КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Н.Б. Плещинский, И.Н. Плещинский

МНОГОПРОЦЕССОРНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ КОМПЛЕКСЫ. ТЕХНОЛОГИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

КАЗАНЬ

2018

УДК 519.68

Публикуется по решению
учебно-методической комиссии
Института вычислительной математики
и информационных технологий,
по рекомендации кафедры прикладной математики

Рецензенты:

доктор пед. наук, профессор **Н.К. Нуриев**
кандидат физ.-мат..наук, доцент **Д.Н. Тумаков**

Плещинский Н.Б.
Многопроцессорные вычислительные комплексы. Технологии параллельного программирования: Учебное пособие / Н.Б. Плещинский, И.Н. Плещинский. – Казанский федеральный университет, 2018. – 80 с.

Изложены основные приемы параллельного программирования на языке C++ с использованием технологий OpenMP, MPI и CUDA.

Пособие предназначено для бакалавров и магистрантов, изучающих современные технологии программирования и их применение при решении задач вычислительной математики.

УДК 519.68

Предисловие

Учебное пособие написано по материалам курса лекций «Многопроцессорные вычислительные комплексы» для магистрантов Института вычислительной математики и информационных технологий Казанского федерального университета.

В первом разделе учебного пособия обсуждаются общие принципы работы многопроцессорных комплексов, особенности архитектуры МВК и различные подходы к их классификации. Особое внимание уделено приемам работы с командной строкой Windows и Linux с использованием свободно распространяемого программного обеспечения.

Во втором разделе рассмотрены возможности технологии параллельного программирования OpenMP.

Третий раздел посвящен технологии параллельного программирования MPI.

В четвертом разделе описана техника программирования для многоядерных графических процессоров с использованием технологии CUDA.

Введение

Если не удается решить вычислительную задачу на персональном компьютере за приемлемое время, то можно попытаться сделать это на многопроцессорном вычислительном комплексе (МВК). Круг задач, для решения которых используют МВК или МВС (многопроцессорные вычислительные системы), на сегодняшний день примерно следующий:

- моделирование физических процессов;
- долгосрочные прогнозы изменений климата и природных катаклизмов;
- разработка новых лекарств, расшифровка генома человека;
- обработка больших объемов данных (Big Data).

Первые МВК использовались в основном в научной сфере, сейчас же и в производстве, и в бизнесе.

Первые процессоры были относительно маломощными. Например, тактовая частота процессора 8086 фирмы Intel была всего лишь 4.77 Мгц. Современные процессоры значительно более совершенны, но повышать их производительность уже не получается. Хотя бы потому, что увеличение тактовой частоты сопровождается большим выделением тепла. Поэтому вполне естественно распределить работу между множеством процессоров, входящих в состав МВК.

Иногда мощные МВК называют суперкомпьютерами. Этот термин корректно использовать в тех случаях, когда нужно подчеркнуть, что речь идет о компьютерах, возможности которых существенно выше, чем у большинства других. Поэтому даже ноутбук

с многоядерным процессором можно рассматривать как МВК, но уж вряд ли как суперкомпьютер.

К основным элементам, из которых собираются вычислительные узлы МВК, относятся:

процессоры – вычислительные устройства, выполняющие операции над данными,

и блоки памяти, в которых данные хранятся. Ядра многоядерного процессора во многих случаях можно рассматривать как самостоятельные процессоры.

Узлы обмениваются информацией через линии связи (коммуникационные каналы).

На сегодняшний день вычисления проводятся существенно быстрее, чем передается информация по коммуникационным каналам. Это следует учитывать как при проектировании МВК, так и при разработке алгоритмов и программ.

Одна из самых серьезных проблем, стоящих перед конструкторами МВК – организация взаимодействия между вычислительными узлами и устройство самих узлов. Архитектура МВК – это система связей между узлами и элементами узлов. Для специализированных комплексов, ориентированных на решение определенного круга задач, архитектура определяется природой этих задач.

Программировать для МВК значительно труднее, чем для обычных компьютеров. При этом активно используются технологии параллельного программирования и, следовательно, специальное программное обеспечение. Идеи параллелизма в вычислениях используются давно. Еще на первых ЭВМ был реализован принцип одновременного выполнения операций над всеми разрядами чисел. С середины 80-х годов прошлого века используется параллелизм на уровне инструкций. Возникла задача: как изменить порядок команд в исполняемом коде и сформировать из них группы, которые можно

выполнять параллельно. Параллелизм на уровне данных предполагает, что в программе можно использовать векторные операции, их планирование – дело программиста. Параллелизм на уровне задач основан на том, что основная задача разбивается на подзадачи, которые могут выполняться в значительной степени независимо.

Итак, вот три направления, в которых должен ориентироваться программист:

архитектура МВК (hard),
программное обеспечение (soft) и
методы разработки параллельных алгоритмов (algo).

Первое и второе определяет та вычислительная техника, которая имеется. В третьем направлении можно совершенствоваться бесконечно. Ясно, что нужно всесторонне изучать доступные hard и soft. Но полезно также познакомиться с тем, что есть у соседей, и вообще быть в курсе новейших разработок.

Для оценки производительности компьютеров и, в частности, МВК используют простые показатели – количество операций, которые могут быть выполнены за единицу времени.

Эти величины имеют единицы измерения:
MIPS (Million instruction per second) – число команд за секунду и
Flops (Floating point operations per second) – число операций над вещественными числами за секунду.

Пиковая производительность является теоретической величиной, для большинства задач она никогда не достигается. Реальная производительность зависит от характера вычислительных задач. Для определения реальной производительности используют различные стандартные тесты. Самый известный из них – LINPAK. Этот тест проверяет, с какой скоростью может быть решена большая система линейных алгебраических уравнений с плотно заполненной

матрицей коэффициентов методом Гаусса с выбором ведущего элемента по строке. Есть и другие тесты. На основе теста LINPAK составляется список Top500 – список самых мощных суперкомпьютеров в мире.

Эффективность использования МВК – понятие не строгое. Наиболее часто под этим понимают ускорение: во сколько раз быстрее стала выполняться программа. Требуемые ресурсы памяти в большинстве случаев не столь существенны.

Важным параметром при использовании МВК является отношение

$$\frac{производительность}{стоимость}.$$

В знаменателе этой дроби нужно учитывать не только стоимость оборудования, но и затраты на его эксплуатацию – техническое обслуживание и ремонт, потребление электроэнергии, обеспечение температурного режима и так далее. Но следует помнить о том, что разработка параллельных программ также может потребовать значительных затрат.

Не любая задача может быть распараллелена. Простой пример – решение некоторого уравнения таким итерационным методом, когда каждое новое уточненное значение решения вычисляется по значению, полученному на предыдущем шаге.

В учебной литературе по параллельному программированию часто приводят закон Амдала (Amdahl's law): ускорение вычислений на нескольких процессорах

$$S = \frac{1}{\alpha + (1-\alpha)/p},$$

где α – доля последовательного кода, p – число процессоров. Эта формула не очень точная, но позволяет кое-что понять.

Проверим. Если $\alpha = 0.1$ и $p = 10$, то $S \approx 5$. Так что если только 10% алгоритма не удалось распараллелить, то на десяти процессорах удается ускориться всего лишь в пять раз. При этом еще совершенно не учитывались задержки при передаче данных от процессора к процессору.

Из опыта авторов: два компьютера, стоящих в соседних комнатах, соединили двумя проводами через последовательные порты. Небольшие текстовые файлы было быстрее сбросить на дискету и отнести дискету на другую машину, чем дождаться, когда они перекачаются по такому примитивному коммуникационному каналу.

Еще один пример. Для процессора Pentium III, тактовая частота порядка 1 Ггц, скорость обмена данными с кэшем для команд (L1) примерно 10000 Мб/сек и с кэшем для данных (L2) примерно 4500 Мб/сек. Но обмен данными с основной память проходит существенно медленнее, со скоростью всего порядка 250 Мб/сек.

Таким образом, успешное использование МВК предполагает, что задача и возможности комплекса соответствуют друг другу.

Архитектура и классификация МВК

Предпринимались и предпринимаются различные попытки классификации МВК.

Классификация Флинна (M. Flynn, 1966) основана на таких понятиях, как потоки данных и потоки команд. Выделены четыре класса вычислительных устройств.

SISD = Single Instruction Single Data

К этому классу относятся последовательные вычислительные устройства с одним процессором.

MISD = Multiple Instruction Single Data

Реальных устройств такого типа пока нет. Трудно представить, как одновременно можно выполнить несколько операций над одними и теми же данными.

SIMD = Single Instruction Multiple Data

Поток команд один, а потоков данных – много. По такому принципу работают, например, многоядерные графические процессоры.

Векторные компьютеры также можно отнести к этому классу. Но считать их устройствами класса SISD тоже можно, если принять, что поток данных состоит из векторов.

MIMD = Multiple Instruction Multiple Data

Это – самое интересное. Предполагается, что одновременно выполняется много задач – или это подзадачи одной большой задачи, или это независимые расчеты, которые проводят в режиме коллективного доступа пользователи суперкомпьютера.

Из компьютеров или только из вычислительных узлов класса SISD или SIMD может быть организован многомашинный вычислительный комплекс. Иногда такие конструкции обозначают MSISD или MSIMD соответственно.

Очень важно, как организовано взаимодействие между процессорами и модулями памяти. Есть два крайних случая. МВК с общей памятью устроены так, что память доступна всем процессорам. Чтобы не было конфликтов, чтобы процессоры не пытались одновременно обращаться к одним и тем же данным, приходится предусматривать различные блокировки. Как следствие, быстродействие всей системы снижается. Принято считать, что

модель общей памяти не эффективна, если число процессоров больше, чем 32.

У МВК с локальной (распределенной) памятью каждый процессор получает свой участок оперативной памяти, но, разумеется, возможен обмен данными между локальными блоками. Как компромисс, используется гибридная архитектура – это типично для современных суперкомпьютеров. В этом случае используются оба подхода: каждый процессор имеет свою локальную память, но при этом может обращаться к глобальной памяти. Искусство программиста во многом определяется тем, насколько удачно удается использовать различные виды памяти и оптимально организовать обмен данными. При наличии общей памяти это сделать, казалось бы, просто. Но возникает проблема синхронизации, о которой будет подробно сказано ниже. В случае распределенной памяти используют обмен данными в форме пересылки сообщений.

При программировании для графических многоядерных процессоров (технология CUDA) оптимальное использование различных видов памяти – наиболее трудная задача.

По способам организации обращений к памяти МВК типа MIMD можно разделить еще и на подклассы.

UMA = Uniform Memory Access

В этом случае у нескольких процессоров имеется общая (разделяемая) память, обращение к ней осуществляется через общую шину. Наличие кэшей у процессоров существенно ускоряет работу.

DSM = Distributed Shared Memory

Вычислительный комплекс (возможно, из отдельных компьютеров) состоит из вычислительных узлов, в составе которых процессор с кэшами, локальная память и даже каналы ввода-вывода. Узлы связаны друг с другом высокоскоростной сетью.

Есть еще один вариант классификации, основанный на различиях в архитектуре МВК.

SMP-архитектура (Symmetric Multi-Processing)

Все процессоры имеют равноправный доступ к общей памяти через системную шину. Работой комплекса, как правило, управляет единая операционная система. Частые обращения к памяти всегда тормозят работу из-за конфликтов, поэтому число процессоров ограничено. Кэши одного процессора недоступны другому процессору.

Оперативное обновление данных в кэшах называют когерентностью кэшей.

MPP-архитектура (Massive Parallel Processing)

Система состоит из независимых узлов, память физически разделена. Узлы связаны высокоскоростными коммуникационными каналами. Полноценная операционная система может работать только на одном узле. Именно такие МВК ставят рекорды по производительности. Но пересылка из памяти узла в память другого узла (передача сообщений) замедляет работу. Для повышения эффективности используют разные топологии – системы связи между узлами (линейка, решетка, куб, тор …)

NUMA-архитектура (Nonuniform Memory Access)

Для комплексов такого типа память узлов физически разделена, но логически связана

(гибридная архитектура). Такая организация памяти наиболее подходит для алгоритмов со слабо связанными подзадачами.

Для машин с векторными процессорами используют название

PVP-архитектура (Parallel Vector Processing)

Кластер – это объединение в один комплекс нескольких компьютеров. Кластер может быть однородным (гомогенным) или неоднородным (гетерогенным). Один или несколько компьютеров в составе кластера обычно являются управляющими. Поддерживается

очередь заданий: каждая программа запрашивает необходимые ей ресурсы и запускается тогда, когда эти ресурсы свободны.

