МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Чувашский государственный университет имени И.Н. Ульянова»

А.Г. Алексеев, А.Р. Йовенко

Параллельное программирование

Учебное пособие

Чебоксары 2015

УДК 004.451:004.4(076.5)
ББК 3973.2–018я73
С38

Рецензенты:

Н.В. Софронова – д-р пед. наук, профессор (кафедра информатики и вычислительной техники Чувашского государственного педагогического университета);

М.В. Киселев – канд. техн. наук, генеральный директор ООО "Компания Мегапьютер Интеллидженс"

Алексеев А.Г., Йовенко А.Р.

А46 Параллельное программирование: учеб. пособие / А.Г. Алексеев, А.Р. Йовенко – Чебоксары: Изд-во Чуваш. ун-та, 2015. – 194 с.

ISBN 978-5-7677-2162-7

Рассмотрены основные принципы организации параллельных вычислений в моделях с разделяемой и распределенной памятью и гетерогенные параллельные вычисления.

Приведены базовые алгоритмы параллельных вычислений (сортировка и алгоритмы на графах), а также примеры адаптации существующих последовательных алгоритмов в их параллельные аналоги.

Для студентов IV курса направления подготовки бакалавров 230100 «Информатика и вычислительная техника», изучающих дисциплину «Параллельное Программирование».

Работа выполнена при поддержке федеральной программы «Кадры для регионов».

Ответственный редактор канд. техн. наук, доцент А.А. Павлов

Утверждено Учебно - методическим советом университета

ISBN 978-5-7677-2162-7

УДК 004.451:004.4(076.5)
ББК 3973.2–018я73
© Издательство
Чувашского университета, 2015
© Алексеев А.Г.,
Йовенко А.Р. 2015

ПРЕДИСЛОВИЕ

Идея параллельных вычислений возникла практически одновременно с появлением электронно-вычислительных машин. Первые шаги в этом направлении были сделаны еще в конце 50-х годов прошлого века. Первоначально эти идеи реализовались лишь в передовых для своего времени суперкомпьютерах, а сегодня уже не редкость применение многоядерного процессора в сотовом телефоне. Большинство серийно выпускаемых процессоров для персональных ЭВМ также являются многоядерными и было бы расточительно не воспользоваться предоставляемыми вычислительными мощностями.

Параллельные вычисления имеют свою специфику и создание эффективных параллельных программ невозможно без изучения основных принципов их построения и работы. Эти знания являются необходимыми для современного специалиста в области информационных технологий. Математическое моделирование, искусственный интеллект, статистическая обработка данных, компьютерная обработка изображений и видео – вот лишь несколько областей, где использование параллельных вычислений помогает решать задачи более эффективно.

Цель учебного пособия – познакомить читателя с удивительным и разнообразным миром параллельных вычислений – от написания “простых” многопоточных программ до организации вычислений на больших кластерах. Рассмотрена организация параллельных вычислений с использованием графических адаптеров, популярность которых значительно выросла за последние пять лет.

Пособие представляет собой полный комплекс лабораторных работ для студентов, изучающих дисциплину “Параллельное программирование”. В индивидуальной работе студентов особое внимание уделено оценке эффективности разработанных программ.

Примеры программ к данному учебному пособию доступны по адресу: https://github.com/huntercbx/parallel-programming.

ОБЩИЕ ТРЕБОВАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

Приводимые требования являются общими для всех лабораторных работ, и если в требованиях к конкретной лабораторной работе не указано иное, следует придерживаться их.

1. Время выполнения параллельной программы зависит от многих факторов, в том числе от используемого оборудования и программного обеспечения, поэтому в отчете должна быть указана следующая информация:

– тип используемого центрального процессора (количество ядер, размер кэша);

– объем и тип используемой памяти;

– версия установленной операционной системы;

– используемый компилятор;

– версия используемых библиотек (если таковые имеются);

– дополнительные сведения в зависимости от лабораторной работы.

2. При оценке времени выполнения программы рекомендуется придерживаться следующих правил:

– завершить работу всех программ, работа которых не является необходимой для запуска и работы оцениваемой программы;

– не следует оценивать время при работе программы в режиме отладки, так как скорость работы программы в этом случае может снижаться в несколько раз;

– при работе в среде MicrosoftVisualStudio необходимо использовать release-версии программ для оценки производительности;

– если входные данные загружаются из входного файла или генерируются автоматически, то время загрузки/генерации данных следует измерить отдельно – обычно эта часть программы выполняется последовательно, что следует учитывать при оценке выигрыша, полученного от параллелизма;

– иногда время работы алгоритма зависит не только от объема данных, но и от самих данных (например, время работы некоторых алгоритмов сортировки сильно зависит от первоначального порядка сортируемых элементов), поэтому при оценке

времени работы такой программы необходимо провести измерения на нескольких наборах входных данных одинакового объема и взять среднее значение;

– как правило, используемые библиотеки имеют встроенные средства для оценки времени выполнения участков кода и там, где это возможно, следует использовать их;

– при анализе результатов необходимо учесть, что измерения времени обладают некоторой погрешностью, обусловленной разрешением используемого таймера; в большинстве случаев разрешение таймера можно получить при помощи специальной функции или эмпирически – его также следует включить в отчет;

– чтобы оценить выигрыш от использования параллелизма, рекомендуется выбирать объем данных таким образом, чтобы время выполнения последовательной версии программы занимало от 20 с до 10 мин (если это не оговорено специально в требованиях к лабораторной работе).

3. При анализе времени выполнения программы данные измерений рекомендуется представлять одновременно в табличном виде и при помощи диаграмм (круговых или столбчатых): таблицы удобны для изучения точных значений, а диаграммы позволяют представить данные более наглядно для приблизительных оценок.

4. При использовании генератора случайных чисел необходимо учесть следующие моменты:

– если входные данные генерируются автоматически (при помощи генератора случайных чисел), то следует запоминать значение для инициализации генератора псевдослучайных чисел (seed-number), чтобы иметь возможность повторить испытание с теми же данными;

– при использовании генератора псевдослучайных чисел необходимо учитывать его разрядность – для некоторого класса задач появление большого количества повторяющихся значений (в случае исходных данных большого объема) может негативно сказаться на достоверности полученных результатов; решением проблемы является использование генератора с большой разрядностью;

– использование генератора псевдослучайных чисел из нескольких потоков в многопоточной программе может потребовать сохранения в общей памяти последнего полученного случайного числа и использования его для инициализации генератора при следующем вызове.

5. При использовании внешних данных рекомендуется использовать простые и стандартные форматы файлов с данными:

– для табличных данных рекомендуется воспользоваться форматом CSV (Comma Separated Values);

– для растровых графических изображений хорошо подойдут форматы BMP (Device Independent Bitmap) и PPM (Portable Pixel Map).

6. При сравнительном анализе времени работы параллельной и последовательной версий программы следует учитывать, что большинство современных процессоров могут изменять тактовую частоту одного или нескольких ядер в случае, если другие ядра не загружены. В результате последовательная версия программы может выполняться на несколько десятков процентов быстрее.

Данная технология от Intel называется Intel Turbo Boost и применяется в процессорах Intel Core i5 и Core i7. Для отслеживания ее работы можно использовать специальное ПО, например, Intel Turbo Boost Technology Monitor. Схожая технология от AMD носит название AMD Turbo Core и впервые появилась в шестиядерных процессорах Phenom II X6.

Лабораторная работа 1

НАКЛАДНЫЕ РАСХОДЫ НА ПАРАЛЛЕЛИЗМ

Цели работы

Изучение архитектуры многопоточных программ с разделяемой памятью, создаваемых при помощи OpenMP.

Знакомство с ситуацией “гонка” и базовыми механизмами синхронизации для ее предотвращения.

Оценка накладных расходов на параллелизм и выигрыша от выполнения программы в параллельном режиме.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с многоядерным процессором (желательно наличие не менее четырех ядер);

– установленная ОС Windows или Linux;

– компилятор для языка C/C++ с поддержкой OpenMP.

В качестве компилятора могут использоваться GNU GCC версии 4.7 и выше (под ОС Windows рекомендуется использовать пакет MinGW) либо MS Visual Studio 2008 и более поздние версии (не все версии MSVS имеют поддержку OpenMP, более подробную информацию можно получить на сайте производителя).

Общие сведения об OpenMP

OpenMP (Open MultiProcessing) представляет собой простой и гибкий инструментарий для создания многопоточных программ. Большинство современных компиляторов для языков программирования Fortran и C/C++ имеют встроенную поддержку OpenMP.

За развитие стандарта OpenMP отвечает некоммечерская организация, в состав которой входят ведущие разработчики аппаратного и программного обеспечения. Стандарт является открытым и доступен на официальном сайте http://openmp.org/. Там же можно найти дополнительные материалы, включая примеры программ.

В качестве модели выполнения используется модель Fork & Join. В этой модели программа состоит из чередующихся секций последовательного и параллельного выполнения. На протяжении всей работы работы программы существует основной поток (master thread), который имеет порядковый номер 0. Более наглядно модель выполнения представлена на рис. 1.1.

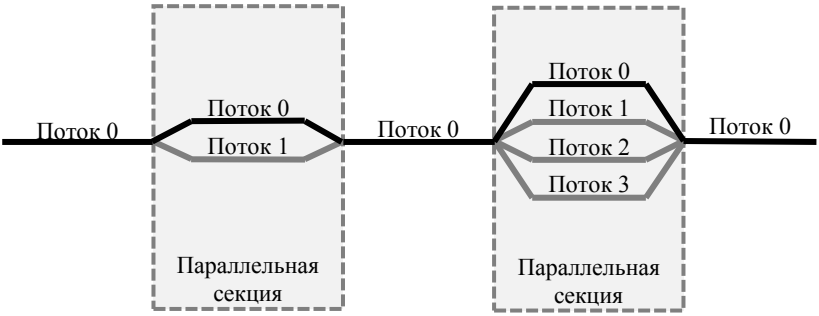


Рис. 1.1. Модель выполнения “Fork & Join” в OpenMP

Каждый поток может обращаться к разделяемой (общей) памяти, доступной для всех потоков. У каждого потока также имеется свой стек и область памяти, недоступная другим потокам (рис. 1.2).

