодостаточными, без внешних зависимостей.

```
__device__ float val1;
__device__ float val2 = 1.0f;
__device__ __const__ float val3 = 2.0f;
__constant__ float val_pi = 3.14;
__device__ static float val4;
```

```
__global__ void kernel(float* val5)
{
   if (!threadIdx.x && !blockIdx.x)
   {
        val4 = sin(val1+val2*val3+val_pi);
       *val5 = val4;
   }
}
```

```
$ nvcc -keep -c test.cu
$ nm test.sm_10.cubin / grep val
0000000000000004 B val1
0000000000000004 D val2
0000000000000000 D val3
0000000000000000 B val4
0000000000000018 R val_pi
```

```
$ g++ -c test.c
$ nm test.o | grep val
0000000000000000 r _ZL4val3
0000000000000004 b _ZL4val4
0000000000000000 B val1
0000000000000000 D val2
0000000000000004 D val_pi
```

```
float val1;
float val2 = 1.0f;
const float val3 = 2.0f;
float val_pi = 3.14;
static float val4;

#include <cmath>

void kernel(float* val5)
{
  val4 = sin(val1+val2*val3+val_pi);
  *val5 = val4;
}
```

```
$ g++ -c test . c
```

```
$ nm test. o |   grep val
0000000000000000 r _ZL4val3
0000000000000004 b _ZL4val4
0000000000000000 B val1
0000000000000000 D val2
0000000000000004 D val_pi
```

Доступ к многомерным массивам в глобальной памяти может быть более эффективен при наличии выравнивания строк. Выравнивание обеспечивается добавлением фиктивных элементов в конец каждой строки и соответствующих сдвигов при индексации. Функция cudaMallocPitch выделяет память с выравниванием строк и возвращает сдвиг через параметр pitch. Например, если выделена память для матрицы с элементами типа T, то для получения адреса элемента, расположенного в строке row и столбце col, используется следующая формула:

Хотя функция cudaMalloc возвращает обычный указатель, его значение имеет смысл только для адресного пространства GPU. Для заполнения памяти GPU данными хоста и наоборот, необходимо использовать специальные функции копирования:

```
cudaError_t cudaMemcpy ( void * dst, const void * src, size_t
size, enum cudaMemcpyKind kind ) ;

cudaError_t cudaMemcpyAsync ( void * dst, const void * src,
size_t size, enum cudaMemcpyKind kind, cudaStream_t stream );

cudaError_t cudaMemcpy2D ( void * dst, size_t dpitch, const
void * src, size_t spitch, size_t width, size_t height, enum
cudaMemcpyKind kind );

cudaError_t cudaMemcpy2DAsync ( void * dst, size_t dpitch,
const void * src, size_t spitch, size_t width, size_t height,
enum cudaMemcpyKind kind, cudaStream_t stream );
```

Копирование данных в глобальной памяти внутри одного GPU обычно производится на порядок быстрее, чем внутри памяти хоста,

например, для GPU Tesla C2050 характерная скорость – 144 Гбайт/сек. Копирование данных между памятью хоста и памятью GPU значительно медленнее. Наиболее распространенный в данный момент стандарт интерфейса PCI Express 2.0 способен обеспечить пропускную способность до 8 Гбайт/сек. С учетом потерь на кодирование, латентности и задержки, на практике, как правило, удается добиться не более 4 Гбайт/сек при копировании между хостом и одним GPU и не более 6 Гбайт/сек – при копировании между хостом и несколькими GPU одновременно. Наилучшая скорость копирования данных между хостом и GPU может быть достигнута при использовании pinned-памяти. Pinned-память – это память хоста, которая может быть либо выделена функциями cudaHostAlloc или cudaMallocHost, заменяющими стандартный вызовы malloc или new, либо получена переводом обычной памяти в категорию pinned функцией cudaHostRegister (см. Unified Virtual Address Space).

```
cudaError_t cudaHostAlloc ( void** pHost, size_t size, un-
signed int flags );
cudaError_t cudaMallocHost ( void** devPtr, size_t size );
cudaError_t cudaFreeHost ( void* devPtr );
cudaError_t cudaHostRegister ( void* ptr, size_t size, un-
signed int flags );
cudaError_t cudaHostUnregister ( void* ptr );
```

При асинхронном копировании памяти между CPU и GPU с помощью функции cudaMemcpyAsync в CUDA версии 4.0 должна использоваться только pinnedпамять. Начиная с CUDA версии 4.0 может использоваться обычная (не pinned) память, но в этом случае вызов cudaMemcpyAsync будет синхронным. Выделение большого количества pinned-памяти может отрицательно сказаться на быстродействии всей системы.