По рейтингу проекта Top500 в июне 2017 года первое место в мире по производительности занимал китайский суперкомпьютер Sunway TaihuLight, скорость вычислений этого комплекса достигла 93 петафлопс (пиковая производительность – 125 петафлопс). Предыдущий рекорд принадлежал также китайскому суперкомпьютеру Tianhe-2. Но новый чемпион имеет еще и хороший показатель энергопотребления – 6 гигафлопс на ватт. Стоимость всей системы порядка 270 млн долларов США.

Суперкомпьютер Sunway TaihuLight собран из 40960 64-битных процессоров SW26010 китайского производства, в конструкции каждого процессора по 4 управляющих ядра и 256 специализированных вычислительных ядер. Каждое ядро имеет 64 Кб памяти для данных и 16 Кб памяти для инструкций. Операционная система разработана на базе Linux.

Суперкомпьютер Tianhe-2 был построен на базе процессоров Intel Xeon E5-2692 с 12 ядрами. В его конструкции 16 тысяч вычислительных узлов, в составе каждого из которых два процессора, три сопроцессора Intel Xeon Phi 31S1P и в сумме 88 Гб оперативной памяти. На материнской плате два узла, 32 платы размещаются в стойке, всего 125 вычислительных стоек. Используется операционная система Kylin Unix, языки программирования Fortran, C/C++, Java и технологии параллельного программирования OpenMP и MPI (версия 3.0).

Российские суперкомпьютеры, к сожалению, никогда не поднимались в Top500 на первые места. МВК "Ломоносов", установленный в МГУ, начинал с производительности 510 терафлопс в 2009 году, но после модернизации этот показатель увеличился почти

втрое. В 2011 году он занимал 13-ю позицию в рейтинге. В июне 2016 года "Ломоносов-2" преодолел планку в 2 петафлопса. Пиковая производительность суперкомпьютера в Российском федеральном ядерном центре (г. Саров) составляет несколько петафлопс.

Отметим, что на самых мощных в мире вычислительных комплексах используются именно те технологии программирования, которые рассматриваются в данном пособии. Примеры программ, которые обсуждаются ниже, проверены на компактной супер-ЭВМ АПК-1М производства Российского федерального ядерного центра. Этот суперкомпьютер установлен в лаборатории суперкомпьютерного моделирования волновых процессов кафедры прикладной математики Института вычислительной математики и информационных технологий Казанского федерального университета и на виртуальной машине, созданной на кластере КФУ.

Hard и Soft

Для первых экспериментов вполне достаточно иметь обычный персональный компьютер, настольный или ноутбук, желательно с многоядерным процессором.

В данном учебном пособии все примеры написаны на языке C++ (или некоторых его расширениях). Для компиляции исходного кода рекомендуем использовать свободно распространяемый продукт – компилятор g++. Одно из его преимуществ состоит в том, что имеются версии этого компилятора для различных операционных систем – и для Windows, и для Linux.

Для работы в среде Windows можно установить на компьютере пакет MinGW, в составе которого находится коллекция бесплатных компиляторов, в том числе с языка C/C++, и различные необходимые

библиотеки. MinGW – это сокращение от Minimalist GNU for Windows.

Рекомендуется скачать необходимый софт с сайта mingw.org . Подробную инструкцию можно найти на сайте Национального открытого университета ИНТУИТ, дополнительный материал 1 к лекции 15 по программированию на C++ . Можно просто скопировать на свой компьютер с другого компьютера папку (каталог) C:\MinGW. Затем нужно в системной переменной PATH указать пути для доступа к компилятору и библиотекам – дописать в список путей ;C:\MinGW\bin;C:\MinGW\msys\1.0\bin .

Имеет смысл работать в командной строке Windows, чтобы облегчить в будущем переход на Linux . Для этого нужно выбрать (создать) рабочий каталог.

Затем через меню Windows вызвать программу «Командная строка» и с помощью команды cd перейти в этот каталог. Предположим, например, что исходный текст программы на C++

```
int main(int argc, char* argv[]) {
#include <stdio.h>

  printf("Hello, World!\n");
  return 0;
  }
```

сохранен в файле p01.cpp. Простая команда в командной строке

```
> g++  p01.cpp
```

(если все прошло успешно) создает в том же каталоге файл a.exe . Команда dir выводит на консоль содержимое каталога. Можно запускать исполняемый файл

```
> a
```

и смотреть, что получится.

Аргументы `int argc, char* argv[]` в этом случае не обязательны, но выработаем привычку их всегда писать. При вызове исполняемого файла операционная система передает ему кое-какую информацию: первый аргумент – сколько параметров, второй – их значения.

Особенность компилятора `g++` в том, что по умолчанию exe-файл получает имя `a.exe` .

Если нужно указать другое имя, например p01.exe , пишем

```
> g++ -o p01 p01.cpp
```

Только для компиляции используется команда

```
> g++ -c p01.cpp
```

После успешной компиляции в том же каталоге появляется объектный файл `p01.o` .

Еще один шаг – сборка, и появится исполняемый файл (он может быть собран из нескольких объектных файлов). Для компиляции OpenMP- и MPI-программ нужно подключать дополнительные библиотеки и указывать нужные ключи.

Если работать непосредственно в командной строке Windows, то в качестве редактора исходного кода можно использовать NotePad (блокнот). Нужно только не забывать сохранять редактируемый файл после изменений, иначе будет компилироваться предыдущая версия.

Удобно пользоваться файловым менеджером, например Far Manager. Встроенный редактор текста там есть, и в рабочем окне видно, как при компиляции появляются новые файлы.

Рекомендуем познакомиться с простым текстовым редактором Geany . Это также свободно распространяемый продукт. Иногда говорят, что Geany – это интегрированная среде разработки (IDE = Integrated Development Environment). Не совсем так. Эта программа не

имеет встроенного компилятора, а использует уже имеющиеся на компьютере, причем вызывает их в командной строке Windows. В главном меню Geany есть кнопки Compile или Скомпилировать (F8), Build или Сборка (F9) и Run или Выполнить, которые передают командной строке те самые команды, которые рассматривались выше, и отправляют на исполнение созданный exe-файл.

Если есть возможность поработать на МВК с операционной системой Linux или родственной ей, то в командной строке Linux все делается почти точно так же. Зарегистрированный пользователь МВК получает имя и пароль для доступа в систему и при входе попадает в свой рабочий каталог. Как правило, компилятор и дополнительные библиотеки уже инсталлировал администратор, а если нет – то его можно об этом попросить. Когда приходится начинать «с нуля», например, на новой виртуальной машине, то нужно все проделать самостоятельно.

Авторы получили доступ к виртуальной машине ДИС КФУ. Для экспериментов был создан виртуальный компьютер всего с 16-ю процессорами, пока этого достаточно. Мы инсталлировали на нашу машину пакет OpenMPI – программную реализацию стандарта обмена сообщениями между параллельными процессами (MPI) с открытым исходным кодом. В составе пакета есть и компилятор g++ , и библиотека для работы с OpenMP.

Для компиляции и запуска программы под Linux используют директивы

```
$ g++ p01.cpp
$ ./a.out
```

В Linux имя исполняемого файла (опять a) назначается по умолчанию, расширение out . Ключ -o позволяет установить нужное имя исполняемого файла. В нашем случае команда

```
$ g++ -o p01.out p01.cpp
```

создает исполняемый файл p01.out . Если же написать $ g++ -o p01 p01.cpp , то компилятор создаст файл p01 (с пустым расширением), который, впрочем, также можно запустить командой $ ./p01

Некоторое неудобство при работе в Linux появляется при редактировании исходных кодов программ. Но обычно с МВК под Linux работают с удаленных терминалов по локальной сети или через Internet. Для редактирования текстов программ и пересылки их в каталог на диске МВК удобно использовать программу WinSCP (вход по логину и паролю). Она представляет собой файловый менеджер, одно окно которого настраивается на работу с каталогом МВК. Второе окно совмещается с папкой на Windows-компьютере, что удобно для большинства пользователей.

Рекомендуется редактирование файлов проводить на удаленном терминале (левое окно), а потом пересылать на МВК (правое окно) и там компилировать и выполнять.

Программа PuTTy используется для доступа с Windows-компьютера к консоли среды Linux (вход по логину и паролю). В ее окне можно выполнять команды для компиляции исходных кодов и их исполнения, а также другие доступные пользователю МВК команды Linux. Если во время работы теряется связь через сеть, нужно закрыть эти программы и запустить заново.

Обе программы PuTTy и WinSCP являются свободно распространяемыми продуктами. Их инсталлируют из Интернета на компьютеры под управлением Windows, с которых и работают на кластере или на любом другом МВК.

Забегая вперед, добавим, что для компиляции OpemMP-программ нужно указать компилятору ключ -fopenmp (и для компиляции, и для сборки).

Для MPI-программ используют готовые сценарии. Если исходный код сохранен в файле q01.cpp , то его компилируют командой

```
$ mpiCC  q01.cpp  -o  q01.out
```

и исполняемый файл запускают командой

```
$ mpiexec  -np  16  q01.out
```

Здесь «16» – заказываемое число параллельных процессов (это значение передается через аргументы главной функции).

Заметим, что расширения файлов иногда можно не указывать, но лучше не злоупотреблять умолчаниями. Тем более, что в разных версиях софта, возможно, умолчания приняты не одинаковые.

При работе с MPI-программами под Windows нужно установить на компьютер пакет MPICH2 .

Использование технологии CUDA возможно только на компьютерах с CUDA-совместимыми многоядерными графическими процессорами (GPU). Специальный софт также необходим, его можно свободно скачать с сайта фирмы NVIDIA. Более подробно об этом будет сказано в четвертой части пособия.

Технология OpenMP

Самый короткий путь к использованию возможностей параллельного программирования – изучить технологию OpenMP (Open Multi-Processing). Последовательную программу можно дополнить специальными директивами для компилятора и вызовами функций времени выполнения, которые организуют параллельные вычисления.

Первая версия OpenMP была разработана в 1997 году для языка Fortran, а через год уже и для C/C++. Стандарт новых версий поддерживает некоммерческая организация OpenMP Architecture Review Board (http://www.openmp.org/), с которой сотрудничают многие крупные разработчики программных продуктов.

Основная идея OpenMP-программирования состоит в следующем. Код разделяется на параллельные и последовательные области. При запуске программы инициализируется основная нить (или поток, или процесс), которая выполняет все последовательные области. Дополнительные нити создаются при входе в параллельные области. Их количество определяется или до запуска программы, или непосредственно в программе с помощью специальной функции. При выходе из параллельной области основная нить ждет, когда завершатся все дополнительные нити.

Технология OpenMP сначала разрабатывалась для вычислительных комплексов с общей памятью (SMP). Но в 2005 году корпорация Intel выпустила софт для кластеров.

Рассмотрим технологию OpenMP на примере языка программирования C/C++. В литературе можно найти аналогичные примеры на языке Fortran.

Первые шаги

В стандартном наборе библиотек C++ имеется библиотечный файл omp.h , в котором содержатся описания типов данных и прототипов функций. Проверим, как работает первый пример:

```
// первая программа на OpenMP
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[]) {
#pragma omp parallel
  {
  printf("Hello, World!\n");
  }
return 0;
}
```

Директива #pragma omp parallel открывает параллельную область. На консоли должен появиться текст Hello, World! столько раз, сколько дополнительных нитей было создано. А сколько их может быть?

По умолчанию это количество определяется операционной системой в соответствии с доступным количеством процессоров на МВК или по количеству ядер у многоядерного процессора на ноутбуке. Если на консоли вашего компьютера появилось

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

то одновременно могут выполняться четыре нити.

Если при вызове компилятора не поставить опцию -fopenmp (или /fopenmp для компилятора Visual Studio), то директивы параллельного исполнения игнорируются. Поэтому может случиться так, что процессоров много, а вся работа будет поручена только

одному. Компилятор сообщает об этом, но не всем. В оболочке Geany в командах сборки по умолчанию устанавливается ключ -Wall. Но при вызове g++ в командной строке обычно не хочется писать лишнее…

Параметры функции main() можно не указывать, но раз уж проголосовали за ….

Легко проверить, поддерживает ли ваш компилятор технологию OpemMP:

```
// поддерживает ли компилятор OpenMP?
#include <stdio.h>
int main(int argc, char* argv[]) {
#ifdef _OPENMP
  printf("OpenMP is supported!\n");
#endif
}
```

С ключом компилятора -fopenmp , скорее всего, OpenMP is supported! Можно заказать много нитей с помощью функции omp_set_num_threads(…); которую нужно вызвать до директивы #pragma omp parallel . Можно также после слова parallel поставить опцию num_threads(…) . Но это вовсе не значит, что все они будут работать одновременно. Если процессоров не хватает на всех, то исполнение параллельных нитей происходит порциями в режиме разделения времени, в соответствии с возможностями системы.