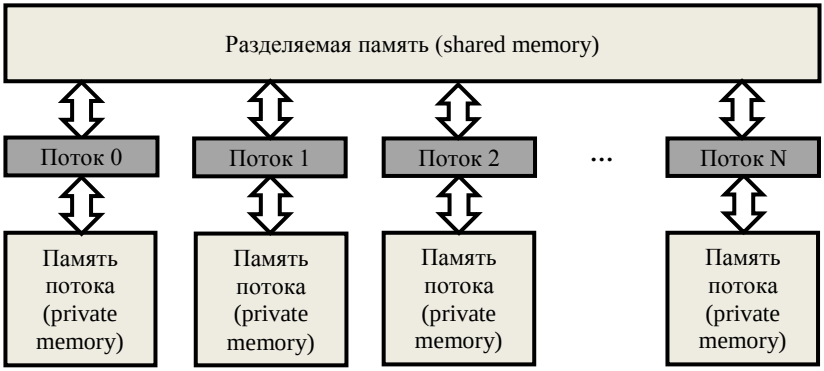


Рис. 1.2. Модель памяти в OpenMP

Пример 1.1. Hello World

Рассмотрим одну из простейших программ, демонстрирующих работу в многопоточном режиме с использованием OpenMP. Данная программа выводит строки “Hello World” из разных потоков и является своего рода многопоточной версией классической программы HelloWorld.

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

int main(int argc, char *argv[])
{
   int th_id, nthreads;
   #pragma omp parallel \
      private(th_id) \
      shared(nthreads) \
      num_threads(10)
   {
      th_id = omp_get_thread_num();
      printf("Hello World from thread %d\n",
         th_id);

      #pragma omp barrier

      #pragma omp master
      // if (th_id == 0)
      {
      nthreads = omp_get_num_threads();
         printf("There are %d threads (thread
%d)\n", nthreads, th_id);
      }
   }
   return 0;
}
```

Ключевые моменты в программе отмечены жирным шрифтом. Прежде всего нам необходимо подключить заголовочный файл **<omp.h>**, где объявлены функции **omp_get_thread_num** и **omp_get_num_threads**. Первая функция возвращает номер потока, основному потоку соответствует номер 0. Вторая – возвращает количество работающих потоков. Помимо функций в данной программе используются три директивы компилятора.

Директива **#pragma omp parallel** сообщает компилятору, что следующий за этой директивой блок должен выполняться параллельно. В нашем примере мы используем три дополнительных параметра для этой директивы. Первые два определяют, будет ли объявленная выше переменная общей для всех по-

токов или же она будет принадлежать потоку. В этом случае для каждого потока будет создана своя копия переменной в локальной памяти потока. Последний параметр определяет количество создаваемых потоков, он не является обязательным, и если его не указать, то будет создано количество потоков, равное количеству доступных процессоров (ядер).

Директива **#pragma omp barrier** является простейшим средством синхронизации и приостанавливает выполнение других потоков до тех пор, пока все потоки не дойдут до этой точки программы. В нашей программе это необходимо, чтобы все потоки успели вывести строку “Hello World” до вывода строки с количеством созданных потоков.

Директива **#pragma omp master** сообщает компилятору о том, что следующий за ней блок должен выполняться только основным потоком программы. Вместо директивы компилятора можно было бы написать условие **if (th_id == 0)**, но использование директивы более наглядно. Существует схожая директива **#pragma omp single**, однако разница состоит в том, что в последнем случае блок будет выполнен любым потоком.

При выполнении данной программы мы получим следующие результаты:

```
K:\OpenMP examples>HelloWorld.exe
Hello World from thread 2
Hello World from thread 0
Hello World from thread 4
Hello World from thread 6
Hello World from thread 8
Hello World from thread 1
Hello World from thread 3
Hello World from thread 5
Hello World from thread 7
Hello World from thread 9
There are 10 threads (thread 0)
```

Результаты работы показывают, что порядок, в котором выполняются потоки, не определен. Несмотря на то, что в OpenMP существуют механизмы для управления порядком выполнением потоков, в общем случае не следует полагаться на порядок выполнения отдельных потоков.

Пример 1.2. Перемножение матриц

В этом примере мы рассмотрим применение OpenMP для параллельного выполнения циклов. Для этого идеально подходит задача параллельного перемножения матриц.

В качестве входных данных мы будем использовать две матрицы A и B размерностью N×N элементов. Напомним, что сложность задачи перемножения таких матриц оценивается как $O(N^3)$.

Фрагмент последовательной программы перемножения матриц мог бы выглядеть следующим образом:

```
// N - размерность матриц
// A, B - исходные квадратные матрицы
// C - результирующая матрица
for (size_t i = 0; i < N; ++i)
{
   for (size_t j = 0; j < N; ++j)
   {
      C[i][j] = 0;
      for (size_t k = 0; k < N; ++k)
         C[i][j] += A[i][k] * B[k][j];
   }
}
```

В задаче умножения матриц отдельные итерации циклов по i и j можно выполнять параллельно, так как результат их выполнения не зависит от предыдущих итераций. Для того чтобы цикл выполнялся параллельно, достаточно внести небольшую модификацию в программу.

Для параллельного выполнения итераций цикла в OpenMP используется директива **#pragma omp parallel for**. Если директива используется без параметров, то все N итераций цикла будут равномерно распределены между P потоками. Первый поток будет отвечать за выполнение итераций с 0 по N/P-1, второй – за итерации в диапазоне от N/P до 2N/P-1 и т.д.

Ниже приведен фрагмент программы с параллельным выполнением цикла по i.

```
#pragma omp parallel for
for (size_t i = 0; i < N; ++i)
{
   for (size_t j = 0; j < N; ++j)
   {
      C[i][j] = 0;
      for (size_t k = 0; k < N; ++k)
         C[i][j] += A[i][k] * B[k][j];
   }
}
```

Теперь попробуем добиться параллельного выполнения цикла по k. Попробуем просто выполнять итерации циклов параллельно так же, как мы делали это раньше:

```
for (size_t i = 0; i < N; ++i)
{
   for (size_t j = 0; j < N; ++j)
   {
      C[i][j] = 0;
      #pragma omp parallel for
      for (size_t k = 0; k < N; ++k)
         C[i][j] += A[i][k] * B[k][j];
   }
}
```

В этом случае можно заметить, что программа будет работать неправильно. В этом легко убедиться, если подать на вход программы матрицы, в которых все элементы равны 1, тогда в результирующей матрице значения элементов должны быть равны N. Ниже приведен результат работы такой программы для N = 100 (чем больше N, тем вероятнее проявление ошибки), показаны только элементы, отличающиеся от N:

```
C[59][17] = 50.0
C[60][29] = 95.0
C[60][34] = 94.0
C[84][28] = 97.0
C[84][53] = 93.0
C[85][ 6] = 96.0
C[89][ 8] = 93.0
C[91][54] = 99.0
```

То, что произошло в нашем примере, называется ситуацией “гонки” (race condition). При ее возникновении потоки конкурируют между собой за доступ к общим ресурсам, и результат работы параллельной программы зависит от того, в какой последовательности выполнялись параллельные секции.

В данном случае ошибка происходит при выполнении операции **`C[i][j] += A[i][k] * B[k][j]`**. Фактически при выполнении этой операции выполняются следующие операции:

1) выполняется умножение элементов **`A[i][k]*B[k][j]`**;

2) из памяти считывается прежнее значение **`C[i][j]`**;

3) к прочитанному значению прибавляется результат умножения;

4) новое значение записывается по адресу **`C[i][j]`**.

Одновременное выполнение операций 2-4 из разных потоков и приводит к ошибке.

Существует несколько вариантов для решения данной проблемы. Мы рассмотрим только один из них – редукцию (reduction). Редукцией называется операция сведения результатов работы отдельных потоков для получения итогового результата. В нашем примере каждый поток мог бы локально накапливать сумму произведений элементов, которая при завершении работы потока безопасно объединилась бы с результатами других потоков. Приведенный ниже фрагмент программы обеспечивает правильную работу алгоритма при помощи редукции с использованием операции “+” по переменной “sum”:

```
for (size_t i = 0; i < N; ++i)
{
   for (size_t j = 0; j < N; ++j)
   {
      double sum = 0;
      #pragma omp parallel for \
      reduction(+:sum)
      for (size_t k = 0; k < N; ++k)
         sum += A[i][k] * B[k][j];
      C[i][j] = sum;
   }
}
```

Полный перечень допустимых операций редукции приведен в табл. 1.1.

Таблица 1.1

Допустимые операции редукции в OpenMP

Операция	Начальное значение переменной
+	0
-	0
*	1
&	~0
|	0
^	0
&&	1
||	0
min	максимальное значение для типа
max	минимальное значение для типа

Пример 1.3. Подсчет количества простых чисел в указанном диапазоне

Рассмотрим задачу подсчета количества простых чисел в заданном диапазоне [min_number, max_number].

Согласно определению, натуральное число считается простым, если оно делится без остатка только на два различных натуральных числа: на единицу и на само число. Отметим также, что единица не считается простым числом.

Для определения является ли число N простым, мы будем проверять остатки от деления N на все числа в диапазоне от 2 до квадратного корня от самого числа N. Если какой-либо остаток от деления равен нулю, то число N не является простым. Данный метод называется методом перебора делителей.

Предложеный метод является самым простым, но имеет очень высокую вычислительную сложность. Это ограничивает его применение на практике, но хорошо подходит для демонстрации особенностей параллельного программирования.

Последовательная версия программы на C++, реализующая данный алгоритм, может выглядеть следующим образом:

```
// количество найденных простых чисел
longn_prime_numbers = 0;

for (long i = min_number; i < max_number; ++i)
{
   bool is_prime = true;
   long last = (long)floor(sqrt(i));
   for (long j = 2; j <= last && is_prime; ++j)
   {
      if ((i % j) == 0)
         is_prime = false;
   }

   if (is_prime)
      ++n_prime_numbers;
}

// вывод количества найденных простых чисел
printf("Found %dprimenumbers\n",
   n_prime_numbers);
```

Для диапазона [2, 1000] существует 168 простых чисел. Именно такой результат мы и получим на выходе нашей программы.

Теперь преобразуем нашу последовательную программу в параллельную, для этих целей будем выполнять итерации цикла параллельно так же, как мы это делали в примере с перемножением матриц. Все, что от нас потребуется – это добавить одну строку в программу:

```
// количество найденных простых чисел
long n_prime_numbers = 0;

#pragma omp parallel for
for (long i = min_number; i < max_number; ++i)
{
    ...

   if (is_prime)
      ++n_prime_numbers;
}
```

Запустив программу несколько раз на том же диапазоне [2, 1000], мы с удивлением обнаружим, что теперь результат программы меняется с каждым запуском – теперь программа находит 165, 167 или 168 простых чисел (результаты запуска программы сильно зависят от аппаратной платформы и программного окружения и могут отличаться от указанных). Что же повлияло на результат работы программы?

Точно так же, как и в примере с умножеием матриц, мы столкнулись с “гонкой” – потоки конкурируют между собой при обновлении переменной n_prime_numbers. Операция инкремента не является атомарной и ее лучше рассматривать как последовательность из трех операций:

– чтения значения переменной из памяти;

– прибавления 1 к прочитанному значению;

– запись нового значения в память.

Представим ситуацию, когда два потока хотят произвести операцию инкремента одновременно (очень близко по времени). Если второй поток прочитал значение переменной после того, как первый поток записал его, то значение переменной в итоге будет увеличено на 2, как и ожидалось. Однако, если второй поток прочитает значение переменной до того, как первый поток успеет его обновить, то значение переменной будет увеличено только на 1.

Таким образом, ситуация “гонки” возникает тогда, когда отсутствует синхронизация при доступе к общему ресурсу.

В OpenMP предусмотрены следующие механизмы синхронизации:

– барьер синхронизации,

– критическая секция,

– блокировка (мьютекс),

– атомарные операции.

Барьер синхронизации не подходит для данного случая, поэтому мы его рассматривать не будем.

Критическая секция гарантирует, что данный участок кода выполняется в каждый момент времени только одним потоком. Фрагмент программы для нашей задачи мог бы выглядеть следующим образом:

```
if (is_prime)
{
   #pragma omp critical
   {
      ++n_prime_numbers;
   }
}
```

Несмотря на всю простоту решения, вариант с критической секцией может существенно сказаться на производительности, так как выполнение потока, который не смог войти в критическую секцию, будет приостановлено на некоторое время (которое зависит от операционной системы).

Для того чтобы не приостанавливать выполнение потока, желающего получить доступ к переменной n_prime_numbers, можно воспользоваться ждущими блокировками. Ниже приведен фрагмент получившегося кода.

```
// объявление и инициализация блокировки
omp_lock_tlock;
omp_init_lock(&lock);

#pragma omp parallel for
for (long i = min_number; i < max_number; ++i)
{
...

   if (is_prime)
   {
      // ожидание захвата блокировки
      while (!omp_test_lock(&lock)) {}

      // выполнение операции
      ++n_prime_numbers;

      // освобожение блокировки
      omp_unset_lock(&lock);
   }
}
// удаление блокировки
omp_destroy_lock(&lock);
```

До начала параллельной секции нужно объявить и проинициализировать связанную с блокировкой переменную. Для того, чтобы поток не блокировался при неудавшемся захвате блокировки, мы организуем пустой цикл ожидания, единственной операцией в цикле будет попытка захвата блокировки. Цикл будет выполняться до тех пор, пока операция не завершится успешно. После захвата блокировки мы выполняем все необходимые операции и освобождаем блокировку. По завершении параллельной секции блокировку можно удалить.

В отличие от критической секции при использовании ждущей блокировки поток продолжает выполняться, даже если доступ к ресурсу получить не удалось. Он находится в цикле ожидания и продолжит свою работу сразу после освобождения ресурса. Именно поэтому такая блокировка называется ждущей.

Отметим, что использование блокировок потребовало внесения довольно больших изменений в программу, что не оправдано для такой элементарной операции, как инкремент. Поэтому идеальным будет вариант с атомарной операцией. Атомарная операция гарантирует, что ее выполнение не будет прервано другим потоком.

```
...

if (is_prime)
{
   #pragma omp atomic
   ++n_prime_numbers;
}
...
```

Общий синтаксис атомарных операций в OpenMP имеет следующий вид (если тип операции не указан, как в нашем примере, то он эквивалентен update):

```
#pragma omp atomic [read|write|update|capture]
выражение
#pragma omp atomic capture
структурныйблок
```

Разрешенные виды выражений в зависимости от типа операции приведены в табл. 1.2:

Таблица 1.2

Разрешенные виды атомарных операций в OpenMP

Тип атомарной операции	**Допустимые виды выражений**
read	v = x;
write	x = <выражение>;
update (не указан)	x++; x--; ++x; --x; x <бин. оп.> = <выражение>; x = x <бин. оп.> <выражение>;

capture	v = x++;
	v = x--;
	v = ++x;
	v = --x;
	v = x <бин. оп.> = <выражение>;

Так же, как и в предыдущем примере, можно использовать альтернативное решение с использование редукции по переменной n_prime_numbers.

```
#pragma omp parallel for \
reduction(+:n_prime_numbers)
for (long i = min_number; i < max_number; ++i)
{
    ...
    if (is_prime)
        ++n_prime_numbers;
}
```

Измерение временных интервалов

Для оценки времени выполнения программы (или ее части) в OpenMP предусмотрена функция **omp_get_wtime**. Поскольку точность измерения временных интервалов зависит от конкретной реализации, операционной системы и других факторов, то существует дополнительная функция **omp_get_wtick** для опре-

деления точности (разрешающей способности) таймера. Ниже приведен пример типового использования этих функций для измерения времени выполнения участка кода.

```
// время начала расчетов
doublet1 = omp_get_wtime();

// участок кода для оценки времени выполнения
...

// время окончания расчетов
doublet2 = omp_get_wtime();

// вывод затраченного времени
printf("Execution time   : %.5f s\n", t2-t1);
printf("Timer resolution : %.5f s\n",
omp_get_wtick());
```

Сборка OpenMP программ в GNU GCC

Для построения программ с использованием OpenMP в GNU GCC необходимо наличие библиотеки **libgomp** (OpenMP runtime for GCC), которая включена в стандартную поставку.

При сборке программы необходимо указать ключ **-fopenmp**, например:

```
g++ -fopenmp TestOpenMP.cpp -o TestOpenMP.exe
```

Сборка OpenMP программ в MSVS

Для сборки в среде Microsoft Visual Studio необходимо включить поддержку OpenMP на уровне языка в свойствах проекта (рис. 1.3). Это аналогично ключу компилятора **/openmp** при сборке из командной строки.

Следует отметить, что на момент публикации MSVS поддерживает только вторую версию стандарта OpenMP и что данная поддержка отсутвует в Express версиях продукта.

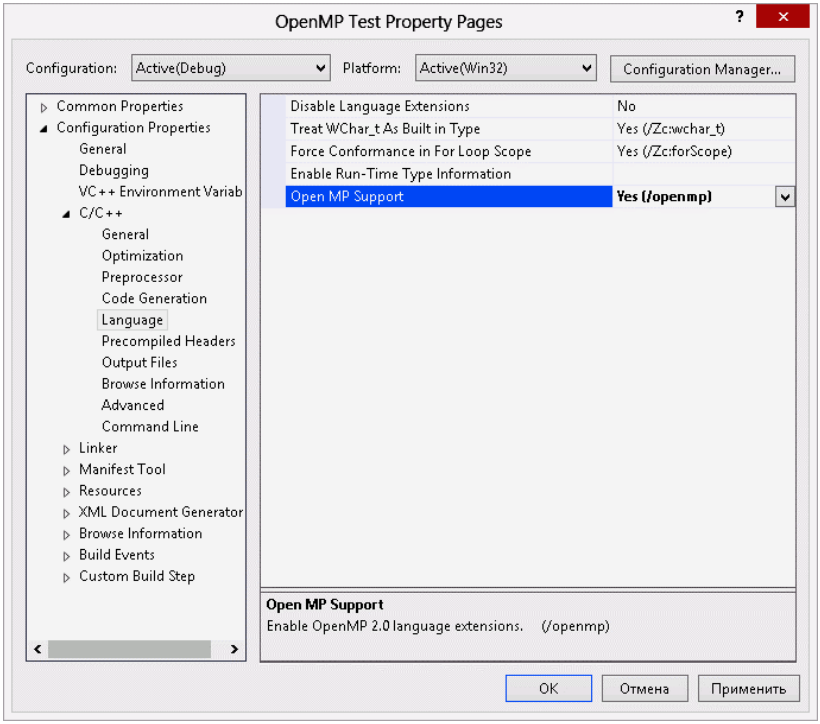


Рис. 1.3. Включение поддержки OpenMP в MSVS

Закон Амдала

Закон был сформулирован в 1967 г. Джином Амдалом (Gene Amdahl). Он позволяет оценить выигрыш от параллелизма и является фундаментальным законом параллельного программирования:

$$S(N) = \frac{1}{(1-P) + P/N},$$

где P – часть программы, которая может быть выполнена параллельно; N – число процессоров; S(N) – ускорение работы программы.

В данном законе сформулировано ограничение на выигрыш от параллелизма в зависимости от последовательной части программы. Так, например, если у нас одна пятая часть задачи выполняется последовательно, то при четырех процессорах мы

получим ускорение в 2,5 раза, а при 8 – в 3,33 раза. Но при любом количестве доступных процессоров выигрыш не превысит 5 раз.

Также в данном законе не учитываются накладные расходы на создание, запуск, коммуникацию и завершение работы параллельных задач.

Задание для лабораторной работы

Используя программу перемножения матриц (пример 1.2), провести с ней следующие эксперименты:

1. Запустить последовательную и параллельную версии программы несколько раз (не менее 7) для различных размеров матриц. При минимальном размере время выполнения должно быть в диапазоне от 0.1 с до 1 с, а при максимальном – от 1 мин до 2 мин.

2. Построить совмещенный график зависимости времени выполнения последовательной и параллельной версий программы от размера исходных матриц.

3. Выбрать размер матрицы, при котором параллельная версия программы выполняется около 30 с. Все дальнейшие эксперименты проводить с матрицами данного размера.

4. Определить, как изменяется время работы программы при распараллеливании внутренних циклов. Объяснить полученные результаты.

5. Для последовательной версии программы замерить время выполнения секции, которая будет выполняться параллельно в параллельной версии, и время выполнения всей программы целиком. Используя полученные значения, определить часть программы, которая будет выполняться параллельно, используя следующее соотношение:

$$P = \frac{T_{\text{секции}}}{T_{\text{программы}}}.$$

Рассчитать теоретическое ускорение работы программы по закону Амдала.

6. Взять лучший результат работы программы из пункта 4 (с минимальным временем выполнения) и рассчитать реальное ускорение как отношение времени выполнения последователь-

ной и параллельной версий программы. Сравнить с теоретическим значением и сделать выводы.

7. При наличии возможности проверить, как изменяется время работы программы при использовании двух, четырех и шести (восьми) потоков. Для этого можно вручную ограничить количество потоков, используя переменную окружения.

Используя программу подсчета количества простых чисел в указанном диапазоне (пример 1.3), провести с ней следующие эксперименты:

8. Запустить последовательную и параллельную версии программы несколько раз (не менее 5) для различных диапазонов. Рекомендуется нижнюю границу зафиксировать на числе 2 и изменять лишь верхнюю границу. При минимальном размере время выполнения должно быть в диапазоне от 0.1 с до 1 с, а при максимальном – от 1 мин до 2 мин.

9. Построить совмещенный график зависимости времени выполнения последовательной и параллельной версии программы в зависимости от размера исходной задачи (верхней границы диапазона).

10. Выбрать диапазон, при котором время выполнения последовательной версии программы будет около 30 с. Все дальнейшие эксперименты проводить с этим диапазоном.

11. Выполнить ряд тестов с различными алгоритмами диспетчеризации (static, dynamic, guided) и различными значениями CHUNK_SIZE (1, 2, 4, 10, 50, 100, 1000). Объяснить полученные результаты. Сделать вывод об оптимальных стратегиях диспетчеризации для данной задачи.

12. Построить совмещенный график зависимости времени выполнения для различных дисциплин диспетчеризации от размера CHUNK_SIZE. Отметить на графике время выполнения последовательной версии программы при помощи горизонтальной линии.

13. Аналогично пунктам 5 и 6 рассчитать априорное и апостериорное значения ускорения работы программы, сравнить их и объяснить результаты.

14. При наличии возможности проверить, как изменяется время работы программы при использовании одного, двух, четырех и шести (восьми) потоков. Для этого можно вручную

ограничить количество потоков, используя переменную окружения.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Характеристики используемого аппаратного и программного обеспечения.
4. Результаты экспериментов для программы перемножения матрицы:

4.1. Таблица и совмещенный график зависимости времени выполнения последовательной и параллельной версии программы в зависимости от размера исходных матриц.

4.2. Фрагменты кода и таблица с временем выполнения программы при распараллеливании каждого из трех циклов программы.

4.3. Расчет ускорения работы программы (теоретическое и реальное значения).

4.4. Таблица и график зависимости времени выполнения программы от количества потоков (при наличии такой возможности).

5. Результаты экспериментов для программы поиска простых чисел в указанном диапазоне:

5.1. Таблица и совмещенный график зависимости времени выполнения последовательной и параллельной версии программы в зависимости от размера исходной задачи (верхней границы диапазона).

5.2. Таблица и совмещенный график зависимости времени выполнения для каждой из трех дисциплин диспетчеризации в зависимости от размера CHUNK_SIZE.

5.3. Расчет ускорения работы программы (теоретическое и реальное значения).

5.4. Таблица и график зависимости времени выполнения программы от количества используемых потоков (при наличии такой возможности).

6. Выводы по работе.

Вопросы для контроля знаний

1. Чем обусловлены накладные расходы в параллельных программах?
2. Опишите модель выполнения “Fork&Join”.
3. Как при помощи средств OpenMP можно узнать количество запущенных в данный момент потоков и номер конкретного потока?
4. Что такое ситуация “гонки”? Когда она возникает?
5. Перечислите все механизмы синхронизации потоков в OpenMP.
6. Что такое редукция? Какие операции редукции допустимы в OpenMP?
7. Чем ждущая блокировка отличается от обычной?
8. Перечислите преимущества атомарных операций.
9. О чем говорит закон Амдала? Учитываются ли накладные расходы в данном законе?
10. Какие функции в OpenMP используются для измерения временных интервалов?
11. Какие виды диспетчеризации потоков реализованы в OpenMP? Каковы их отличительные особенности?

Лабораторная работа 2
АЛГОРИТМЫ ПАРАЛЛЕЛЬНОЙ СОРТИРОВКИ

Цели работы

Знакомство с алгоритмами параллельной сортировки. Получение практического опыта по реализации алгоритма параллельной сортировки с использованием библиотеки OpenMP.

Изучение проблемы балансировки нагрузки. Определение коэффициента использования (загруженности) системы.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с многоядерным процессором (желательно наличие не менее четырех ядер);

– установленная ОС Windows или Linux (для данной работы рекомендуется использовать 64-битную версию ОС, это позволит избежать нехватки памяти при обработке больших объемов данных);

– компилятор для языка C/C++ с поддержкой OpenMP.

В качестве компилятора могут использоваться GNU GCC версии 4.7 и выше (под ОС Windows рекомендуется использовать пакет MinGW) либо MS Visual Studio 2008 и более поздние версии (не все версии MSVS имеют поддержку OpenMP, более подробную информацию можно получить на сайте производителя).

Вводная часть

Сортировка является одной из наиболее часто выполняемых операций с набором данных. Практически ни одна программа не обходится без сортировки. Большинство алгоритмов используют сортировку как один из этапов своей работы. Учитывая распространенность задачи сортировки, желание ускорить ее решение путем использования параллельных вычислений вполне очевидно.