В CUDA используется прямая адресация глобальной памяти GPU, т.е., в отличие от OpenCL, для хранения адресов подходят обычные указатели, и для них корректна адресная арифметика. Если память

выделяется с помощью cudaMalloc, то выделенный диапазон имеет смысл только в контексте GPU-ядра. Память, выделенная на одном GPU некорректна по отношению к другому GPU (см. контексты GPU). Если память выделяется cudaHostAlloc(..., , cudaHostAllocMapped), то выделенный на CPU диапазон памяти становится доступен как для CPU, так и для всех GPU (см. Unified Virtual Address Space).

Разделяемая память

Разделяемая память размещена непосредственно в каждом мультипроцессоре и доступна для чтения и записи всем нитям блока. Ее наличие отличает CUDA от традиционных графических API. Разделяемая память может существенно улучшить производительность GPU-приложения в случае, если ее удается использовать как буфер, заменяющий обращения к глобальной памяти. Всего каждому мультипроцессору устройства с compute capability 2.x может быть доступно 16 или 48 Кбайт разделяемой памяти в зависимости от размера L1-кэша, который может быть настроен программно. Объем разделяемой памяти делится поровну между все и блоками нитей, запущенными на мультипроцессоре. Разделяемая память также используется для передачи значений аргументов ядра.

Размер разделяемой памяти может быть задан в CUDA-ядре при определении массивов с атрибутом shared или в параметрах запуска ядра. В последнем случае размеры используемых shared -массивов могут не указываться. Если таких массивов несколько, то все они будут указывать на начало выделенной блоку дополнительной разделяемой памяти, т.е. если они должны следовать друг за другом, то потребуется явно указывать смещение:

```
__global__ void kernel1(float* a)
{
    // Явно задано выделить 256*4 байт на блок.
    __shared__ float buf [256];
```

```
    // Запись значения из глобальной памяти в разделяемую.
    buf [threadIdx.x] = a [blockIdx.x * blockDim.x +
threadIdx.x];

    ...
}

__global__ void kernel2(float* a)
{
    // Размер явно не указан.
    __shared__ float buf [];

    // Запись значения из глобальной памяти в разделяемую.
      buf [threadIdx.x] = a [blockIdx.x * blockDim.x +
threadIdx.x];

    ...
}

// Запустить ядро и задать выделяемый (под buf)
// объем разделяемой памяти в байтах.
kernel2<<<dim3(n / 256), dim3(256), k * sizeof(float)>>> ( a
);

__global__ void kernel3(float* a, int k)
{
    // Размер явно не указан.
    __shared__ float buf1 [];

    // Размер явно не указан, считаем, что он передан как k.
    __shared__ float buf2 [];

      buf1 [threadIdx.x]     = a [blockIdx.x * blockDim.x +
threadIdx.x];
      buf2 [k + threadIdx.x] = a [blockIdx.x * blockDim.x +
threadIdx.x + k];
```

В CUDA-программе нельзя гарантировать, что операция, только что завершенная в текущей нити, уже выполнена и в нитях других варпов. По этой причине любая коллективная операция с разделяемой памятью, после которой нить будет использовать значения, измененные другими нитями, должна завершаться барьерной синхронизацией – syncthreads(). Многие приложения используют следующую последовательность действий:

• загрузить необходимые данные в shared-память из глобальной памяти;

• syncthreads ();

• выполнить вычисления над загруженными данными;

• syncthreads ();

• записать результат в глобальную память.