Нумерация нитей

Каждая нить в параллельной области имеет свой номер, который определяется во время выполнения программы. Нумерация начинается с нуля. Основная нить получает номер 0 . Общее количество нитей также можно определить. Вот еще один стандартный пример:

```
// номер нити и количество нитей
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[]) {
  omp_set_num_threads(10);
  #pragma omp parallel
    {
    int t_n=omp_get_thread_num();
    int n_t=omp_get_num_threads();
    printf("Thread %i  from %i threads\n", t_n, n_t);
    }
  return 0;
  }
```

Существенно, что при выводе на консоль порядок следования строк не всегда соответствует номерам нитей. Нет соглашения о том, что нити выполняются последовательно, от нулевой нити до последней. Более того, при выводе данных они получают доступ к консоли в неопределенном порядке. Если ветви параллельного алгоритма полностью независимы, это не опасно.

Легко организовать серию расчетов по одному и тому же алгоритму при различных исходных данных. Их значения нужно вычислять в параллельной области по номеру нити или доставать элементы из массивов, подготовленных заранее в последовательной области. Например, если нужно составить таблицу значений функции $(x-1)(x-2)(x-3)$:

```
// таблица значений функции
#include <stdio.h>
#include <omp.h>
float f(float x) { return (x-1)*(x-2)*(x-3); };
#define n 101
```

```
int main(int argc, char* argv[]) {
  float a=0, b=4, h=(b-a)/(n-1), x, y;
  omp_set_num_threads(n);
  #pragma omp parallel
    {
    int t_n=omp_get_thread_num();
    x=a+h*t_n;  y=f(x);
    printf("x=%f,  y=%f\n", x, y);
    }
  return 0;
  }
```

Здесь и далее мы иногда не соблюдаем традицию, ставим в одну строку несколько операторов C/C++ .

Общие и частные данные

Очень важно разобраться, в какой области памяти хранятся и какое время существуют значения тех или иных переменных.

При входе в параллельную область каждая нить получает локальную область памяти. В ней размещаются объявленные в параллельной области переменные, они недоступны другим нитям. После завершения параллельной области все эти переменные уничтожаются.

Проверим. В примере 3 напишем еще раз строку `printf("%i %i \n", t_n, n_t);` перед `return 0;` (после закрывающейся фигурной скобки). Компилятор сообщит об ошибке.

Если переменные `tnum` и `numt` объявить до параллельной секции, то они буду общими для всех нитей (по умолчанию!). Например, если написать

…

```
int t_n, n_t;
#pragma omp parallel
  {
  t_n=omp_get_thread_num();
  n_t=omp_get_num_threads();
  printf("%i  %i \n",t_n,n_t);
  }
 printf("%i  %i \n",t_n,n_t);
...
```

то после выхода из параллельной области основная нить выведет на консоль значения. Но какие? У нас было так, что чаще всего выводились значения, вычисленные той нитью, которая последней получила доступ к консоли. Но совсем не обязательно …

Таким образом, объявленные до параллельной области переменные по умолчанию являются общими (shared) для параллельных нитей. За исключением переменных циклов. Если значения общих переменных были заданы в последовательной области, то они восстанавливаются при выходе из параллельной области. Если значения общих переменных не были заданы, то при выходе из параллельной области их значения непредсказуемы.

Более того, если несколько нитей захотят записать данные в общую область памяти, то не ясно, в каком порядке это получится. Особенно опасно, если при этом другие нити будут данные читать. Такую нежелательную ситуацию называют «гонка данных» (data race).

При входе в параллельную область можно и, как правило, нужно задавать явно ограничения на доступ к переменным, которые уже объявлены. Для этого в директиву `#pragma omp parallel` добавляются опции – списки переменных со спецификаторами:

`shared(…)` – переменные из этого списка являются общими для всех нитей;

private(…) – для переменных из этого списка создаются локальные копии для каждой нити, значения не устанавливаются при входе (нужно их задавать) и не сохраняются при выходе из параллельной области;

а также

firstprivate(…) – создаются локальные копии, значения устанавливаются такие же, какие были в предыдущей последовательной области, но они не сохраняются при выходе.

Через общие переменные можно передавать данные от нити к нити. Нужно только иметь гарантию, что читается именно то, что уже было записано.

Рассмотрим следующий пример:

```
// общие и частные данные
#include <stdio.h>
#include <omp.h>
int main() {
  int i=0, j=1, k=2;
  omp_set_num_threads(10);
  #pragma omp parallel shared(i) private(j) firstprivate(k)
    {
    i=i+omp_get_thread_num();
    j=i;
    k=j;
    }
  printf("%i  %i  %i\n",i, j, k);
  return 0;
  }
```

Значение переменной i после завершения параллельной секции будет 45 (сумма чисел от 0 до 9), значения j и k останутся 1 и 2 соответственно.

Иногда принимают соглашение: все переменные, объявленные в последовательной области – глобальные, в параллельной области – локальные. При этом нужно явно задавать начальные значения локальных переменных. За все отвечает программист!

Распараллеливание циклов

Если итерации цикла могут быть выполнены в любом порядке, независимо друг от друга, то их легко распределить между параллельными нитями. Простой, но не самый лучший способ удвоить элементы массива состоит в следующем:

```
// первый способ распараллеливания цикла
#include <stdio.h>
#include <omp.h>
#define n 100
int main(int argc, char* argv[]) {
  int m[n];
  for (int j=0; j<n; j++)  m[j]=j;
  omp_set_num_threads(10);
  #pragma omp parallel shared(m)
    {
    int n=10*omp_get_thread_num();
    for (int j=0; j<10; j++) m[n+j]=2*m[n+j];
    }
  return 0;
  }
```

Директива `#pragma omp for` позволяет сделать это еще проще:

```
// второй способ распараллеливания цикла
#include <stdio.h>
```

```
#include <omp.h>
#define n 100
int main(int argc, char* argv[]) {
  int m[n];
  for (int j=0; j<n; j++)  m[j]=j;
  omp_set_num_threads(10);
  #pragma omp parallel shared(m)
    {
    int t_n;
    #pragma omp for
    for (int j=0; j<n; j++) {
      m[j]=2*m[j];
      t_n=omp_get_thread_num();
      printf("Thread %i duplicates element %i\n", t_n, j);
      }
    }
  return 0;
  }
```

Эта директива распараллеливает следующий за ней цикл. Одно условие: количество повторений цикла должно быть заранее известно.

Как же распределена работа между нитями? На консоли видно, что нулевая нить обработала первые десять элементов массива, первая нить – вторые десять и так далее. Сообщения об этом выведены не последовательно, а так, как уж получилось.

Если количество элементов в массиве не делится нацело на 10, например, их 96, то можно использовать оператор `for (j<96) …`

При таком распараллеливании ускорения в 10 раз, конечно, не получится. Кроме присутствия последовательной области в программе, есть еще одна причина. На организацию параллельной области, выделение локальной памяти и прочее тоже нужно время. Этот процесс требует порядка тысячи машинных команд. Так что массив из 100 элементов дешевле обрабатывать последовательно.

Итак, директива #pragma omp for автоматически распараллеливает следующий за ней цикл или внешний цикл, если за этой директивой стоит конструкция из вложенных циклов. После директивы могут стоять опции со спецификаторами shared() , private() , firstprivate() и lastprivate() со списками переменных. Для переменных из списка опции lastprivate() создаются локальные копии, при входе в параллельную область их значения не инициализируются, но при выходе сохраняются значения, установленные нитью, которая выполнялась последней.

Можно режим распараллеливания цикла устанавливать вручную. Например, если каждой нити назначается 5 прогонов цикла, то пишем

```
#pragma omp for schedule (static,5)
```

Если раньше вычисления разбивались на столько порций, сколько устанавливалось параллельных нитей (конкретно: 10), то сейчас в каждой порции ровно 5 проходов цикла. После того, как все 10 нитей завершили работу, они снова получают данные. Нулевая нить обрабатывает элементы массива с 50 по 54, и так далее.

Опция вида schedule (dynamic,5) также предусматривает порции по 5 проходов цикла, но новые порции передаются не по порядку следования нитей, а по мере их освобождения. Вариант schedule (guided,5) предусматривает также динамическую загрузку нитей, но объем работы (не более, чем указано) устанавливает операционная система. Так должно быть быстрее.

Если указан параметр runtime , то размер порции определяется операционной системой.

Можно попробовать распараллелить в группе вложенных циклов не только внешний цикл, но также и внутренние циклы. Идеология OpenMP допускает, что внутри параллельной области каждая нить может породить новую вложенную параллельную

область, в которой эта нить будет главной. Не все компиляторы поддерживают такое.

Кроме того, если число процессоров ограничено, то все равно будет использован режим разделения времени.

Что такое «редукция» ?

Предположим, что нужно вычислить сумму элементов большого массива. Простейший, но не лучший (пока еще не сняты знаки комментариев) способ такой:

```
// пример на редукцию
#include <stdio.h>
#include <omp.h>
#define N 100
int main(int argc, char* argv[])
{
double a[N], s=0.0;
for (int j=0; j<N; j++) a[j]=j+1;
omp_set_num_threads(10);
#pragma omp parallel  // firstprivate(s) // reduction(+: s)
  {
  #pragma omp for // reduction(+: s)
  for (int j=0; j<N; j++) s=s+a[j];
  printf("%f\n",s);
  }
printf("%f\n", s);
return 0;
}
```

Что здесь плохого? Только то, что переменная s размещена в общей памяти и, следовательно, все нити обращаются к ней, толкаясь в очереди. Поставим firstprivate(s) после директивы parallel . Тогда каждая нить будет работать независимо. Но теперь после завершения

параллельной секции нужно собрать вместе найденные суммы. Передавать их основной нити и окончательно складывать в ней не очень эффективно. Выгоднее использовать опцию reduction(…) .

Если после директивы parallel написать reduction(+: s) , то для нитей создаются локальные значения s и после завершения параллельной секции их сумма помещается в глобальную переменную s (плюс ее начальное значение).

Если reduction(+: s) стоит после директивы for , то каждая нить также получает локальную переменную для суммирования, но редукция выполняется при выходе из цикла. Тогда у всех локальных s получается одно и то же значение – полная сумма элементов массива.

При редукции начальное значение локальных переменных устанавливается какое требуется: 0 для операции + , 1 для операции * . Редукция возможна для некоторых логических операций, а также при определении минимумов и максимумов (вместо + пишем min или max). Например:

```
// еще один пример на редукцию
#include <stdio.h>
#include <omp.h>
#define n 10000
int main(int argc, char* argv[]) {
  float x[n];
  for (int j=0; j<n; j++)  x[j]=5.0*rand()/RAND_MAX;
  omp_set_num_threads(4);
  float max_x=0.0;
  #pragma omp parallel for reduction(max: max_x)
    for (int j=0; j<n; j++)
      if (x[j]>max_x) max_x=x[j];
  printf("%f \n", max_x);
  return 0;
  }
```

Заметим, что в данном примере максимум находится очень быстро и в последовательной программе. Но ведь с распараллеливанием красивее?

Разрешается использовать сокращенные варианты директив. Например,

```
#pragma omp parallel for reduction(+: s)
  for (int j=0; j<N; j++) s=s+a[j];
```

Итак, любая директива OpenMP начинается со слов #pragma omp , затем ставится имя директивы и опции с параметрами, если они нужны. Можно каждую опцию начинать с новой строки, при этом впереди ставится знак \ .

Параллельные секции

Если нужно, то каждой нити в параллельной области кода можно дать свое задание. Проще всего сделать это с помощью оператора if :

```
…
int n=omp_get_thread_num();
if (n==0) … ;
if (n==1) … ;
…
```

Но лучше использовать специальные директивы #pragma omp sections и #pragma omp section . Например, так:

```
// параллельные секции
#include <stdio.h>
#include <omp.h>
int main() {
  omp_set_num_threads(10);
  #pragma omp parallel
```

```
{
int tn=omp_get_thread_num();
#pragma omp sections
 {
     #pragma omp section
     { printf("section 0, thread %i\n",tn); }
    #pragma omp section
     { printf("section 1, thread %i\n",tn); }
    #pragma omp section
     { printf("section 2, thread %i\n",tn); }
    }
 };
return 0;
}
```

Для выполнения действий внутри секции назначается одна нить. Какая нить именно – не регламентировано, но это всегда можно узнать. Возможно, при выделении секций некоторые нити будут простаивать. Если нитей меньше, чем секций, то каким-то нитям достанется больше работы. В приведенном примере нумерация нитей реальная, а нумерация секций – условная.