Некоторые алгоритмы последовательной сортировки могут быть адаптированы к параллельному вычислению. Сегодня разработано множество алгоритмов параллельной сортировки, часть из них требует специального оборудования для своей работы. Мы рассмотрим лишь наиболее часто используемые решения.

В общем виде задача сортировки последовательности (a_1, a_2, a_3, …, a_N) сводится к нахождению перестановки (a_i, a_j, …, a_k) такой, что $a_i \le a_j \le \ldots \le a_k$. Часть информации, используемая для сравнения элементов последовательности, называется ключом, остальная – сопутствующей информацией. Часто вместо сортировки самой последовательности, особенно если элементы последовательности довольно большие, сортируют последовательность индексов (1, 2, …, N), и такая операция называется построением индекса. Это позволяет избежать ненужного копирования элементов во время выполнения перестановок элементов. Построение индексов также очень удобно в случае, если

нам надо одновременно работать с последовательностью, отсортированной разными способами – например, если требуется отсортировать страны отдельно по площади территории и по численности населения.

Рассматриваемые в этой работе алгоритмы сортировки будут предполагать, что исходная последовательность целиком располагается в оперативной памяти – такие алгоритмы называются внутренними сортировками (internal sort). В противоположность им существуют алгоритмы внешней сортировки (external sort), которые могут сортировать большие последовательности данных, не помещающихся в оперативной памяти.

Блочная сортировка

Данную сортировку называют также корзинной сортировкой (bucket sort, bin sort). Идея сортировки состоит в распределении элементов исходной последовательности по конечному числу “корзин” таким образом, чтобы элементы в следующей “корзине” были больше (или меньше), чем в предыдущей. Далее элементы в каждой корзине сортируются этим же, либо другим алгоритмом и собираются обратно в единую последовательность.

Данный алгоритм требует знаний о природе сортируемых данных, чтобы обеспечить равномерное распределение по корзинам. Другим недостатком являются затраты оперативной памяти под “корзины”.

В качестве операций, выполняемых параллельно, помимо сортировки отдельных “корзин” можно использовать операции распределения элементов по корзинам и сбор элементов в единую последовательность.

Для определения позиции элемента при сборе в единую последовательность достаточно вычислить префиксную (кумулятивную) сумму по числу элементов в каждой корзине. Эта префиксная сумма и будет порядковым номером элемента для первого элемента из каждой корзины (рис. 2.1).

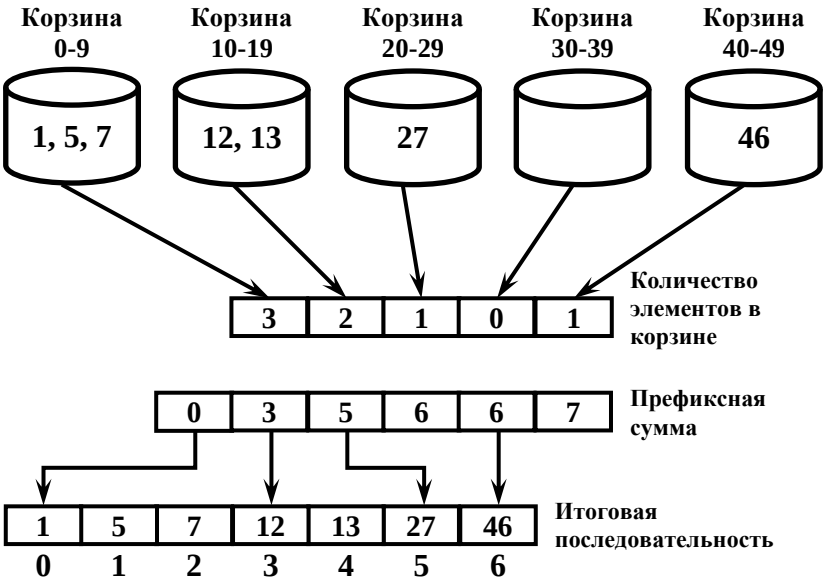


Рис. 2.1. Определение позиций элементов корзины в итоговой последовательности

В данном примере мы распределили элементы исходной последовательности между пятью корзинами, при этом одна из корзин оказалась пустой. Отсортировав данные для каждой корзины, мы начинаем собирать их в единую последовательность. Количество элементов в каждой корзине нам известно. Используя эту информацию, вычисляем префиксную сумму, которая в данном случае означает количество элементов во всех предшествующих корзинах или позицию в итоговой последовательности для первого элемента корзины. После того, как позиции элементов определены, их копирование может выполняться параллельно для каждой из корзин.

В зависимости от входных данных и используемого алгоритма для сортировки данных внутри “корзины” последовательный алгоритм имеет сложность от $O(k+N)$ до $O(N^2)$.

Время работы параллельного алгоритма складывается из следующих слагаемых:

– времени распределения элементов по “корзинам”,

– времени сортировки “корзин”,

– времени сбора элементов в итоговую последовательность (вычисление префиксной суммы и копирования элементов).

Если в параллельном алгоритме данные между “корзинами” распределены приблизительно равномерно, количество “корзин” совпадает с числом процессоров P и для сортировки “корзины” используется алгоритм со сложностью $O(N \cdot \log_2 N)$, то итоговая сложность будет выглядеть следующим образом:

$$O(N + \frac{N}{P}\log_2 \frac{N}{P} + P + \frac{N}{P}).$$

Для устранения неравномерного распределения элементов можно выбрать число корзин большим, чем число доступных процессоров в несколько раз и динамически распределять нагрузку между ними. Несмотря на то, что данный подход не решает проблему полностью, в большинстве случаев он обеспечивает более оптимальное распределение нагрузки между процессорами.

Быстрая сортировка

Одним из наиболее часто используемых последовательных алгоритмов сортировки является быстрая сортировка (quicksort). Ее изобрел в 1960 г. английский информатик Энтони Ричард Хоар (Antony Richard Hoare) во время прохождения стажировки в СССР в Московском государственном университете.

Идея алгоритма очень проста, сам алгоритм состоит из нескольких шагов:

1. В исходной последовательности выбирается некоторый элемент a_M. В качестве такого элемента может выступать последний (или первый) элемент последовательности, можно использовать средний элемент или выбирать его случайным образом.

2. Исходная последовательность перестраивается таким образом, что все элементы меньше a_M располагаются слева от него, а все элементы больше a_M – справа.

3. Алгоритм вызывается рекурсивно для левой и правой части соответственно. Для последовательностей размером 1 элемент никаких операций не производится, для последовательно-

стей размером 2, при необходимости выполняется перестановка элементов.

Алгоритм имеет среднюю сложность работы $O(N \cdot \log_2 N)$, а в худшем случае сложность составит $O(N^2)$.

Стратегия, используемая в этом алгоритме, называется “разделяй и властвуй” и позволяет легко преобразовать его в параллельную версию. Помимо параллельного выполнения сортировки для левой и правой частей последовательности, полученной в результате разбиения исходной последовательности, перестроение последовательности также можно выполнять параллельно. Алгоритм параллельной перестройки последовательности подробно описан в книге “Introduction to parallel computing” [11].

Схема работы этого алгоритма представлена на рис 2.2.

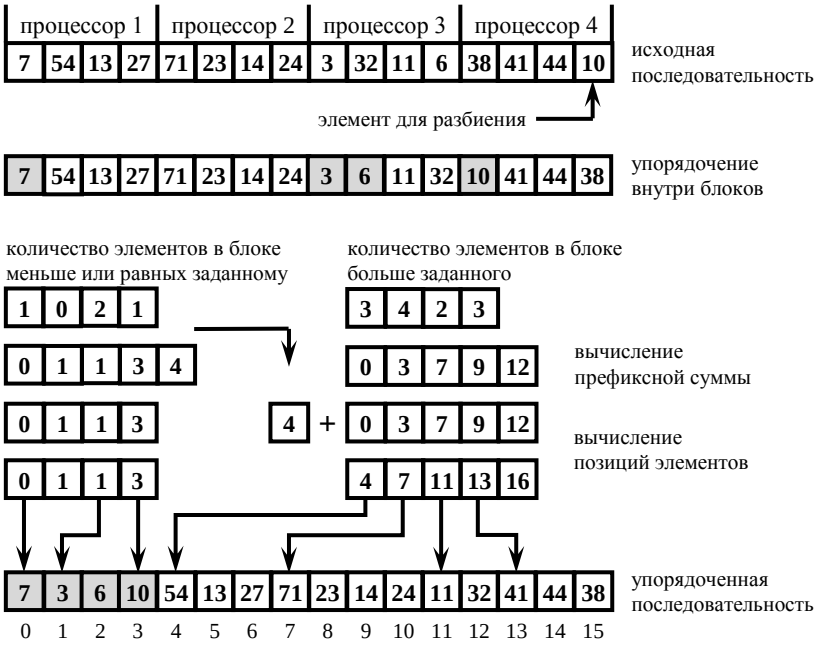


Рис. 2.2. Параллельное перестроение последовательности в алгоритме быстрой сортировки

Первоначально исходная последовательность делится на примерно равные части между всеми процессорами. После выбора элемента для разбиения (мы используем в качестве его последний элемент - “10”), каждый из процессоров перестраивает свою часть последовательности. При этом элементы, меньшие или равные выбранному, располагаются в левой части, а остальные – в правой. После этого каждый из процессоров вычисляет, сколько элементов оказалось в левой и правой частях соответственно. Для полученных последовательностей вычисляется префиксная сумма.

Последнее значение префиксной суммы для левой части нужно прибавить к каждому значению из правой части. Таким образом, мы получим позиции элементов, с которых начинается размещение элементов левой и правой частей последовательности, например, для процессора 3 элементы левой части будут располагаться в упорядоченной последовательности начиная с позиции 1, а правой – с позиции 11.

Недостатком такого алгоритма являются дополнительные затраты памяти, так, при формировании упорядоченной последовательности нам все еще необходима копия элементов исходной последовательности. Последовательная операция перестраивания последовательности может работать без дополнительных затрат памяти и требует O(N) операций.

После разбиения работу следует перераспределить между процессорами и продолжить работу. Для данного примера логично выделить процессор 1 для сортировки левой части, а остальные три процессора – для правой.

Так, если в нашем распоряжении имеется P процессоров, то при самом простом подходе достаточно разделить исходную последовательность на P приблизительно равных частей и далее сортировать их параллельно друг с другом при помощи последовательного алгоритма. При этом мы получим среднюю сложность параллельного алгоритма $O(\frac{N}{P}\log_2\left(\frac{N}{P}\right))$.