Константная память

Константная память – кэширумая область DRAM размером 64 Кбайт, доступная с GPU только для чтения и для чтения и записи с хоста при помощи следующих функций:

```
cudaError_t cudaMemcpyToSymbol (const char * symbol, const
void * src, size_t count, size_t offset, enum cudaMemcpyKind
kind);

cudaError_t cudaMemcpyFromSymbol (void * dst, const char *
symbol, size_t count, size_t offset, enum cudaMemcpyKind
kind);

cudaError_t cudaMemcpyToSymbolAsync (const char * symbol,
const void * src, size_t count, size_t offset, enum cudaMem-
cpyKind kind, cudaStream_t stream);

cudaError_t cudaMemcpyFromSymbolAsync (void * dst, const char
* symbol, size_t count, size_t offset, enum cudaMemcpyKind
kind, cudaStream_t stream);
```

Параметр kind задает направление операции копирования и может принимать одно из следующих значений: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost и cudaMemcpyDeviceToDevice. Параметр stream позволяет организовать несколько асинхронных потоков команд. По умолчанию используется синхронный режим, соответствующий нулевому потоку.

Константная память может быть выделена только путем статического объявления в коде программы с добавлением атрибута constant . В следующем фрагменте кода массив в константной памяти заполняется данными из DRAM хоста:

```
__constant__ float contsData [ 2 5 6 ] ; // константная память
GPU
float hostData [ 2 5 6 ] ; // данные в памяти CPU
. . .
// Скопировать данные из памяти CPU в константную память GPU
cudaMemcpyToSymbol ( constData, hostData, sizeof( data ), 0,
cudaMemcpyHostToDevice ) ;
```

Поскольку константная память кэшируется, то она подходит для размещения небольшого объема часто используемых неизменяемых данных, которые должны быть доступны всем нитям. Дополнительно в устройствах с compute capability 2.x доступно аналогичное константной памяти кэширование произвольного участка глобальной памяти (инструкция LDU или LoaD Uniform). Данный режим будет автоматически активирован при запросе только для чтения по адресу памяти, не зависящему от индекса нити.

```
__global__ void kernel ( float *g_dst, const float *g_src )
{
  g_dst = g_src [ 0 ] ; // не зависит от индекса нити ->
uniform load
  g_dst = g_src [ blockIdx . x ] ; // не зависит от индекса
нити -> uniform load
  g_dst = g_src [ threadIdx . x ] ; // зависит от индекса ->
non-uniform
}
```

Текстурная память

Текстурная память и аппаратные схемы интерполяции объединены на GPU в текстурные блоки, которые используются в графических задачах для заполнения треугольников двумерными изображениями – текстурами.

В зависимости от архитектуры, каждому мультипроцессору может быть доступно различное количество текстурных блоков. В текстурном блоке аппаратно реализованы следующие функции:

• фильтрация текстурных координат;

• билинейная или точечная интерполяция;

• разумное возвращаемое значение в случае, когда значения текстурных координат выходят за допустимые границы;

• обращение по нормализованным или целочисленным координатам;

• возвращение нормализованных значений;

• кэширование данных.

В общем случае текстура представляет собой простой и удобный интерфейс для работы с одномерными, двумерными и трехмерными массивами в режиме «только для чтения». Текстуры могут быть полезны на GPU с compute capability 1.x, если обеспечить объединение запросов в память не представляется возможным. С появлением L1- и L2-кэшей в устройствах архитектуры Fermi роль текстурной памяти снижается, поскольку она обладает меньшей скоростью, чем L1-кэш.

Текстурная память выделяется с помощью функции cudaMallocArray:

```
cudaError_t cudaMallocArray ( struct cudaArray **arrayPtr,
const struct cudaChannelFormatDesc *desc, size_t width, size_t
height );

cudaError_t cudaFreeArray ( struct cudaArray *array );
```

Однако в отличие от методов работы с глобальной памятью, arrayPtr является непрозрачным контейнером, управление которым осуществляет драйвер. Доступ к контейнеру из CUDA-ядра возможен только через специальные текстурные ссылки (texture reference), которые в графических API назывались сэмплерами (sampler).