У директивы `#pragma omp sections` могут быть опции: `private()` , `firstprivate()` , `lastprivate()` , `reduction()` и `nowait` . Последняя опция из этого списка разрешает выполнившим свою работу нитям, не дожидаться, когда завершатся остальные нити.

Если некоторый участок кода в параллельной области нужно поручить выполнить только одной нити, то используется директива

`#pragma omp single`

с возможными опциями. Обычно это используется в случае, когда нужно что-то изменить в общих данных. Какая нить будет задействована – не уточняется.

Если нужно такое задание передать основной нити, то используют директиву

#pragma omp master

Есть возможность с помощью директивы #pragma omp task запустить новую независимую задачу. Ее выполнение может начаться сразу, если позволяют ресурсы, или может быть отложено на некоторое время.

Синхронизация

Наконец рассмотрим более подробно вопросы синхронизации параллельных процессов.

Нити в параллельной области завершаются не одновременно. По принятому в OpenMP правилу основная нить дожидается, пока не закончат работу все остальные нити. Точно так же при распараллеливании циклов система ожидает, пока не будут выполнены все итерации. Такая синхронизация называется барьерной. Для параллельных секций барьерная синхронизация также установлена неявно, по умолчанию.

Для директивы #pragma omp single установлена неявная барьерная синхронизация, а для #pragma omp master – не установлена.

Директива #pragma omp taskwait заставляет ждать, пока не завершатся все независимые задачи task .

Можно с помощью директивы

#pragma omp barrier

задать явную барьерную синхронизацию, именно в этом месте кода параллельные процессы будут приостановлены до тех пор, пока не

завершится последний из них. При необходимости можно отменить барьерную синхронизацию с помощью директивы

```
#pragma omp nowait
```

(если только это не приведет к непредсказуемым результатам).

Участок параллельного кода можно объявить критическим: `#pragma omp critical` . В каждый момент времени только одна нить может выполнять критический участок, остальные дожидаются своей очереди. Это нужно, например, если нити изменяют значения общих переменных.

Аналогично, директива `atomic` перед оператором присваивания не разрешает нескольким нитям одновременно выполнять эту операцию.

Переменные окружения

Функции OpenMP являются надстройками над API Windows или Linux. Значения переменных окружения операционных систем используются при определении параметров параллельного исполнения программ. Эти параметры можно изменять.

В переменной окружения OMP_NUM_THREADS хранится количество нитей, которые запускаются по директиве `#pragma omp parallel` . Функция `omp_set_num_threads(…)` изменяет значение OMP_NUM_THREADS . Это можно также сделать до запуска программы из командной строки.

С помощью функций `omp_get_num_procs()` и `omp_get_max_threads()` определяют количество доступных процессоров и максимальное количество нитей, которые можно использовать в параллельной области. Эти значения устанавливаются операционной системой исходя из ресурсов, имеющихся на текущий момент времени.

Иногда система пытается оптимизировать количество параллельных нитей, если OMP_DYNAMIC=1 . Функция omp_get_dynamic() возвращает значение этой переменной окружения, а функция omp_set_dynamic(…) устанавливает ее значение, 0 или 1.

Можно создавать вложенные параллельные области, хотя и не все компиляторы это принимают. При этом каждая нить, которая создает новую параллельную область, становится главной. Разрешены ли вложенные параллельные области – определяет переменная окружения OMP_NESTED .

Количество циклов, назначаемых нитям по опции schedule(runtime) , определяет переменная окружения OMP_SHEDULE .

Оптимизация параллельного кода

Прежде всего, поставим вопрос: какой код следует считать оптимальным? Сделать так, чтобы программа и работала очень быстро, и результат ее работы был всегда правильным, и оперативная память лишнего не расходовалась, и программировать было легко – явно не получится. Поскольку основная цель параллельного программирования – сократить время выполнения программы, будем стремиться к этому.

Время счета можно измерить программными средствами. Есть простая конструкция:

```
double t_start, t_finish;
t_start=omp_get_wtime();
…
t_finisht=omp_get_wtime();
double time=t_finish-t_start;
```

Здесь по показаниям системного таймера вычисляется время (в секундах), затраченное на выполнение фрагмента программы между двумя контрольными точками.

Технология OpenMP идеально приспособлена для последовательного улучшения кода: начинаем с параллельной программы, распараллеливаем циклы с независимыми итерациями, смотрим, что из этого получится. Затем пробуем сделать что-нибудь еще …

В директивы `#pragma omp parallel for` и в некоторые другие можно поставить опцию условного выполнения `if(…)` с условием, что параллельный процесс будет инициализирован только при достаточно большом числе повторений цикла, как правило, их должно быть не меньше 2000. Иначе затраты на создание дополнительных нитей, выделение локальной памяти и прочее перекроют выигрыш от распараллеливания.

Вряд ли следует заводить нитей больше, чем имеется доступных процессоров (хотя это и не запрещено).

Понятно, что последовательные области кода должны быть минимальными. Но все зависит от задачи. Трудно эффективно распараллелить алгоритм, если он последовательный по своей сути.

Желательно минимизировать количество обращений из параллельных нитей к общим глобальным переменным. Любая синхронизация замедляет работу программы.

И так далее … Нужно экспериментировать! Желаем удачи!

Еще два примера

Простая алгебраическая операция – перемножение двух матриц – хороший пример для экспериментов.

```
// перемножение двух матриц

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <omp.h>

#define N 250

int main(int argc, char* argv[]) {
double a[N][N], b[N][N], c[N][N];
srand(time(NULL));
for (int k=0; k<N; k++)
  for (int j=0; j<N; j++) {
    a[k][j]=double(rand())/RAND_MAX;
    b[k][j]=double(rand())/RAND_MAX;
    };

double t_start, t_finish;
t_start=omp_get_wtime();
double p;
// omp_set _num_threads(1);
// #pragma omp parallel for
for (int k=0; k<N; k++)
  for (int j=0; j<N; j++) {
        p=0.0;
        for (int i=0; i<N; i++) p=p+a[k][i]*b[i][j];
        c[k][j]=p;
    };
t_finish=omp_get_wtime();
printf("%f\n",t_finish-t_start);

printf("%f\n",c[0][0]);
return 0;
}
```

Мы предлагаем заполнить случайными числами две матрицы, перемножить их и измерить время, затраченное на перемножение. Нулевой элемент произведения выводится на консоль для того, чтобы компилятор не беспокоился.

Если снять комментарии, то окажется, что время счета немного увеличивается. Сказываются затраты на создание параллельной области. Увеличить размер матриц удается только в том случае, если имеется достаточный объем оперативной памяти.

Второй пример – решение системы линейных алгебраических уравнений (СЛАУ) методом Гаусса с выбором ведущего элемента.

```
// система линейных алгебраических уравнений
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <omp.h>

#define N 400

int main(int argc, char* argv[]) {
double a[N][N], b[N];
srand(time(NULL));
for (int k=0; k<N; k++) {
  for (int j=0; j<N; j++) a[k][j]=double(rand())/RAND_MAX;
  a[k][k]=a[k][k]+5.0;
  b[k]=double(rand())/RAND_MAX;
  };
double c,d; int n;
for (int k=0; k<N-1; k++) {
  c=fabs(a[k][k]); n=k;
  for (int i=k+1; i<N; i++) {
    d=fabs(a[i][k]);
    if (d>c) {c=d; n=i;};
```

```
    };
  if (n>k) {
    for (int j=k; j<N; j++) {
      c=a[k][j]; a[k][j]=a[n][j]; a[n][j]=c;
      };
    c=b[k]; b[k]=b[n]; b[n]=c;
    };
  for (int i=k+1; i<N; i++) {
    c=a[i][k]/a[k][k];
    for (int j=k+1; j<N; j++) a[i][j]=a[i][j]-a[k][j]*c;
    b[i]=b[i]-b[k]*c;
    };
  };
b[N-1]=b[N-1]/a[N-1][N-1];
for (int k=N-2; k>=0; k--) {
  for (int j=k+1; j<N; j++) b[k]=b[k]-a[k][j]*b[j];
  b[k]=b[k]/a[k][k];
  };
printf("%f  %f\n",b[0],b[1]);
return 0;
}
```

Матрица коэффициентов здесь заполняется случайными числами, но искусственно обеспечивается диагональное преобладание. Считается, что метод Гаусса плохо распараллеливается. Предлагаем читателю убедиться в этом (или опровергнуть такое мнение) самостоятельно. Но экспериментировать нужно на больших размерностях, поскольку программа работает достаточно быстро.

Заметим, что во многих сложных задачах, сводящихся к решению СЛАУ, основное время счета уходит на вычисление матриц коэффициентов и векторов правых частей. В таком случае СЛАУ можно решать и на одной нити.

Технология MPI

MPI (Message Passing Interface) – это стандарт и технология параллельного программирования, основанная на обмене сообщениями между параллельно выполняющимися процессами. Сообщения – это данные различных типов, которые передаются от одного процесса другому во время работы программы.

Технология MPI используется обычно на MPP-машинах (Massively Parallel Processing), наиболее эффективно на суперкомпьютерах и на многопроцессорных вычислительных комплексах. Тем не менее, изучение MPI и отладку программ можно проводить на одном процессоре. Несколько процессов могут выполняться в режиме разделения времени.

Первая версия MPI появилась в 1994 году. Свободная реализация MPICH2 (MPI Chameleon) для различных коммуникационных платформ активно развивалась в Аргоннской национальной лаборатории (США). В настоящее время имеется много различных версий MPI, свободных и коммерческих, для UNIX и для Windows, ориентированных в основном на использование языков программирования C++ и Fortran. Мы предпочитаем использовать пакет OpenMPI в среде Linux и MPICH2 для экспериментов под Windows.

Введение в MPI

Начнем с анализа текста простой MPI-программы. которая всего лишь создает коммуникационную среду:

```
// первая программа на MPI
// mpi01.cpp
#include <stdio.h>
#include “mpi.h”
int main (int argc, char* argv[])
 {
 // «1»
 MPI_Init (&argc, &argv);
 // «2»
 MPI_Finalize();
 // «3»
 return 0;
 }
```

Здесь на будущее зарезервированы три позиции:
«1» для предварительной части кода,
«2» для основной части параллельного кода,
«3» для завершающей части кода.
Предварительной и завершающей частей в программе может и не быть.

В отличие, например, от OpenMP-программ, последовательных секций здесь нет. Весь код выполняют все процессы, их количество задается при запуске откомпилированной программы.

Обсудим кратко отдельные детали.

Библиотека mpi.h содержит описания типов данных и функций MPI, обычно она отсутствует в стандартном наборе библиотек поддержки C/C++. В этом случае необходимо установить ее на компьютер самостоятельно.

Параметры у функции main() обязательны (напомним, что это – количество параметров командной строки и массив параметров).

Функция MPI_Init() создает группу параллельных процессов. Стандартное имя этой группы MPI_COMM_WORLD (его обычно называют коммуникатором). Есть возможность создавать и другие

группы. Вся среда передачи сообщений может быть настроена программистом.

Каждый процесс получает уникальный номер (или ранг) в группе. Процессы могут обмениваться данными друг с другом через область связи, которая выделяется при инициализации группы.

Функция MPI_Finalize() завершает основную часть кода. После ее вызова обмен данными между процессами становится невозможным.

Вторая программа, которая определяет номер процесса и общее количество процессов (классический пример, приводится во всех руководствах по MPI):

```
// номер процесса и количество процессов
// mpi02.cpp
#include <stdio.h>
#include “mpi.h”
int main (int argc, char* argv[])
 {
 int rank, size;
 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
 MPI_Comm_size (MPI_COMM_WORLD, &size);
 printf(“Hello from process %d of %d \n”, rank, size);
 MPI_Finalize();
 return 0;
 }
```

Обратим внимание на следующее. При запуске MPI-программы каждому процессу выделяется необходимая область оперативной памяти. Каждый процесс имеет доступ только к своим собственным переменным (в нашем примере: rank и size). Переменные можно объявлять и после того, как будет инициализирована группа процессов. Но вот функции MPI_Comm_rank() и MPI_Comm_size()

нельзя использовать раньше, чем будет создана эта группа, а также после вызова функции MPI_Finalize() . Но значения переменных rank и size (и других переменных, если они были объявлены) сохраняются после разрушения области связи группы. Разумеется, они уникальны для каждого процесса.

Существенно, что при запуске программы процессы «передают привет» не в порядке возрастания номеров, а в какой-то случайной последовательности (это легко проверить). Невозможно предусмотреть заранее, в каком порядке процессы получат доступ к консоли. Но заметим, что по стандарту MPI все процессы работают с консолью того компьютера (если их несколько), с которого была запущена программа.