Точно так же, как и в предыдущем случае, для равномерного распределения нагрузки между всеми процессорами можно разбить исходную последовательность на гораздо большее (в несколько раз) число блоков и далее динамически распределять

нагрузку между процессорами. Учитывая, что при каждом разбиении из одного блока получается два, то для разбиения на каждом шаге следует выбирать блок максимальной длины. Чтобы не увеличивать накладные расходы, блоки меньше определенного размера следует сортировать последовательным алгоритмом.

Сортировка Шелла

Еще одним видом сортировки, которая легко адаптируется к параллельному выполнению, является сортировка Шелла (shellsort), изобретенная Дональдом Шеллом (Donald Shell) в 1959 г.

Сортировка Шелла фактически является сортировкой вставками с несколькими проходами. На каждом проходе сортируются элементы отстоящие друг от друга на расстоянии d. На каждом следующем проходе значение d уменьшается, пока в результате оно не станет равным 1. Последний проход фактически является классической сортировкой вставками.

Сложность алгоритма зависит от выбора последовательности d_1, d_2, …, d_k. Эффективность, по сравнению с классическим алгоритмом сортировки вставками, достигается за счет того, что элементы “быстрее” встают на свои места и требуется меньшее количество перестановок. Вполне очевидно, что для первой итерации должно выполняться условие $d_1 \leq N/2$.

Каждый из проходов алгоритма, кроме последнего, может обрабатывать данные параллельно, при этом на каждом проходе может быть задействовано до d потоков одновременно. Например, при d = 3 одновременно может работать до 3 потоков, при этом первый поток будет сортировать элементы a_1, a_4, a_7 и т.д.; второй поток – элементы a_2, a_5, a_8 и т.д.; третий – элементы a_3, a_6, a_9 и т.д. соответственно.

Таким образом, выбор расстояния d также повлияет на то, сколько потоков смогут работать одновременно. Существует множество вариантов для выбора последовательности {d}, часть из них приведена в табл. 2.1:

Таблица 2.1

Основные последовательности чисел для выбора расстояния в сортировке Шелла

Последовательность	**Элементы последовательности**	**Сложность последовательного алгоритма (худший случай)**	**Автор**
$\lfloor N/2^k \rfloor$	$\lfloor N/2 \rfloor$, $\lfloor N/4 \rfloor$, …, 1	$O(N^2)$	Шелл
2^k-1	1, 3, 7, 15, 31, 63, …	$O(N^{3/2})$	Хаббард
$(3^k-1)/2$, не более $\lceil N/3 \rceil$	1, 4, 13, 40, 121, …	$O(N^{3/2})$	Кнут
$4^k+3\cdot 2^{k-1}+1$, дополненная 1	1, 8, 23, 77, 281, …	$O(N^{4/3})$	Седжвик
числа Фибоначчи	1, 2, 3, 5, 8, 13, …	—	—

Сортировка слиянием

Фактически под этим термином скрывается целый класс алгоритмов сортировки, построенных на общем принципе. Первоначальный алгоритм предложен Джоном Фон Нейманом (John von Neumann) в 1945 г. Алгоритм также использует принцип “разделяй и властвуй” и состоит из трех этапов:

1. Исходная последовательность делится на две части, примерно одинакового размера.

2. Каждая из последовательностей сортируется отдельно, например этим же алгоритмом. Последовательность длиной 1 считается отсортированной.

3. Две отсортированные последовательности соединяются в одну.

Сложность алгоритма для худшего случая составит $O(N \cdot \log_2 N)$, такая же сложность получается и в среднем. Основным недостатком алгоритма является необходимость дополнительной памяти при слиянии двух упорядоченных последовательностей, в общем случае необходима дополнительная память под N элементов.

Алгоритм может работать в двух вариантах:

– “сверху-вниз” (нисходящая), когда исходная последовательность делится на две равные части, каждая из которых затем также делится пополам и т.д.;

– “снизу-вверх” (восходящая), когда соседние элементы последовательности (нечетный с четным) объединяются в отсортированные пары, которые затем объединяются в “четверки” и т.д., пока не получится итоговая последовательность.

Параллельная версия данного алгоритма получается путем параллельного выполнения сортировок последовательностей, полученных при разбиении. Операция слияния двух последовательностей выполняется последовательно на одном процессоре. Если в нашем распоряжении имеется P процессоров, то исходную последовательность можно сразу поделить на P равных частей, отсортировать их последовательным алгоритмом на каждом из процессоров и затем объединять друг с другом, пока не получится итоговая последовательность.

Особенностью данного алгоритма является возможность сортировки больших объемов данных, расположенных на внешних носителях с последовательным доступом (например, расположенных в виде файлов на жестком диске). На первом этапе данные порциями сортируются в оперативной памяти, промежуточные результаты сохраняются в файлы, которые затем объединяются в более крупные файлы, пока не будет получена итоговая последовательность. Эта особенность позволяет легко адаптировать алгоритм к выполнению на вычислительном кластере.

Сортирующие сети (сети сортировки)

Сети сортировки – это абстрактная математическая модель. Любой алгоритм сортировки, в котором последовательность операций сравнения не зависит от предыдущих результатов сравнения, может быть представлен в виде сети сортировки. Например, сортировка Шелла, сортировка методом пузырька и сортировка вставками могут быть представлены в виде сети сортировки, а быстрая сортировка – нет.

Сеть сортировки обычно представляется в виде горизонтальных линий, соответствующих передаче сортируемого элемента слева направо, и вертикальных соединениях пар линий, называемых компараторами и соответствующих операции “сравнить и переставить” элементы местами. Пример такой сети для сортировки пузырьком последовательности из 5 элементов приведен на рис. 2.3. Более детальное описание сетей сортировки можно найти в книге Роберта Седжвика “Алгоритмы на C++” [4].

Сортирующую сеть легко адаптировать к параллельному выполнению. Сравнения, в которых задействованы несвязанные пары чисел, можно проводить параллельно. В рассмотренном алгоритме пузырьковой сортировки таких операций немного, но они есть: например, компаратор на линии 5-6 и следующий за ним компаратор на линии 1-2. Как мы увидим далее, существуют более оптимальные сети сортировки для параллельного выполнения.

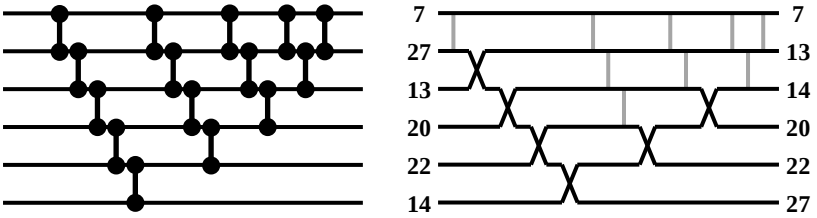


Рис. 2.3. Сеть сортировки для алгоритма пузырьковой сортировки:
а – общая структура сети с компараторами; *б* – пример работы сети для последовательности из шести чисел

Строить поэлементную сеть для большого количества элементов довольно сложно. Поэтому в случае применения сети для параллельной сортировки используют приводимый ниже подход.

Вся исходная последовательность делится на равные части между всеми доступными процессорами. Очень важно, чтобы размеры всех частей были одинаковыми, поэтому при необходимости последовательность можно дополнить элементами-пустышками, которые гарантированно окажутся в конце результирующей последовательности и могут быть легко удалены после окончания сортировки. Каждый процессор сортирует свою часть данных, после чего для объединения их в общую итоговую последовательность используется сортирующая сеть.

Равный размер частей необходим для правильной работы компараторов слияния. Компаратор слияния принимает на входе две упорядоченные последовательности элементов длиной M и выдает на выходе две упорядоченные последовательности той же длины, в первой из них содержатся M наименьших элементов из входных последовательностей, а во второй – M наибольших элементов. Правомерность замены поэлементных компараторов на компараторы слияния рассмотрена в книге Роберта Седжвика [4].

Пример работы компараторов слияния для последовательностей, состоящих из символов, приведен на рис. 2.4. В данном примере достаточно трех параллельных шагов для получения результата.

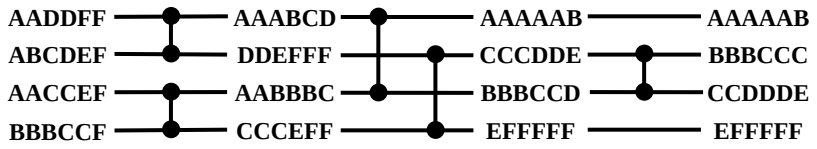


Рис. 2.4. Пример адаптации сетей сортировки для последовательностей элементов при помощи компараторов слияния