Задача такой абстракции – отделить данные (cudaArray) и способ их хранения от интерфейса доступа к ним (texture reference). Для чтения cudaArray из CUDA-ядра необходимо сначала ассоциировать его с текстурной ссылкой:

```
cudaError_t cudaBindTextureToArray ( const struct textureRef-
erence *texref, const struct cudaArray *array, const struct
cudaChannelFormatDesc *desc );

cudaError_t cudaBindTextureToArray ( const struct texture<T,
dim, readMode> &tex, const struct cudaArray * array );
```

Первый вид вызова является низкоуровневым и требует задания переменной textureReference и дескриптора канала вручную:

```
textureReference texref;
texref.addressMode[ 0 ] = cudaAddressModeWrap;
texref.addressMode[ 1 ] = cudaAddressModeWrap;
texref.addressMode[ 2 ] = cudaAddressModeWrap;
texref.channelDesc = cudaCreateChannelDesc<uchar4>( );
texref.filterMode = cudaFilterModeLinear;
texref.normalized = cudaReadModeElementType;
cudaChannelFormatDesc desc = cudaCreateChannelDesc<uchar4>();
```

Второй тип вызова использует шаблоны для задания текстурных ссылок и наследует дескриптор канала от переданного cudaArray:

```
texture<uchar4, 2, cudaReadModeElementType> texName;
```

Чтение текстуры из ядра производится функциями tex1D(), tex2D(), tex3D(), которые принимают текстурную ссылку и одну, две или три координаты. В случае, если выбран ненормализованный режим обращения к текстуре, стоит учитывать, что для точного попадания в центр пиксела, необходимо добавлять смещение, равное половине пиксела (т.е. равное 0:5f или 0:5f / ширину, 0:5f / высоту при нормализованных координатах):

```
uchar4 a = tex2D ( texName, texcoord.x + 0.5f, texcoord.y +
0.5f );
```

Текстурные ссылки можно получить, зная имя текстурного шаблона, с помощью следующей функции:

```
const textureReference* pTexRef = NULL;
cudaGetTextureReference( &pTexRef, "texName" );
```

Кроме cudaArray, привязать к текстурной ссылке можно и обычную линейную память.

```
cudaError_t cudaBindTexture ( size_t * offset, const struct
texture<T, dim, readMode> &tex, const void * dev_ptr, size_t
size );
```

```
cudaError_t cudaBindTexture2D ( size_t * offset, const struct
texture<T, dim, readMode> &tex, const void * dev_ptr, const
struct cudaChannelFormatDesc *desc, size_t width, size_t
height, size_t pitch_in_bytes );
```

Основным полезным свойством текстуры является возможность кэширования данных в двумерном измерении.

Список литературы

1. Перепёлкин Е.Е. Вычисления на графических процессорах (GPU) в задачах математической и теоретической физики / Е.Е. Перепёлкин, Б.И. Садовников, Н.Г. Иноземцева. – М., 2014. – 176 с.

2. Боресков, А.В. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: учеб. пособие / А.В. Боресков. – М.: Изд-во Московского университета, 2012. – 336 с.

3. Боресков, А. В. Основы работы с технологией CUDA / А.В. Боресков. – ДМК-Пресс Москва, 2010. – 231 с.