Нетерпеливый читатель может попробовать начать освоение технологии MPI с проверки, как будет работать его последовательная программа одновременно на нескольких процессорах при различных исходных данных. Для этого нужно поместить код программы в позицию «2» после того, как определены значения size и rank . Если исходные данные для счета вычислять по значению rank , то каждый процесс вернет результат, соответствующий именно этим данным.

Компиляция и запуск

Для параллельных вычислений наиболее удобна платформа операционной системы Linux или ей подобной. Как правило, именно эта ОС устанавливается на МВК и на суперкомпьютерах.

Для компиляции обычно используются готовые сценарии (или директивы) mpicc (для программ на языке C) и mpiCC (для программ на C++). Например, так:

```
mpiCC mpi01.cpp –o mpi01.out
```

Если компиляция завершилась удачно, то в текущем каталоге появится исполняемый файл omp01.out . Его можно запустить:

```
mpirun –np 16 pr1.out
```

Здесь 16 – желаемое количество параллельных процессов. С той же целью можно использовать сценарий mpiexec .

В некоторых случаях расширения cpp и out можно не указывать. Кроме того, при обращении к сценариям можно указывать некоторые другие ключи (опции).

Как уже было сказано, принцип работы параллельной программы следующий. Каждому процессу группы передается полная копия исполняемого кода и выделяется память. Вычисления проводятся независимо (если не был предусмотрен обмен сообщениями). Каждый процесс выводит свои результаты счета на консоль (если это планируется), но в каком-то непредсказуемом порядке.

За ошибками, которые могут возникнуть во время выполнения программы, предлагается следить самому программисту.

Большинство функций среды MPI возвращают код MPI_SUCCESS при успешном завершении или код ошибки. Этим можно пользоваться, например, так:

```
// код завершения функции
// omp03.cpp
#include <stdio.h>
#include “mpi.h”
int main (int argc, char* argv[])
 {
 int code;
 code=MPI_Init (&argc, &argv);
 if (code==MPI_SUCCESS)
  printf(“Successful initialization\n”);
 else
  printf(“Initialization failed! Error code %d\n”,code);
```

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

Подобные проверки можно проводить очень часто, но вряд ли такое кому-либо захочется. Но это имеет смысл делать при отладке алгоритма.

Параллельная обработка массивов

Многие операции над векторами и матрицами можно выполнять параллельно. Большие массивы естественно разрезать на куски и распределять вычислительную работу между независимыми параллельными процессами. При этом возможны различные подходы к распараллеливанию алгоритмов, имеющие как положительные, так и отрицательные стороны. Если цель – сократить время счета, то нужно искать компромисс между ускорением вычислений при увеличении числа процессоров и потерями во времени выполнения из-за долгой пересылки данных от процесса к процессу.

Например, одна из простейших операций в линейной алгебре – умножение вектора на скаляр – может быть осуществлена так.

```
// умножение вектора на скаляр
// в позиции «1»
float x[256], a;
for (int j=0; j<256; j++) x[j]=j;
a=1.5;
// в позиции «2»
int rank, size;
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
for (int j=rank*256/size; j<(rank+1)*256/size; j++)
x[j]=x[j]*a;
```

Самое главное здесь – техника разрезания массива на части (предположим, что 256 делится нацело на size). Но не все так просто.

Каждый процесс формирует исходный массив x[]. Но при этом, легко видеть, выполняется лишняя работа. Можно было бы поручить вычислять элементы массива только одному процессу, а потом переслать результат остальным (как это сделать, будет показано ниже). Но в это время остальные процессы будут простаивать.

Можно заполнение массива также разделить между процессами, но тогда каждый процесс будет иметь доступ только к своей части данных. В данном примере этого было бы достаточно, но в более сложных задачах нужно заботиться об обмене данными. Неизвестно, что быстрее.

Но, скорее всего, без пересылки значений элементов массива все равно не обойтись. Дело в том, что вычисленные произведения не слиты в один массив. Если в позиции «3» поставить строку

```
for (int j=0; j<256; j++) printf("%f\n", x[j]);
```

то на консоли появится size наборов по 256 чисел, в каждом из которых только size чисел отличаются от значений элементов исходного массива. Советуем проверить это (можно уменьшить размерность массива).

Напомним, что значение size задается при запуске программы. Так что вопрос о том, на сколько частей будет разделен массив, откладывается до времени выполнения.

Трудные вопросы:

- Все-таки, если ли разница, когда объявлять переменные в программе – в позиции «1» или в позиции «2»?
- Пусть в позиции «2» определены ранги процессов. Если в позиции «3» вывести на консоль их значения, то порядок также будет неопределенным?
- Могут ли параллельные процессы иметь локальную память разного размера?

- Может ли быть полезна параллельная программа, в которой процессы не объединены в группу?

Простейший обмен сообщениями

Технология MPI предоставляет широкий набор функций для обмена сообщениями между процессами. Начнем с самого простого варианта.

```
// блокирующий обмен сообщениями
. . .
  int mess;
  MPI_Status status;
  if (rank==0)
    for (int j=1; j<size; j++)
      {
      MPI_Recv(&mess, 1, MPI_INT, MPI_ANY_SOURCE,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      printf("Message from process %d", message);
      };
    else
    MPI_Send(&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  . . .
```

Нулевой процесс принимает сообщения от всех остальных процессов, а они, в свою очередь, отправляют сообщения ему. Это сообщение короткое – всего лишь номер процесса. Для этого используются две функции с длинными списками параметров.

Функция MPI_Send () посылает сообщение. Ее параметры: адрес буфера, количество элементов в сообщении, тип элементов, номер принимающего процесса, тэг сообщения, имя коммуникатора.

Функция MPI_Recv () принимает сообщение. Ее параметры: адрес буфера, количество элементов в сообщении, тип элементов,

номер передающего процесса, тэг сообщения, имя коммуникатора, статус принимаемых данных.

В качестве буфера обычно используются области памяти, отведенные под какие-либо переменные. Значения этих переменных и пересылаются. Сообщение состоит из одного или нескольких элементов. Принятые в MPI типы элементов соответствуют аналогичным типам данных в используемых языках программирования (стандарт MPI не привязан к какому-либо конкретному языку).

Тэг сообщения (иногда говорят: ярлык, или просто номер сообщения) – целое число, позволяющее отличить одно сообщение от другого при тех же самых номерах источника и приемника.

Обычно значения тэга лежат в диапазоне от 0 до 32767.

Предусмотрено, что сообщения могут быть прочитаны не в том порядке, в каком они были отправлены.

Структура MPI_Status имеет три поля:

MPI_SOURCE – источник сообщения,

MPI_TAG – тэг сообщения,

MPI_ERROR – код ошибки полученного сообщения.

Возвращаемое значение status позволяет определить источник и тэг сообщения, если они не указаны явно. Его также можно использовать для дополнительного контроля над процессом передачи данных.

Итак, в данном примере ненулевые процессы отправляют сообщения нулевому процессу. Он же готов принять данные от любого процесса (указано, что номер передающего процесса – любой, MPI_ANY_SOURCE). Тэг при приеме также может быть любым.

Коды MPI_ANY_SOURCE и MPI_ANY_TAG в принимающей функции можно заменить на j и 0 . Ясно, что второе действие ничего не меняет. Но первая замена заставляет принимающую функцию читать не первое попавшееся сообщение, а сообщение от конкретного процесса с номером j . При этом, скорее всего, время выполнения

программы увеличится (интересно, намного ли?), так как процессы отправляют сообщения не одновременно, а когда операционная система предоставит такую возможность.

Наконец, условные операторы вида if (rank==…) позволяют назначать каждому процессу в группе свое задание.

Возможные ошибки

Уточним некоторые детали. Процесс-отправитель пересылает сообщение не непосредственно в буфер процесса-получателя, а в системный буфер (в область памяти, выделенную при инициализации режима параллельного выполнения процессов для временного хранения сообщений). Процесс-получатель достает сообщение из системного буфера. В простых программах это незаметно, поскольку пересылка коротких сообщений происходит быстро.

По стандарту MPI простейший обмен сообщениями является блокированным в том смысле, что процесс-отправитель ждет, пока сообщение не скопируется в системный буфер, а процесс-получатель ждет, пока сообщение не перепишется из системного буфера в буфер получателя. Другие протоколы обмена данными будут рассмотрены ниже.

При блокированном обмене нельзя допускать ситуацию, когда несколько процессов одновременно отправляют сообщения друг другу и ждут подтверждения, что эти сообщения получены.

Таким образом, если выполняются действия:

```
// процесс 0
MPI_Recv(…,1,…);
MPI_Send(…,1,…);
// процесс 1
MPI_Recv(…,0,…);
MPI_Send(…,0,…);
```

то возникает тупиковая ситуация deadlock . Есть два способа исправить положение: или

```
// процесс 0
MPI_Recv(…,1,…);
MPI_Send(…,1,…);
// процесс 1
MPI_Send(…,0,…);
MPI_Recv(…,0,…);
```

или

```
// процесс 0
MPI_Send(…,1,…);
MPI_Recv(…,1,…);
// процесс 1
MPI_Send(…,0,…);
MPI_Recv(…,0,…);
```

Но в последнем случае может оказаться, что места в системном буфере не хватит.

При пересылке сообщений нужно следить за соответствием типов данных в параметрах функций. Основные типы данных MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE и MPI_BYTE совместимы с соответствующими типами в языках C/C++ и Fortran. Есть возможность схитрить и переслать данные любого типа, если использовать параметры sizeof(…), MPI_BYTE ☺

Тип MPI_PACKED используется при обмене упакованными данными. Упаковку и распаковку выполняют функции MPI_PACK() и MPI_UNPACK() .

Нельзя при обмене сообщениями получить данных больше, чем отослано, но можно принять их меньше. То, что не было принято, теряется.

С помощью функции MPI_Get_count() можно узнать, сколько точно данных было принято. Например, так:

```
MPI_Status stat;
int count;
MPI_Recv(…, MPI_FLOAT, …, &stat);
MPI_Get_count(&stat, MPI_FLOAT, &count);
```

Если нужно узнать, сколько данных было отправлено (чтобы подготовить приемный буфер), используют функцию MPI_Probe() . Например,

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
                                    MPI_COMM_WORLD, &stat);
MPI_ Get_count(&stat, MPI_BYTE, &count);
char *buf;
buf=malloc(count);
MPI_Recv(buf,count,MPI_BYTE,…);
```

При этом сообщение остается в системном буфере, но тип данных остается неопределенным. Как интерпретировать принятую информацию – дело программиста …

Для упрощения процедуры пересылки данных предусмотрена совмещенная функция MPI_Sendrecv() , в списке ее параметров первые пять – как у MPI_Send() , а следующие семь – как у MPI_Recv() . Совмещенная функция более безопасна, так как в этом случае исполняющая среда MPI организует пересылку сообщения без конфликтов.

Процесс может пересылать сообщения самому себе при вызове функции MPI_Sendrecv_replace() .

Неблокирующие функции обмена

Можно запустить процесс передачи данных и, не дожидаясь его завершения, продолжать вычисления. Стандартная неблокированная

передача данных начинается с вызова функции MPI_Isend() , параметры которой: адрес буфера, количество элементов в сообщении, тип элементов, номер принимающего процесса, тэг сообщения, имя коммуникатора и имя запроса. Запрос представляет собой объект типа MPI_Request , он используется для контроля за состоянием сообщения.

Стандартный неблокированный прием данных организует функция MPI_Irecv() , в списке ее параметров также присутствует имя запроса: адрес буфера, количество элементов в сообщении, тип элементов, номер передающего процесса, тэг сообщения, имя коммуникатора, имя запроса. А где же статус сообщения?

Функция MPI_Wait() имеет два параметра: имя запроса и статус сообщения. Она возвращает управление, когда операция обмена завершилась. Возвращаемый параметр статус используется так же, как при блокированном обмене, а объект-запрос получает значение MPI_REQUEST_NULL , и с этого момента он может использоваться в других операциях неблокированного обмена. Блокировка процесса обмена все-таки есть, но она откладывается на некоторое время.

Функция MPI_Test() имеет три параметра: имя запроса, флаг и статус сообщения. Если пересылка данных еще не закончилась, то через параметр-флаг возвращается значение false , и выполнение программы продолжается. Если обмен завершен, то возвращается значение true , в параметр-статус записывается статус сообщения и объект-запрос обнуляется. Таким образом, можно в цикле выполнять полезную работу и время от времени спрашивать, не пришли ли нужные для дальнейшего данные. Если ждать больше не хочется, можно попробовать остановить процесс передачи сообщения вызовом функции MPI_Cancel() , ее единственный параметр – имя запроса.

Опасно обращаться к буферам отправителя и получателя до тех пор, пока обмен данными не завершен.

Для приема данных в режиме обмена сообщениями между двумя процессами имеется только две функции. Но для передачи данных возможностей больше.

Буферизованный обмен сообщениями предусматривает, что с помощью функции MPI_Buffer_attach() будет предварительно создан буфер в памяти процесса. Параметров у этой функции два: имя буфера и размер в байтах. Если память выделить не удалось, функция вернет код ошибки. Такой способ используют в тех случаях, когда ожидается, что в системном буфере места для данных, возможно, будет недостаточно.

При буферизованной передачи сообщений используют функции MPI_Bsend() и MPI_Ibsend(). Функция MPI_Buffer_detach() освобождает выделенную под буфер память.

Функции MPI_Ssend() и MPI_Issend() используются для синхронной передачи данных, то есть только после того, как запущена соответствующая принимающая функция.

Наконец, передача данных «по готовности» возможна только в том случае, когда принимающая функция уже запущена. Иначе результат операции будет непредсказуемым. Имена блокирующей и неблокирующей функций MPI_Rsend() и MPI_Irsend().

Такое разнообразие функций обмена позволяет повышать надежность и быстродействие программ. Отметим, что по идеологии MPI блокирующие и неблокирующие функции совместимы: можно отправлять данные одним способом, а принимать другим.

Коллективный обмен данными

Если нужно вычислить, например, скалярное произведение двух длинных векторов, то последовательность действий может быть следующей. Один из процессов (как правило, нулевой) заполняет исходные массивы данных и рассылает их остальным процессам. Каждый процесс группы вычисляет сумму произведений элементов

массивов и возвращает ее нулевому процессу. Нулевой процесс складывает принятые значения.

В MPI имеются функции, рассылающие сообщения всем процессам группы и принимающие сообщения от всех процессов. Покажем, как их использовать. Начнем так:

```
// скалярное произведение векторов

#include <stdio.h>
#include <mpi.h>
int main (int argc, char* argv[])  {
int N=102400;
double x[N], y[N];
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if rank==0 {
  // процесс 0 заполняет массивы
  …
  };
MPI_Bcast(x,N,MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(y,N,MPI_DOUBLE, 0, MPI_COMM_WORLD);
// и рассылает данные всем остальным процессам группы
```

Предположим, что N нацело делится на size и продолжим.

```
double sum=0.0;
for (int j=rank*N/size; j<(rank+1)*N/size; j++)
    sum=sum+x[j]*y[j];
double *buf;
buf=(double*) malloc(size*sizeof(double));
// но лучше  double buf[size];
MPI_Gather(&sum,1,MPI_DOUBLE,
     buf,1,MPI_DOUBLE,0, MPI_COMM_WORLD);
// нулевой процесс собрал данные
```

Осталось вычислить сумму res элементов массива buf[] и завершить программу

```
if (rank==0) printf("Product = %f", res);
MPI_Finalize();
return 0;
}
```

Если есть опасение, что не все параллельные процессы одновременно заканчивают вычисления, то нужно для барьерной синхронизации вызывать функцию

```
MPI_Barrier(MPI_COMM_WORLD);
```

Попробуем улучшить код программы. Некоторое неудобство состоит в том, что функция MPI_Bcast() рассылает массивы по процессам целиком, а хорошо бы отправлять только те их части, которые нужны для вычислений. Для этого предназначена функция MPI_Scatter() . В нашем примере можно организовать рассылку так:

```
double xx[N/size], yy[N/size];
MPI_Scatter(x,N/size,MPI_DOUBLE,
   xx,N/size,MPI_DOUBLE, 0,MPI_COMM_WORLD);
```

Задумаемся теперь, хорошо ли было поручать нулевому процессу подготовку исходных данных для всех остальных процессов?

Но это еще не все. После того, как процессы вычислили свои части большой суммы, лучше собрать их вместе следующим образом:

```
double res;
MPI_Reduce (&sum, &res, 1, MPI_DOUBLE, MPI_SUM,
   0,MPI_COMM_WORLD);
```

Функция MPI_Reduce () выполняет уже знакомую операцию редукции – она собирает в переменной res сумму (это задано параметром MPI_SUM, это имя одной из возможных предопределенных редуцированных операций) значений переменных

sum из разных процессов. В итоге у всех процессов получается одно и то же значение res .

Итак, функция MPI_Bcast() передает данные от одного процесса всем процессам группы; функция MPI_Scatter() «разбрасывает» данные от одного процесса порциями по всем процессам, а функция MPI_Gather() собирает данные от всех процессов в одном процессе. Есть также функции, обеспечивающие обмен сообщениями «всем от всех» и векторные варианты функций коллективного обмена данными, когда передача данных идет неравными порциями.

Задача Дирихле для уравнения Лапласа

Явная разностная схема, которая используется при численном решении задачи Дирихле для уравнения Лапласа, – удобный пример для иллюстрации различных вариантов обмена данными в технологии MPI. Как известно, исходная задача ставится так: нужно найти в области решение уравнения Лапласа, которое принимает заданные значения на границе области. Пусть эта область – прямоугольник в плоскости x, y. Построим сетку с узлами (x_j, y_k), $j = 0..m+1$, $k = 0..n+1$. Будем искать значения искомого решения в узлах: $u_{j,k} = u(x_j, y_k)$. Предположим для простоты рассуждений, что сетка является равномерной и расстояния между соседними узлами $h_x = h_y = h$. Тогда для всех внутренних узлов сетки

$$u_{j,k} = 0.25\,(u_{j+1,k} + u_{j-1,k} + u_{j,k+1} + u_{j,k-1}),\ j = 1...m,\ \ k = 1...n.$$

Используем итерационный метод: будем в цикле перевычислять эти значения до тех пор, пока не будет достигнута заданная точность. В граничных узлах значения $u_{j,k}$ не изменяются, они определяются по граничным условиям. В качестве начального приближения можно взять нулевые значения.

В теле итерационного цикла напишем, например, такой код:

```
int j, k;
float u [m+2] [n+2], v [m] [n];
while (…)  // пока точность не достигнута
{
for (j=1; j<=m; j++)
  for (k=1; k<=n; k++)
     v [j-1] [k-1] = 0.25 * (u [j+1] [k] +u [j-1] [k] +u [j] [k+1] +u [j]
[k-1]);
for (j=1; j<=m; j++)
  for (k=1; k<=n; k++)
     u [j] [k] = v [j-1] [k-1];
}
```

Ясно, что все вычисления независимы друг от друга, их легко распределить между несколькими параллельными процессами. Обсудим, как это лучше сделать.

Первый вопрос, на который нужно ответить, – как разрезать массивы u и v на части. Основных вариантов три: по строкам, по столбцам или прямоугольными блоками. Так как матрицы хранятся в памяти компьютера по столбцам, то в данной задаче разделять их на части целесообразно также по столбцам.

Удобно все рассуждения проводить на уровне номеров столбцов. В расчетах участвуют столбцы матрицы u с номерами от 0 до $m+1$, но перевычисляются только столбцы с номерами от 1 до m. Пусть будет запущено p параллельных процессов, причем m делится нацело на p и $q = m/p$. Таким образом, каждый процесс будет перевычислять q столбцов большой матрицы. При этом каждому процессу должны быть доступны еще два столбца – один левее левого и один правее правого.

В результате процесс 0 будет работать со столбцами с номерами от 0 до $q+1$, процесс 1 – с номерами от q до $2q+1$, и так далее. Для

каждого процесса будем использовать локальную нумерацию столбцов, как у нулевого процесса: от 0 до $q+1$,

После того, как завершится прогон цикла, нужно провести обмен крайними столбцами. В процессе такого обмена

процесс 0:

из столбца q пересылает данные в столбец 0 процесса 1,

в столбец $q+1$ принимает данные из столбца 1 процесса 1;

каждый процесс с номером r от 1 до $p-1$:

из столбца q пересылает данные в столбец 0 процесса $r+1$,

в столбец $q+1$ принимает данные из столбца 1 процесса $r+1$;

из столбца 1 пересылает данные в столбец $q+1$ процесса $r-1$,

в столбец 0 принимает данные из столбца q процесса $r-1$;

процесс p :

из столбца 1 пересылает данные в столбец $q+1$ процесса r,

в столбец 0 принимает данные из столбца q процесса r.

Может быть, это выглядит немного сложно … но так должно быть.

Пусть у каждого процесса данные хранятся и перевычисляются в массивах

```
float a [q+2] [n+2], b [q] [n];
```

С помощью блокирующих функций MPI обмен организуется так:

```
if (rank==0)
{
MPI_Send(&a[q][1], n, MPI_FLOAT, 1, tag,
                                        MPI_COMM_WORLD);
MPI_Recv(&a[q+1][1], n, MPI_FLOAT, 1, tag,
                                MPI_COMM_WORLD, &status);
}
```

и так далее …

Беспокоиться можно только о том, что хватит ли размера системного буфера, если все процессы отправят данные соседу, но не успеют прочитать то, что им было послано.

Аналогичные действия можно разделить между процессами с нечетными и с четными номерами: все нечетные процессы отправляют данные влево, а все четные (кроме, может быть, последнего) принимают справа. Затем все нечетные принимают данные слева, а все четные (кроме, может быть, последнего) передают вправо.

Более простой и понятный код получится, если использовать объединенные функции для передачи и приема сообщений. Кроме того, удобно использовать константу MPI_PROC_NULL – нулевой процесс. Процессу с таким номером можно передавать сообщения и принимать сообщения от него. Но при этом ничего не произойдет.

Код такой:

```
if (rank==0)  left=MPI_PROC_NULL;
else left=rank-1;
if (rank==p-1) right=MPI_PROC_NULL;
else right=rank+1;
MPI_Sendrecv(&a[1][1], n, MPI_FLOAT, left, tag,
   &a[0][1], n, MPI_FLOAT, left, tag, MPI_COMM_WORLD,
&status);
MPI_Sendrecv(&a[q][1], n, MPI_FLOAT, left, tag,
   &a[q+1][1], n, MPI_FLOAT, left, tag, MPI_COMM_WORLD,
&status);
```

Можно попробовать сократить время выполнения программы, если совместить операции обмена сообщениями с вычислениями. Нужно вычислять первыми крайние столбцы массива *b* и сразу отсылать их соседям с помощью функций неблокированного обмена [4].

Еще немного об MPI

При параллельном программировании каждый решает сам, что важнее – время выполнения программы, экономия памяти или красота и наглядность кода.

Оценить время выполнения параллельного кода можно следующим образом:

```
double start_t, finish_t;
start_t=MPI_Wtime();
…
finish_t=MPI_Wtime();
printf(“Time = %f\n”,finish_t-start_t);
```

Таким способом время, прешедшее между двумя временными метками, вычисляется в секундах. Время это «местное», каждый процесс выведет на консоль свой результат.

Мы ничего не написали о типах данных, определяемых пользователем. Пусть это будет заданием для самостоятельной работы.

Наконец, имеется возможность создавать виртуальные топологии вычислительных процессов. Речь идет об организации связей между процессами. По умолчанию каждый процесс может обмениваться данными с любым другим процессом, такое взаимодействие называют «полный граф». При большом количестве процессоров параллельные вычисления можно ускорить, если сократить количество возможных взаимодействий и перейти к топологиям «линейка», «кольцо», «решетка» или «тор». Все зависит от программируемого алгоритма.

Технология CUDA

Технология CUDA (Compute Unified Development Architecture) используется при решении вычислительных задач на графических процессорах (GPU, Graphics Processing Unit).

Компания NVIDIA, специализирующаяся на разработке ускорителей трехмерной графики, выпустила в 2006 году видеокарту GeForce 8800 GTX. Оказалось, что это устройство можно использовать и для решения более широкого круга вычислительных задач. Через несколько месяцев был создан компилятор для нового языка CUDA C. Как следствие, появилась новая технология – CUDA.

Графические ускорители стали мощными SIMD-процессорами или, иными словами, массивно-параллельными устройствами с памятью большого объема. На персональных компьютерах GPU устанавливается на видеокарте и может рассматриваться как сопроцессор к CPU (Central Processing Unit). В более сложных многопроцессорных вычислительных комплексах таких устройств может быть много, но они уже не используются для построения изображения на экране монитора.

Мы рассмотрим далее версию CUDA, ориентированную на язык программирования C/C++.

Инсталляция CUDA

Если у читателя есть доступ к МВК с графическими процессорами, поддерживающими CUDA, то, скорее всего, необходимые программы на нем уже имеются. Установить нужный софт на личном ноутбуке достаточно просто. При работе под Windows, к нашему сожалению, не обойтись без Microsoft Visual Studio. Дело в том, что компилятор от NVIDIA (nvcc.exe) обрабатывает CUDA-код и формирует команды только для GPU. Все остальное доделывает компилятор Visual Studio. В среде Linux можно

работать с компилятором g++, но для Windows это пока не предусмотрено (возможно, в ближайшем будущем …).

Итак, если это еще не сделано, инсталлируем на свой компьютер Microsoft Visual Studio. При установке typical , скорее всего, компилятор командной строки cl.exe сразу не загрузится. Нужно еще раз вызвать установщик, перейти в режим modify и выбрать опцию Language: C++ .

Чтобы в дальнейшем можно было работать из командной строки или из оболочки Geany, в переменную среды path допишем путь к папке, где находится файл cl.exe . Скорее всего, это C:\Program Files (x86)\Microsoft Visual Studio …\VC\bin .

С сайта developer.nvidia.com скачаем установщик CUDA Toolkit (выбрав правильную версию) и запустим его. В старых версиях CUDA необходимо было еще предварительно загрузить драйверы для графического устройства.

Для работающих в Geany нужно в разделе меню Документ установить тип файла: cuda . Тогда в настройках компилятора и сборщика сразу появятся нужные команды. Расширение файлов с CUDA-программами должно быть .cu , а не .c и не .cpp .

При работе в командной строке (лучше использовать командную строку MS Visual Studio, тогда будут автоматически подключаться необходимые библиотеки) нужно использовать директиву