4. Джейсон Сандерс. Технология CUDA в примерах / Джейсон Сандерс, Эдвард Кэндрот. – ДМК Пресс, 2011. – 232 с.

```
/ The kernel executes as a large number of threads in parallel.
__global__ void sum_kernel ( float * a, float * b, float * c
)
{
    // Global thread index.
    int idx = threadIdx.x+blockIdx.x * blockDim.x;
    //Process the data corresponding to this thread.
    c [idx] = a [idx] + b [idx];
}
#include <stdio.h>
int sum_host(float *a, float *b, float *c, int n)
{
    int nb = n * sizeof ( float );
    float *aDev = NULL, *bDev = NULL, *cDev = NULL;
    // Allocate the GPU memory.
    cudaError_t cuerr = cudaMalloc ( (void**)&aDev, nb );
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot allocate GPU memory for aDev:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaMalloc ( (void**)&bDev, nb );
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot allocate GPU memory for bDev:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    cuerr = cudaMalloc ( (void**)&cDev, nb );
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot allocate GPU memory for cDev:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    // Set the launch configuration for n threads.
    dim3 threads = dim3(BLOCK_SIZE, 1);
    dim3 blocks  = dim3(n / BLOCK_SIZE, 1);
    // Copy input data from the CPU memory to the GPU memory.
    cuerr = cudaMemcpy ( aDev, a, nb, cudaMemcpyHostToDevice
);
    if (cuerr != cudaSuccess)
```

```
    {
        fprintf(stderr, "Cannot copy data from a to aDev:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;}
    cuerr = cudaMemcpy (bDev, b, nb, cudaMemcpyHostToDevice
);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy data from b to bDev:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    // Launch the kernel for data processing, using given
configuration.
    sum_kernel<<<blocks, threads>>> (aDev, bDev, cDev);
    cuerr = cudaGetLastError();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot launch CUDA kernel: %s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    // Wait for kernel to complete.
    cuerr = cudaDeviceSynchronize();
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot synchronize with the GPU:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    // Copy results to the CPU memory
    cuerr = cudaMemcpy ( c, cDev, nb, cudaMemcpyDeviceToHost
);
    if (cuerr != cudaSuccess)
    {
        fprintf(stderr, "Cannot copy data from cdev to c:
%s\n",
            cudaGetErrorString(cuerr));
        return 1;
    }
    // Free allocated GPU memory
    cudaFree ( aDev );
    cudaFree ( bDev );
    cudaFree ( cDev );
    return 0;}
```

```
! The kernel executes as a large number of threads in parallel.
attributes(global) subroutine sum_kernel (a, b, c)
    implicit none
    real, device, dimension(*) :: a, b, c
    integer :: idx
    ! Global thread index.
    idx = threadIdx%x + (blockIdx%x - 1) * blockDim%x
    ! Process the data corresponding to this thread.
    c (idx) = a (idx) + b (idx)
end
function sum_host (a, b, c, n )
    use cudafor
    implicit none
    real, dimension(n), intent(in) :: a, b
    real, dimension(n), intent(out) :: c
    integer, intent(in) :: n
    real, device, allocatable, dimension(:) :: aDev, bDev,
cDev
    integer :: sum_host, istat
    type(dim3) :: blocks, threads
    sum_host = 1
    istat = 0
    ! Allocate the GPU memory.
    allocate(aDev(n), stat = istat)
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for aDev: ',
&
            cudaGetErrorString(istat)
        return
    endif
    allocate(bDev(n), stat = istat)
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for bDev: ',
&
            cudaGetErrorString(istat)
        return
    endif
    allocate(cDev(n), stat = istat);
    if (istat .ne. 0) then
        write(*, *) 'Cannot allocate GPU memory for cDev: ',
&
            cudaGetErrorString(istat)
        return
    endif
    ! Set the launch configuration for n threads.
    threads = dim3(BLOCK_SIZE, 1, 1);
    blocks  = dim3(n / BLOCK_SIZE, 1, 1);
    ! Copy input data from the CPU memory to the GPU memory.
```

```
    aDev = a
    bDev = b
    ! Launch the kernel for data processing, using given con-
figuration.
    call sum_kernel<<<blocks, threads>>> (aDev, bDev, cDev);
    istat = cudaGetLastError();
    if (istat .ne. cudaSuccess) then
        write(*, *) 'Cannot launch CUDA kernel:',&
            cudaGetErrorString(istat)
        return
    endif
    ! Wait for the kernel to complete.
    istat = cudaThreadSynchronize();
    if (istat .ne. cudaSuccess) then
        write(*,*)'Cannot synchronize with theGPU',&
            cudaGetErrorString(istat)
        return
    endif
    ! Copy results to the CPU memory
    c = cDev
    ! Free allocated GPU memory
    deallocate(aDev)
    deallocate(bDev)
    deallocate(cDev)
    sum_host = 0
    return
end
```