```
nvcc <имя файла>.cu  -o  <имя файла>.exe
```

Заметим, что при использовании не самых новых видеокарт NVIDIA возможны некоторые проблемы. Новые версии CUDA не работают со старыми картами (компиляция программ пройдет, но exe-файлы выполняться не будут), о чем установщик CUDA, скорее всего, предупредит.

Для старой видеокарты нужно установить соответствующую версию CUDA и, возможно, необходимые драйверы. Но старые версии CUDA не работают с новыми версиями cl.exe . Поэтому перед

установкой CUDA на компьютер хорошо бы выяснить, совместимы ли hard и soft .

Заметим также, что не все возможности cuda-расширения языка C реализованы в старых версиях CUDA.

Принята следующая терминология. Центральный процессор (CPU) называют хост (host) и, соответственно, доступную ему оперативную память на материнской плате – память хоста. Графический процессор (GPU) называют устройство (device) На устройстве имеется несколько видов доступной ему памяти, об этом будет подробнее сказано позже.

Предусмотрено, что можно пересылать данные из памяти хоста в память устройства, и, разумеется, обратно.

Общий принцип работы следующий:

1) на хосте готовятся исходные данные для расчета;
2) эти данные пересылаются в память устройства;
3) на устройстве вызываются функции, обрабатывающие данные;
4) результат возвращается в память хоста.

В простых задачах пересылка данных из памяти в память занимает существенно больше времени, чем счет. Поэтому таких пересылок должно быть как можно меньше. Да и вообще, эффективность применения технологии CUDA зависит, прежде всего, от того, насколько грамотно используются ее возможности.

Нити, блоки, решетки, связки и ядра

На устройстве можно запустить вычислительный процесс, состоящий из большого числа параллельных нитей (threads). Нити объединяются в блоки (blocks), блоки образуют решетку (grid).

Решетка представляет собой одно-, двух- или трехмерный массив, состоящий из блоков (для старых GPU трехмерные массивы не поддерживаются). Блок представляет собой одно-, двух- или

трехмерный массив, состоящий из нитей. Размерность этих массивов задается в программе.

По техническим причинам нити выполняются связками (варпами, warps), обычно по 32 нити в связке, но для новейших GPU и по 64 нити. Все нити в связке одновременно выполняют одну и ту же операцию (но каждая со своими данными). Можно организовать вычисления только на одной нити, но GPU все равно запустит целую их связку. Остальные нити отработают те же действия, но результат не сохранится.

Параллельный вычислительный процесс на устройстве организуется при вызове ядра (kernel) . В C/C++ ядро оформляется как функция со специальным синтаксисом:

```
__global__ void имя ядра (формальные параметры) {
…
}
```

Спецификатор __global__ указывает, что функция выполняется на устройстве и вызывается с хоста.

При запуске ядра нужно указывать параметры параллельного выполнения. Обязательный минимум – размер решетки и размер блока. Например, так:

имя ядра<<<размер решетки, размер блока>>>(фактические параметры);

CUDA C поддерживает новые векторные типы данных, построенные на основе скалярных типов. Данные типа int3, uint3 и dim3 представляют собой структуры с полями x, y, z целого типа. При определении размера решетки и размера блока (тип dim3) обычно задают верхние границы для всех трех полей – например, dim3(128,256,32) . Пропущенные (справа) значения конструктор устанавливает равными единице. Так что dim(1024,1,1) , dim(1024) и 1024 – одно и то же.

Все нити параллельного процесса выполняют операции, перечисленные в ядре. Всем нитям доступны встроенные переменные gridDim и blockDim (тип uint3), их значения – заданные при вызове ядра размер решетки и размер блока. Встроенные переменные blockIdx и threadIdx (тип uint3) показывают индекс блока в решетке и индекс нити в блоке. Нумерация полей этих переменных начинается с нуля. Таким образом, каждая нить в блоке и каждый блок в решетке могут обращаться к разным элементам больших массивов. Размер связки содержится во встроенной переменной warpSize (тип int).

В следующем простом примере показано, как определить индекс блока и индекс нити. Хорошо, что CUDA разрешает нитям выводить данные на консоль. Но в старых версиях (например, CUDA 3) это не работает.

```
// индекс нити и индекс блока
#include <stdio.h>
__global__ void ker()
{
int k=blockIdx.x;
int j=threadIdx.x;
printf("block %i , thread %i : Hello!\n",k,j);
}
int main(){
ker<<<2,3>>>();
return 0;
}
```

Имеются ограничения на максимально возможные значения размеров решетки и блоков. Для достаточно мощной (в свое время) карты C1060 это (65535,65535,1) и (512,512,64). При этом общее количество нитей в блоке не может быть больше, чем 512. Возможные предельные случаи: или (512,1,1), или (64,8,1), или (8,8,8).

Для относительно новой видеокарты GeForce 940 MX, которая устанавливается сейчас в ноутбуках, максимальные значения индексов (2147483647,65537,64) и (1024,1024,64). В блоке может быть 1024 нити, в связке по-прежнему 32 нити.

Есть специальные функции cudaGetDeviceCount() и cudaGetDeviceProperties() , которые определяют количество графических устройств на компьютере и свойства каждого из них. Для каждого GPU определяется вычислительная производительность (compute capability) в виде пары целых чисел m.n . Этот параметр определяет, какие действия способен выполнить графический процессор и какой софт может работать с этим устройством. Например, CUDA 8 уже не поддерживает карты с производительностью ниже 2.0 .

Работа с массивами данных

Каждый массив представляет собой упорядоченное множество однотипных элементов. Позиция элемента в множестве определяется одним, двумя или более целочисленными индексами. Блоки из нитей и решетки из блоков – такие же массивы. Рассмотрим некоторые принципы, по которым устанавливают соответствие между индексами элементов массивов данных и индексами нитей в блоках и блоков в решетке.

По идеологии C/C++ каждый массив можно рассматривать как указатель. Заведем массивы на хосте

```
float *hA;
hA=(float*)malloc(n*sizeof(float));
```

и на устройстве

```
float *dA;
cudaMalloc((void**)&dA(n*sizeof(float));
```

Для пересылки данных используется функция cudaMemcpy(…) . В нашем случае данные с хоста на устройство пересылаются так:

```
cudaMemcpy(dA,hA,n*sizeof(float),cudaMemcpyHostToDevic
e);
```

Наверное, понятно, как вернуть результат с устройства на хост?

Для очистки памяти устройства вызывается функция cudaFree(…) .

Самое главное – решить, как построить решетку и блоки в ней. Количество m нитей в блоке нужно выбирать кратными 32 (это значение warpSize). Если количество n элементов в массиве кратно m , то блоков в решетке должно быть n/m . Ядро тогда нужно будет вызывать так:

```
ker<<<dim3(n/m),dim3(m)>>>(dA);
```

Это ядро может быть, например, таким:

```
__global__ void ker(float *a) {
  int j=blockIdx.x*blockDim.x+threadIdx.x;
  a[j]=__sinf(a[j]);
  };
```

Здесь вычисляются синусы элементов массива. Для этого используется функция __sinf(…) , работающая на GPU быстрее, чем обычная функция sin(…). Имеется целый набор подобных функций.

Хотя массив a будто бы и одномерный, но мы стали рассматривать его как двумерный. Первый индекс элемента – номер блока в решетке, второй индекс – номер нити в блоке. Чтобы использовать привычную нумерацию элементов одномерного массива, вычисляется значение переменной j – индекс нити в ядре.

Если же n не делится нацело на m , то блоков нужно завести n/m+1 и в ядре поставить оператор цикла while (j<n) …

Полный CUDA-код в данном случае следующий:

```
// jобработка массива
#include <stdio.h>

__global__ void ker(float *a, int n) {
   int j=blockIdx.x*blockDim.x+threadIdx.x;
   while (j<n) a[j]=__sinf(a[j]);
   };

int main() {
int n=1000, m=32;
float *da;
cudaMalloc((void**)&da, n*sizeof(float));
ker<<<dim3(n/m), dim3(m)>>>(da, n);

float *ha;
hA=(float*)malloc(n*sizeof(float));
cudaMemcpy(ha, da, n*sizeof(float),
                          cudaMemcpyDeviceToHost);
cudaFree(da);

for (int i=0; i<10; i++) printf("%Lf \n",ha[i]);
return 0;
}
```

Решетка может быть и двумерной. Число блоков представим в виде произведения p*q и укажем при вызове ядра размерность решетки dim3(p,q) . Тогда внутри ядра gridDim.x=p, gridDimy=q, и номер блока в линейном одномерном списке вычисляется следующим образом: blockIdx.x*gridDim.x+blockIdx.y или blockIdx.y*gridDim.y+blockIdx.x .

Оба варианта равноценны, они соответствуют размещению блоков в двумерном массиве или по строкам или по столбцам. Для данной простой задачи все равно.

Предположим теперь, что исходный массив был определен как двумерный: a[k,j], k=1..K, j=1..J . Если количество элементов этого массива не слишком велико, то можно сделать блоки минимальными – по 32 нити.

Пусть J кратно размеру блока: J=32*L . Тогда определим размерность решетки так: dim3(K,L,1) (K строк в массиве, L блоков в строке) и размерность блока: dim3(32,1,1) .

В этом случае индексы элемента массива в ядре можно легко вычислить:

```
k=blockIdx.x+1;
j=blockInd.y*blockDim.x+threadIdx.x+1;
```

Заметим, что блоки выполняются на GPU вовсе не одновременно, а по мере освобождения ресурсов. Многие функции CUDA, в том числе и ядра, являются асинхронными – управление в вызывающую их функцию возвращается раньше, чем была закончена операция. Если нужно дождаться, когда завершатся все блоки решетки, используют функцию __syncthreads(); (барьерная синхронизация).

Виды памяти CUDA

Технология CUDA примечательна тем, что можно использовать разные виды памяти для хранения данных. Выбор памяти существенно влияет на эффективность программы.

Память хоста – это оперативная память DRAM на материнской плате. Область видимости – всюду, время жизни данных устанавливается динамически, например, с помощью функций malloc() и free() (есть другие способы).

На устройстве поддерживаются разные виды памяти, не все из них доступны программисту.

Глобальная память – это основной вид памяти в DRAM устройства, используется по умолчанию. Ее объем обычно от 0.5 Гб до 4.0 Гб, в зависимости от типа видеокарты.

Данные в памяти размещаются, как правило, динамически, с помощью функции cudaMalloc() . Память освобождается с помощью функции cudaFree() . Есть возможность определять и статические данные. Как уже было сказано, обмен данными между памятью хоста и глобальной памятью устройства организуется при вызовах функции cudaMemcpy() .

Глобальная память доступна любой нити ядра для чтения и записи. Но память эта относительно медленная (имеет высокую латентность). Кроме того, для оптимального ее использования нужно использовать специальные приемы, о которых речь пойдет в следующем разделе.

Непосредственно в кристалле GPU размещены регистры, которые распределяются между нитями во время компиляции. Каждая нить имеет доступ только к своим регистрам. Это самый быстрый вид памяти. Регистры имеют размер в 32 бита, их общее количество от 256 до 2048 (или даже больше). Число блоков, которые могут одновременно выполняться на GPU, определяется по количеству регистров. Если регистров все-таки недостаточно, в DRAM устройства организуется локальная память. Ее быстродействие существенно ниже. Некоторые типы данных также размещаются в локальной памяти. Поэтому задача программиста – постараться свести к минимуму использование локальной памяти (хорошо бы понять, как это делается).

Для ускорения вычислений используется быстрая разделяемая память, она выделяется блокам поровну (от 16 до 48 Кб). Через разделяемую память передаются параметры при запуске ядра. Кроме того,

в ней выгодно хранить данные, которые часто используются.

Есть два стандартных способа размещения данных в разделяемой памяти. Первый вариант:

```
__global__ void ker1(float* a) {
  __shared__ float buf[256];
  buf[threadIdx.x]=a[blockIdx.x*blockDim.x+threadIdx.x];
  … …
  };
```

В этом случае выделяется 256x4 байта на блок.

Второй вариант:

```
__global__ void ker2(float* a) {
  __shared__ float buf[];
  buf[threadIdx.x]=a[blockIdx.x*blockDim.x+threadIdx.x];
  …
  };
```

Здесь каждому блоку выделяется дополнительно разделяемая память и в ней, начиная с 1-го байта, будет размещен массив !

Сколько же разделяемой памяти выделено? Это определяет третий параметр параллельного выполнения при вызове ядра. Например, так:

```
ker2<<<dim3(N/256),dim3(256),k*sizeof(float))>>>(b);
```

Константная память также быстрая. Хотя она находится в DRAM GPU, но кэшируется. В этой памяти целесообразно хранить небольшие объемы данных, которые часто используются всеми нитями. Данные в константную память засылаются из памяти хоста. Например, так:

```
__constant__ float cData[256];
float hData[256];
…
```

```
cudaMemcpyToSymbol(cData, hData, sizeof(cData), 0,
      cudaMemcpyHostToDevice);
```

Все нити могут только читать данные из константной памяти.

Наконец, есть еще один вид памяти устройства – текстурная память. Этот вид памяти предназначен для работы с изображениями, но и для вычислений он иногда используется.

Еще раз о редукции

Рассмотрим следующий пример [7], в котором содержится много полезных приемов:

```
// скалярное произведение векторов
#include <stdio.h>
#define imin(a,b)(a<b?a:b)
const int N=33*1024;
const int TpB=256;
const int BpG=imin(32,(N+TpB-1)/TpB);

__global__ void scpr(float *a,float *b,float *c)
 {
 __shared__ float cache[TpB];
 int nt=blockDim.x*blockIdx.x+threadIdx.x;
 int nc=threadIdx.x;
 float sum=0;
 while (nt<N){
   sum+=a[nt]*b[nt];
   nt+=blockDim.x*gridDim.x;
   };
 cache[nc]=sum;
 __syncthreads();
 int k=blockDim.x/2;
```

```
  while (k!=0){
    if (nc<k) cache[nc]+=cache[nc+k];
    __syncthreads();
    k/=2;
    };
  if (nc==0) c[blockIdx.x]=cache[0];
  }

int main()
  {
  float *a,*b,c,*pc;
  float *da,*db,*dpc;
  a=(float*)malloc(N*sizeof(float));
  b=(float*)malloc(N*sizeof(float));
  pc=(float*)malloc(BpG*sizeof(float));
  for (int j=0;j<N;j++){ a[j]=j*2; b[j]=j/2; };
  cudaMalloc((void**)&da,N*sizeof(float));
  cudaMalloc((void**)&db,N*sizeof(float));
  cudaMalloc((void**)&dpc,BpG*sizeof(float));
  cudaMemcpy(da,a,N*sizeof(float),
        cudaMemcpyHostToDevice);
  cudaMemcpy(db,b,N*sizeof(float),
        cudaMemcpyHostToDevice);
  scpr<<<BpG,TpB>>>(da,db,dpc);
  cudaMemcpy(pc,dpc,BpG*sizeof(float),
        cudaMemcpyDeviceToHost);
  cudaFree(dpc);   cudaFree(db);  cudaFree(da);
  c=0;
  for (int j=0;j<BpG;j++) c+=pc[j];
  printf("%f\n",c);

  return 0;
  }
```

Здесь вычисляется скалярное произведение двух длинных векторов. В решетке создается 256 блоков. В разделяемой памяти каждому блоку выделяется вспомогательный массив cache[] для хранения сумм произведений элементов перемножаемых массивов.

Вызов функции __syncthreads(); нужен для того, чтобы все нити успели завершить вычисления до начала следующей операции.

Затем вычисляется сумма элементов массивов cache[] . Чтобы ускорить процесс, массив условно делится на две части и к элементам первой части добавляется по одному элементу второй части. В итоге нужные данные остаются в массиве вдвое меньшей длины. Этот массив также условно делится на две части … и так до тех пор, пока сумма всех элементов массива не соберется в его нулевом элементе. Этот процесс также называют редукцией.

Нулевые элементы массивов cache[] пересылаются на хост в массиве c[] и, наконец, вычисляется сумма элементов этого массива.

Может быть это не так красиво, как в MPI, но все равно хорошо работает.

Оптимизация параллельного кода

Прежде всего, нужно убедиться, что программа работает без ошибок. Все функции из библиотеки CUDA (кроме ядер) возвращают значение типа cudaError_t . Если функция завершилась успешно, то это значение – cudaSuccess , иначе возвращается код ошибки. По этому коду можно получить описание ошибки в виде короткого текста. Для этого используется функция

```
char* cudaGetErrorString(cudaError_t code);
```

Вряд ли имеет смысл проверять при каждом обращении к одной из функций CUDA, не случилось ли чего. Можно получить код последней ошибки, если обратиться к функции

```
cudaError_t GudaGetLastError();
```

Так обычно и поступают при отладке кода.

Теперь еще раз зададим себе вопрос: что считать оптимальной программой? Критериев может быть несколько. При параллельном программировании самое существенное – время выполнения программы. Если ресурсы компьютера ограничены, то нужно заботиться об экономии памяти. Наконец, определенную роль играет и «человеческий фактор» – насколько сложно программисту освоить новую технологию и оправдывают ли ожидания затраты на разработку параллельной программы.

Если на первый план ставится ускорение счета, то нужно научиться определять время выполнения программы.

Для измерения времени выполнения операций на устройстве, включая обмен данными между памятью хоста и памятью устройства, используется механизм событий CUDA.

События (events) – это специальные объекты, в которых можно сохранить показания системного таймера. Сравнивая две такие временные метки, можно определить заключенный между ними отрезок времени.

Обычно используют следующую конструкцию:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);
… … …
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
float elapTime;
cudaEventElapsedTime(&elapTime,start,stop);
printf("ElapsedTime : %3.1f  ms \n",elapTime);
cudaEventDestroy(stop);
cudaEventDestroy(start);
```

Время измеряется в миллисекундах.

При размещении данных в глобальной памяти их адреса выравниваются на блоки по 256 байт, то есть объем выделяемых участков кратен 256 (нумерация начинается с нуля). При обращении к глобальной памяти чтение и запись данных происходит словами длиной 4 байта, 8 или 16 байтов. Поэтому имеет смысл для всех адресов данных предусмотреть выравнивание по размеру слова.

Рассмотрим типичный пример. Пусть объявлена структура

```
struct vec3 {
  float x,y,z;
}
```

Это плохо, так как данные такого типа занимают в памяти 12 байтов. Если разместить в памяти массив из таких элементов, то адрес первого элемента (точнее, нулевого) будет кратен 256. Но адрес следующего элемента не делится на 8. Поэтому его чтение или запись будут проходить в два этапа.

Для ускорения рекомендуется добавить в структуру лишнее поле (и не пользоваться им). Это можно сделать двумя способами: или

```
struct vec4 {
  float x,y,z,u;
}
```

или

```
struct __align__(16) vec3 {
  float x,y,z;
}
```

Объектное программирование

Технология CUDA допускает использование классов и структур с функциями как в коде для CPU (что очевидно), так и в коде для GPU (это иногда отрицается в руководствах). Если на хосте для работы с

переменными комплексного типа удобно пользоваться типом данных вида

```
struct hComp
 {
 float re, im;
 hComp(float re_, float im_): re(re_), im(im_){};
 hComp operator+(const hComp& c){
  return hComp(re+c.re, im+c.im);}
 };
```

то его аналогом для устройства является

```
struct dComp
 {
 float re, im;
 __device__ dComp(float re_, float im_): re(re_), im(im_){};
 __device__ dComp operator+(const dComp& c){
  return dComp(re+c.re, im+c.im);}
 };
```

Отличие только в том, что для объектных типов на устройстве нужно указывать, что операции над данными должны быть выполнены именно на устройстве.

Предлагаем читателю самостоятельно расширить список возможных операций на комплекснозначными переменными.

Следующая программа демонстрирует, что два структурных типа совместимы:

```
// комплексные переменные
#include <stdio.h>
struct hComp
 {
 float re, im;
 hComp(float re_, float im_): re(re_), im(im_){};
 hComp operator+(const hComp& c){
```

```
    return hComp(re+c.re, im+c.im);}
  };
struct dComp
  {
  float re, im;
  __device__ dComp(float re_, float im_): re(re_), im(im_){};
  __device__ dComp operator+(const dComp& c){
    return dComp(re+c.re, im+c.im);}
  };
__global__ void ker1(dComp* C)
  {
  int j=blockIdx.x*blockDim.x+threadIdx.x;
  C[j]=C[j]+dComp(1,1);
  }
int main()  {
hComp a(1,2), b(3,7);
a=a+b;
printf("%Lf,%Lf\n",a.re,a.im);
int n=32000;
hComp *hC;
hC=(hComp*)malloc(n*sizeof(hComp));
for (int j=0; j<n; j++) hC[j]=hComp(1,2);
printf("%Lf,%Lf\n",hC[1].re,hC[1].im);
dComp *dC;
cudaMalloc((void**)&dC,n*sizeof(hComp));
cudaMemcpy(dC,hC,n*sizeof(hComp),
                                cudaMemcpyHostToDevice);
ker1<<<1000,32>>>(dC);
cudaMemcpy(hC,dC,n*sizeof(hComp),
                                cudaMemcpyDeviceToHost);
printf("%Lf,%Lf\n",hC[1].re,hC[1].im);
return 0;
}
```

Литература

1. Максимов Н.В., Партыка Т.Л., Попов И.И. Архитектура ЭВМ и вычислительных систем. – М.: ФОРУМ: ИНФРА-М, 2005. – 512 с.

2. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.

3. Левин М.П. Параллельное программирование с использованием OpenMP: учебное пособие. – М.: Интернет-Университет Информационных Технологий; БИНОМ: Лаборатория знаний, 2012. – 118 с.

4. Корнеев В.Д. Параллельное программирование в MPI. – Москва-Ижевск: ИКИ, 2003. – 304 с.

5. Гришагин В.А., Свистунов А.Н. Параллельное программирование на основе MPI. Учебное пособие. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2005. – 93 с.

6. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI. – Минск: Изд-во БГУ, 2002. – 323 с.

7. Сандерс Дж., Кэндрот Э. Технология CUDA в примерах: введение в программирование графических процессоров. – М.: ДМК Пресс, 2011. – 232 с.

8. Параллельные вычисления на GPU. Архитектура и программная модель CUDA: Учебное пособие / А.В. Боресков и др. – М.: Изд-во Москов. ун-та, 2012. – 336 с.

9. Технология программирования CUDA: учеб. пособие / Д.Н. Тумаков и др. – Казань:Изд-во Казан. ун-та, 2017. – 112 с.

Содержание

Введение 4
Архитектура и классификация МВК 8
Hard и Soft 13
Технология OpenMP 19
Первые шаги 20
Нумерация нитей 21
Общие и частные данные 23
Распараллеливание циклов 26
Что такое «редукция»? 29
Параллельные секции 31
Синхронизация 33
Переменные окружения 34
Оптимизация параллельного кода 35
Еще два примера 36
Технология MPI 40
Введение в MPI 40
Компиляция и запуск 43
Параллельная обработка массивов 45
Простейший обмен сообщениями 47
Возможные ошибки 49
Неблокирующие функции обмена 51
Коллективный обмен данными 53
Задача Дирихле для уравнения Лапласа 56
Еще немного об MPI 60
Технология CUDA 61
Инсталляция CUDA 61
Нити, блоки, решетки и связки 63
Работа с массивами данных 66
Виды памяти CUDA 69
Еще раз о редукции 72
Оптимизация параллельного кода 74
Объектное программирование 76
Литература 79