Приложение 2

Текст программы Multiply.c

```
#include "mpi.h"
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{   int my_rank;
    int rank_size;
    MPI_Init(&argc, &argv); // Важно!
    MPI_Status status;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
    int nsize, msize;
    ifstream f_in_a (argv[1]);
    ifstream f_in_x (argv[2]);
    ofstream f_out_y (argv[3]);
    f_in_a >> nsize;
    f_in_a >> msize;
    double a[nsize][msize];
    double x[msize];
    double y[nsize];
    for (int i = 0; i < nsize; i++){
        for (int j = 0; j < msize; j++){
            f_in_a >> a[i][j];}}
    for (int j = 0; j < msize; j++){
        f_in_x >> x[j];}
    int start = (my_rank*nsize)/rank_size;
    int final = ((my_rank+1)*nsize)/rank_size;
    for (int i = start; i < final; i++){
        y[i] = 0.0;
        for (int j = 0; j < msize; j++){
            y[i] += a[i][j]*x[j];}}
    int tmp[2];
    if (my_rank%2 == 1){
        tmp[0] = start;
        tmp[1] = final;
        MPI_Send(&tmp,   2,   MPI_INT,   my_rank   -   1,   0,
MPI_COMM_WORLD );
       MPI_Send(&y[tmp[0]],   tmp[1]-   tmp[0],   MPI_DOUBLE,
my_rank - 1, 0, MPI_COMM_WORLD );}
    if (my_rank%2 == 0) {
        MPI_Recv(&tmp,   2,   MPI_INT,   my_rank   +   1,   0,
MPI_COMM_WORLD, &status );
       MPI_Recv(&y[tmp[0]],  tmp[1]  -  tmp[0],  MPI_DOUBLE,
my_rank + 1, 0, MPI_COMM_WORLD, &status );}
    if (my_rank%4 == 2){
```

```
        tmp[0] = start;
       MPI_Send(&tmp,   2,   MPI_INT,   my_rank   -   2,   0,
MPI_COMM_WORLD );
       MPI_Send(&y[start],  tmp[1]  -  tmp[0],  MPI_DOUBLE,
my_rank - 2, 0, MPI_COMM_WORLD );}
    if (my_rank%4 == 0){
        MPI_Recv(&tmp,   2,   MPI_INT,   my_rank   +   2,   0,
MPI_COMM_WORLD, &status );
       MPI_Recv(&y[tmp[0]],  tmp[1]  -  tmp[0],  MPI_DOUBLE,
my_rank + 2, 0, MPI_COMM_WORLD, &status );}
    if (my_rank%8 == 4){
        tmp[0] = start;
       MPI_Send(&tmp,   2,   MPI_INT,   my_rank   -   4,   0,
MPI_COMM_WORLD );
       MPI_Send(&y[start],  tmp[1]  -  tmp[0],  MPI_DOUBLE,
my_rank - 4, 0, MPI_COMM_WORLD );}
    if (my_rank%8 == 0){
        MPI_Recv(&tmp,   2,   MPI_INT,   my_rank   +   4,   0,
MPI_COMM_WORLD, &status );
       MPI_Recv(&y[tmp[0]],  tmp[1]  -  tmp[0],  MPI_DOUBLE,
my_rank + 4, 0, MPI_COMM_WORLD, &status );}
    if (my_rank == 0){
      for (int i = 0; i < nsize; i++){
           f_out_y << y[i] << endl;}}
    MPI_Finalize(); // Важно!
    return 0;
}
```

Учебное издание

Сагатов Евгений Собирович,
Якимов Павел Юрьевич

LINUX В СУПЕРКОМПЬЮТЕРНЫХ СИСТЕМАХ

Учебное пособие

В авторской редакции
Технический редактор А.В. Ярославцева
Компьютерная вёрстка А.В. Ярославцевой

Подписано в печать 17.10.2018. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 7,25.
Тираж 100 экз. Заказ .

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086 САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086 Самара, Московское шоссе, 34.