Как правило, каждая горизонтальная линия при параллельной обработке соответствует отдельному процессору, а каждый компаратор фактически отображает обмен данными между па-

рой процессоров. Операция слияния последовательностей между парой процессоров обычно выполняется по следующей схеме:

– процессоры обмениваются последовательностями друг с другом, таким образом, у каждого процессора получается две последовательности длины M;

– каждый из процессоров выполняет операцию слияния и получает единую последовательность длиной 2M;

– один из процессоров заменяет исходную последовательность первыми M элементами, а второй процессор – последними M элементами.

При использовании сети сортировки на модели с разделяемой памятью, слияние может выполнять лишь один процессор из двух. Схема работы с участием обоих процессоров позволяет легко адаптировать сети сортировки к выполнению на вычислительном кластере. Далее мы рассмотрим три сети для параллельной сортировки.

Параллельная сортировка чет-нечет

Данная сортировка, представляющая собой довольно простой алгоритм, была разработана для параллельных систем, где каждый элемент был соединен с двумя соседями – левым и правым. Идея алгоритма состоит в следующем: на каждом шаге алгоритма выполняются операции слияния-разбиения с соседним процессором. На нечетном шаге четные процессоры взаимодействуют с нечетным соседом справа, а на четном шаге – с нечетным соседом слева. Максимальное количество шагов, необходимое для завершения сортировки, равно P, где P – число процессоров. Сортировку можно завершить и раньше, если за последние две итерации порядок элементов не изменился. Сеть параллельной сортировки чет-нечет для 8 процессоров показана на рис. 2.5.

Количество компараторов, задействованных в данной сети, будет равно $(N-1) \cdot N/2$. Для сетей размером восемь их число составит 28, а для сетей размером шестнадцать – 120.

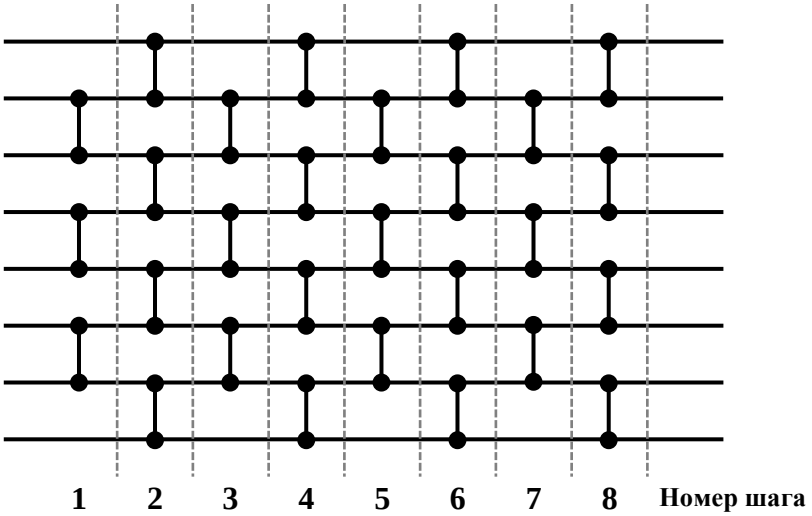


Рис. 2.5. Структура сети для параллельной сортировки чет-нечет с использованием восьми процессоров

Нечетно-четная сортировка слиянием Бэтчера

Недостатком предыдущего алгоритма является большое число параллельных шагов. Данная сеть сортировки (Batcher`s odd-even mergesort) позволяет уменьшить число шагов параллельного алгоритма до $(\log_2 P)^2/2$, где P – количество процессоров. Алгоритм был разработан Бэтчером в 1968 г.

Как видно на рис. 2.6, данная сеть имеет рекурсивную структуру: сеть на 16 линий содержит две сети на 8 линий, четыре сети на 4 линии и 8 сетей на 2 линии.

Также на рисунке мы видим, что для данной сети достаточно 10 шагов для 16 процессоров и 6 шагов для 8 процессоров, что лучше, чем в предыдущем алгоритме. Еще одним достоинством данной сети является меньшее число компараторов, а значит требуется меньше взаимодействия между отдельными процессорами.

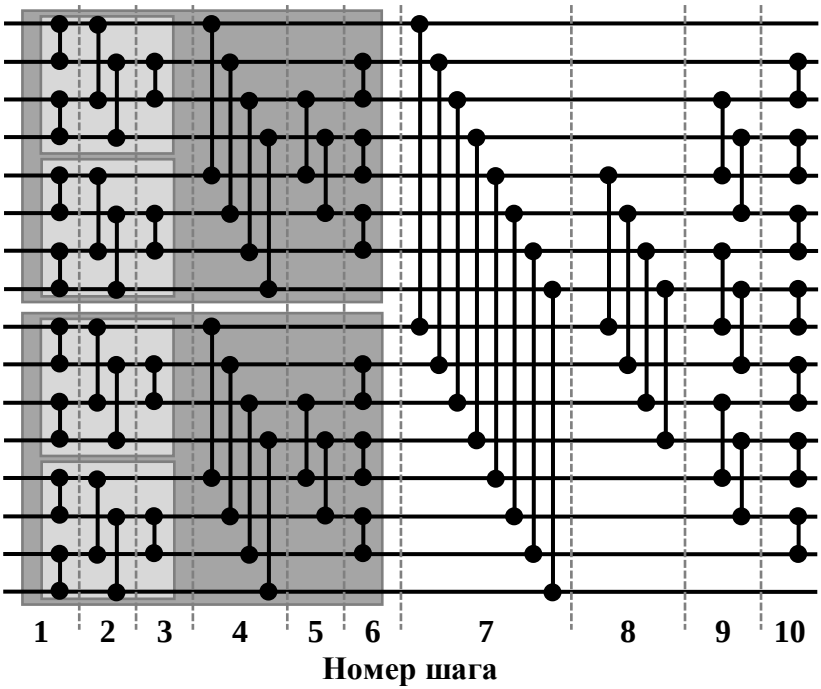


Рис. 2.6. Нечетно-четно сортирующая сеть Бэтчера для шестнадцати процессоров

Для сети из P процессоров потребуется порядка $P\cdot(\log_2 P)^2/4$ компараторов. Точное значение для сети из 16 процессоров будет равно 63 компараторам, а для сети из 8 процессоров – 19.

Несмотря на то, что было доказано существование сортирующих сетей с $O(N \cdot \log_2 N)$ компараторами (AKS сети), сети Бэтчера остаются одним из наиболее практических методов параллельной сортировки. Данные сети популярны и при гетерогенных вычислениях (с использованием видеоадаптеров), подробное описание которых можно найти в книге “GPU Gems 2” [15, глава 46 – Improved GPU Sorting].

Битоническая сортировка

Битоническая сортировка также была разработана Бэтчером в 1968 г., она имеет приблизительно такое же количество ком-

параторов и шагов алгоритма. Работа алгоритма основана на понятии битонической последовательности.

Последовательность $\{a_1, a_2, \ldots, a_{k-1}, a_k, a_{k+1}, \ldots, a_n\}$ называется битонической, если элементы в ней сначала монотонно не убывают, а потом монотонно не возрастают, т.е.

$\forall_i\ 0 \le i < k,\ a_i \le a_{i+1}$,

$\forall_j\ k \le j < n,\ a_j \ge a_{j+1}$.

Последовательность, полученная из битонической последовательности путем циклического сдвига, также считается битонической.

Очевидно, что любая отсортированная последовательность является битонической; обратное утверждение неверно. Также очевидно, что реверсирование битонической последовательности также будет являться битонической последовательностью.

В алгоритме используется свойство битонических последовательностей, которое позволяет разделить ее на две битонические последовательности равной длины, причем элементы первой последовательности окажутся меньше элементов второй последовательности.

Пусть у нас имеется битоническая последовательность $\{a_0, a_1, \ldots, a_{2n-1}\}$, тогда

$S_1 = \{\min(a_0, a_n), \min(a_1, a_{n+1}), \ldots, \min(a_{n-1}, a_{2n-1})\}$,

$S_2 = \{\max(a_0, a_n), \max(a_1, a_{n+1}), \ldots, \max(a_{n-1}, a_{2n-1})\}$.

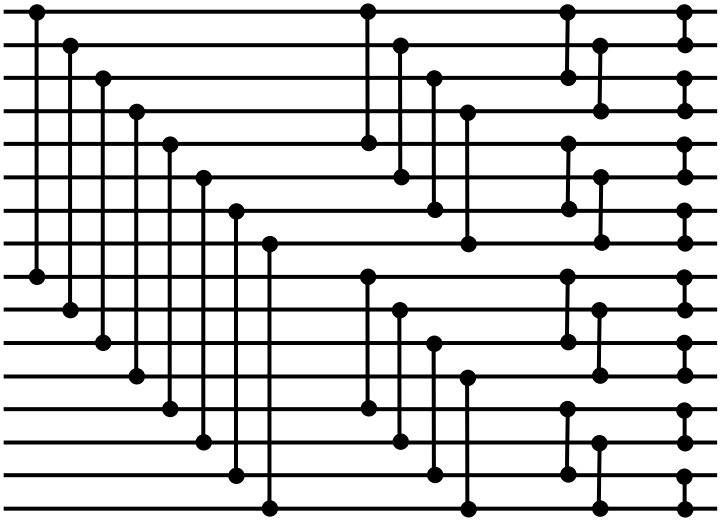
Обе полученные последовательности также являются битоническими и при этом выполняется следующее условие:

$\forall_x \forall_y\ x \in S_1,\ y \in S_2\ x \le y$.

Таким образом, используя простые компараторы, можно легко построить сеть для преобразования битонической последовательности в отсортированную (рис. 2.7, *а*).

Поскольку в общем случае исходная последовательность не является битонической, ее необходимо предварительно преобразовать в битоническую. Любая последовательность из двух элементов является битонической, две такие последовательности можно объединить в последовательности из “четверок” и т.д. при помощи сети, изображенной на рис. 2.7, *б*.

а



б

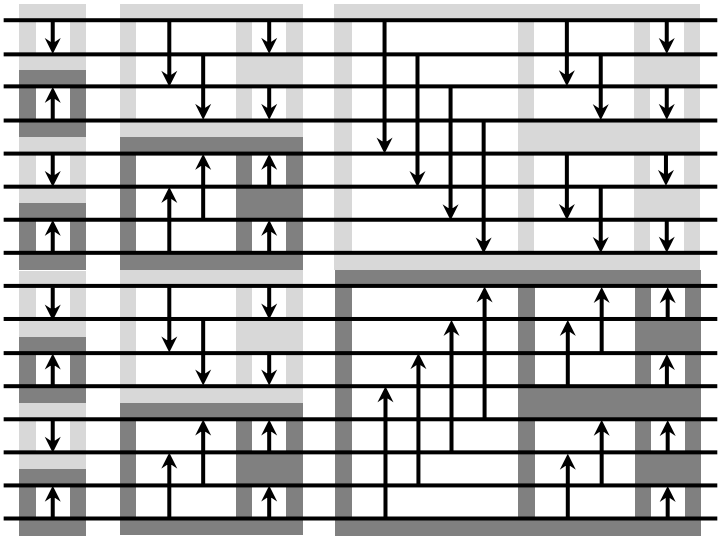


Рис. 2.7. Битоническая сеть для шестнадцати элементов: *а* – сеть слияния; *б* – сеть для преобразования исходной последовательности элементов в битоническую (с разнонаправленными компараторами)

Компараторы в данной сети разнонаправленные, стрелка указывает на позицию, куда будет помещен больший элемент. На рисунке хорошо виден принцип построения сети из отдельных блоков.

Все блоки белого цвета имеют одинаковую структуру и попарно сравнивают первую половину элементов со второй. Если на вход такого блока подать две битонические последовательности (на верхнюю половину входов подается одна последовательность, на нижнюю – другая), то на выходе мы также получим две битонические последовательности, причем все элементы первой последовательности будут меньше второй.

Последовательность из блоков белого цвета формирует области светло-серого и темно-серого цветов. За исключением порядка компараторов эти области эквивалентны. Если на вход светло-серого блока подать битоническую последовательность, то на выходе мы получим последовательность, отсортированную по возрастанию, а для темно-серого – по убыванию.

Объединив обе сети, мы получим сеть битонической сортировки. Путем некоторых преобразований можно добиться использования во всей сети однонаправленных компараторов. Итоговая сеть сортировки (модифицированная) представлена на рис. 2.8.

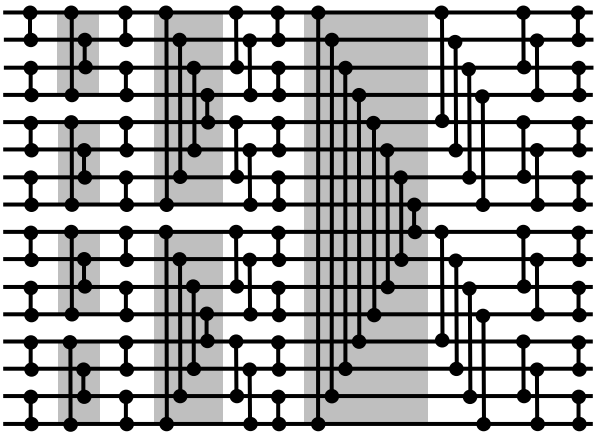


Рис. 2.8. Сеть битонической сортировки для шестнадцати элементов

Серым цветом на рисунке показаны модифицированные элементы для использования однонаправленных компараторов. Следует отметить, что для правильной работы алгоритма количество процессоров, задействованных в сети сортировки, должно быть степенью числа 2.

Сортировка перечислением

В заключение мы рассмотрим еще один алгоритм сортировки, который легко адаптируется к параллельному вычислению.

Идея этого алгоритма сортировки состоит в ранжировании элементов. В качестве ранга используется число элементов в исходной последовательности меньше данного. Для этого каждый элемент a_k исходной последовательности $\{a_1, a_2, \ldots, a_n\}$ попарно сравнивается с каждым другим элементом последовательности. Полученное число и будет позицией элемента в итоговой последовательности.

Последовательная версия данного алгоритма имеет сложность $O(N^2)$. К недостаткам данного алгоритма также можно отнести необходимость выделения памяти под итоговую последовательность.

Реализация алгоритма на языке C++ может выглядеть следующим образом:

```
// x - исходная последовательность
// y - итоговая последовательность
for (size_t i = 0; i < N; ++i)
{
   int rank = 0;
   for (size_t j = 0; j < N; ++j)
   {
      if (x[i] > x[j])
         ++rank;
   }
   y[rank] = x[i];
}
```

Как видно из приведенного алгоритма, внешний или внутренний цикл легко распараллеливается. Заметим, что приведенная реализация не учитывает наличие одинаковых элементов в

исходной последовательности, для этого необходимо внести в алгоритм небольшие модификации.

Коэффициент использования системы

В ряде случаев вычислительную нагрузку не удается распределить равномерно между всеми доступными процессорами (ядрами). Такое поведение может быть обусловленно спецификой самого алгоритма, особенностями входного набора данных или другими факторами. В результате при решении задачи часть вычислителей вынуждена простаивать, и реальный выигрыш от параллелизма получается меньше ожидаемого. Данная особенность характерна для большинства алгоритмов сортировки, приведенных в данной главе.

Для наглядности рассмотрим пример выполнения восходящей сортировки слиянием на четырех процессорах.

На первом шаге исходная последовательность делится на четыре равные части, и каждая из них сортируется при помощи последовательного алгоритма. При этом можно считать, что все процессоры равномерно загружены работой.

Однако уже на втором шаге, когда выполняются слияния четырех отсортированных последовательностей в две последовательности большего размера, работают только два процессора из четырех.

И, наконец, на третьем шаге, когда формируется результирующая последовательность, три процессора из четырех будут простаивать.

Для оценки того, насколько эффективно при решении задачи использовались вычислительные ресурсы системы, введено понятие коэффициента использования системы (utilization). Для систем с симметричными мультипроцессорами (а именно к этому классу относятся многоядерные процессоры для ПК) он определяется как отношение времени, в течение которого вычислители (процессоры) были нагружены полезной работой, к общему фонду временных затрат (сумма времени работы и времени простоя каждого из вычислителей). Как видно из определения, значение коэффициента лежит в пределах от 0 до 1. Чем

ближе значение коэффициента к 1, тем более эффективно используются вычислительные ресурсы системы.

В гетерогенных системах при определении коэффициента следует учесть вычислительные мощности каждого из входящих в ее состав вычислителей (например, при помощи весовых коэффициентов).

Часто данный коэффициент умножают на 100 и записывают в виде процентов.

Задание для лабораторной работы

Необходимо реализовать один из рассмотренных алгоритмов сортировки в соответствии с табл. 2.2. Номер варианта выбирается исходя из порядкового номера студента по списку.

В качестве сортируемых данных следует использовать строки (std::string или std::wstring). Допускается наличие одинаковых строк во входных данных.

Программа должна иметь возможность считывать входные данные из текстового файла и сохранять результаты в файл. Имена входных и выходных файлов должны задаваться через параметры командной строки. В качестве входных данных рекомендуется использовать файлы со строками фиксированного размера (например, длиной 20 символов).

В качестве последовательного алгоритма для сортировки части исходной последовательности разрешается использовать библиотечную версию алгоритма сортировки std::sort.

Используя созданную программу, необходимо провести ряд экспериментов. При измерениях времени работы не следует учитывать время чтения входных данных и записи результатов, только время работы самого алгоритма сортировки.

Поскольку время работы некоторых алгоритмов сортировки очень зависит от первоначального порядка элементов, то необходимо проводить эксперименты не менее, чем на трех тестовых наборах данных. При выборе тестовых данных желательно учитывать особенности алгоритма сортировки (постараться воспроизвести наихудший случай на одном из тестовых наборов).

Таблица 2.2

Алгоритм сортировки для индивидуального задания

Номер варианта	**Алгоритм сортировки**	**Дополнительное исследование**
1	Блочная сортировка	Исследовать влияние количества корзин на время работы параллельного алгоритма
2	Быстрая сортировка	Исследовать влияние параллельной/последовательной перестройки последовательности на общее время работы параллельного алгоритма
3	Сортировка Шелла	Исследовать влияние выбора последовательности на время работы параллельного алгоритма
4	Сортировка слиянием	Определить, как изменится время работы параллельного алгоритма при нисходящей/восходящей сортировке
5	Параллельная сортировка чет-нечет	Исследовать время работы параллельного алгоритма для 4, 8 и 16 процессоров
6	Нечетно-четная сортировка слиянием Бэтчера	Исследовать время работы параллельного алгоритма для 4, 8 и 16 процессоров
7	Битоническая сортировка	Исследовать время работы параллельного алгоритма для 4, 8 и 16 процессоров
8	Сортировка перечислением	Исследовать, как изменится время работы алгоритма при распараллеливании внешнего и внутреннего цикла

1. Определить, как размер входных данных влияет на время работы программы.

1.1. Для каждого тестового набора данных следует произвести не менее 5 измерений для различных размеров входной по-

следовательности, для этого следует использовать первые N строк из входного текстового файла (для каждого из тестовых наборов данных). Рекомендуемые значения для N: 1 000 000, 2 000 000, 5 000 000, 10 000 000, 20 000 000.

1.2. По полученным результатам определить среднее время работы алгоритма для каждого размера входных данных.

1.3. По полученным результатам построить совмещенные графики зависимости времени работы программы от размеров входных данных для каждого из трех (или более) наборов входных данных и усредненного значения.

1.4. Сравнить время работы параллельного алгоритма с временем работы библиотечной сортировки std::sort на тех же тестовых данных. Отметить ее линией на графике.

1.5. Сделать выводы о том, как полученные результаты соотносятся с теоретической сложностью алгоритма.

2. При наличии возможности проверить, как изменяется время работы программы при использовании 1, 2, 4 и 6(8) потоков. Для этого можно вручную ограничить количество потоков, используя переменную окружения. При выполнении этого пункта следует выбрать максимальный размер входных данных: 20 000 000. Испытания провести для всех трех (или более) тестовых наборов данных.

3. Модифицировать программу таким образом, чтобы обеспечить вывод информации о распределении работы между процессорами с привязкой по времени в файл журнала. Основываясь на данной информации, определить коэффициент использования системы (utilization). При проведении этого эксперимента достаточно выбрать тестовый набор данных, на котором получается средняя продолжительность работы при максимальном объеме данных в 20 000 000 строк.

4. Произвести дополнительные эксперименты для выявления особенностей алгоритма параллельной сортировки, согласно индивидуальному заданию.

Содержание отчета

1. Титульный лист.
2. Цель работы.

3. Характеристики используемого аппаратного и программного обеспечения.

4. Таблица и совмещенный график зависимости времени выполнения программы от размера входных данных.

5. Таблица и график зависимости времени выполнения программы от количества используемых потоков (при наличии такой возможности).

6. Рассчитанный коэффициент использования системы путем обработки файла журнала.

7. Результаты дополнительных экспериментов.

8. Выводы по работе.

9. Приложение 1. Фрагмент текста программы, реализующий алгоритм параллельной сортировки, со всеми необходимыми для понимания его работы комментариями, объявлениями структур, переменных и т.п.

10. Приложение 2. Фрагмент лог-файла (один лист А4).

Вопросы для контроля знаний

1. Для чего префиксная сумма используется в алгоритме блочной сортировки?

2. Какие этапы алгоритма быстрой сортировки можно выполнять параллельно?

3. Что такое сеть сортировки? Каким образом сеть сортировки используется при создании параллельного алгоритма?

4. Какие алгоритмы сортировки не могут быть представлены в виде сортирующей сети?

5. Какие из рассмотренных алгоритмов сортировки можно легко адаптировать к выполнению на вычислительном кластере?

6. Что является основным недостатком параллельной сортировки чет-нечет?

7. Дайте определение битонической последовательности. Какие свойства битонических последовательностей используются в алгоритме битонической сортировки?

7. Что показывает коэффициент использования системы (utilization)?

Лабораторная работа 3

КОНВЕЙЕРНАЯ ОБРАБОТКА ДАННЫХ

Цели работы

Знакомство с базовыми принципами конвейерной обработки данных. Получение практического опыта по реализации программного конвейера средствами Intel Thread Building Blocks.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с многоядерным процессором (желательно наличие не менее четырех ядер);

– установленная ОС Windows или Linux;

– компилятор для языка C/C++;

– установленная библиотека Intel TBB версии не ниже 4.1;

– установленная библиотека OpenCV версии не ниже 2.4.3;

– установленный видеокодек для обработки исходного видеоролика.

Поскольку библиотека Intel TBB ориентирована прежде всего на процессоры фирмы Intel, рекомендуется использовать персональные компьютеры на базе этих процессоров. В случае отсутствия такой возможности использовать ПК с процессорами других производителей.

Рекомендуется использовать компиляторы с поддержкой стандарта C++11 (C++0x), так как это значительно упрощает работу с библиотекой Intel TBB. Компилятор с частичной поддержкой этого стандарта входит в состав Microsoft Visiual Studio 2010 и его более поздних версий.

Библиотека Intel TBB может быть загружена с официального сайта: https://www.threadingbuildingblocks.org/.

Библиотека OpenCV (Open Computer Vision) может быть загружена с официального сайта: http://opencv.org/.

При выборе видеокодека следует ориентироваться на формат сжатия видео выбранного видеоролика. Рекомендуется использовать видеоролики со сжатием в одном из следующих

форматов: DivX, Xvid, Motion JPEG. Видеокодек для Xvid можно загрузить с официального сайта: http://www.xvid.org/.

Общие принципы построения вычислительного конвейера

Модель конвейера (pipeline, pipes & filters) часто используется на практике и представляет собой последовательную цепочку блоков обработки информации, при которой выход одного блока является входом другого. Любой современный процессор использует конвейер для очереди команд. Графический конвейер присутствует в графических библиотеках OpenGL и Direct3D. Конвейеры часто используют при обработке файлов с текстовой информацией, в частности, компиляция исходного кода программы в исполняемый код может быть рассмотрена как последовательность лексического анализатора, синтаксического анализатора и генератора исполняемого кода. Один из шаблонов проектирования – Pipes & Filters – фактически представляет собой конвейер.

Проще всего использовать данную модель при обработке потоковых данных, где в качестве отдельного элемента может выступать запись (строка таблицы базы данных), последовательность байт (кадр видеоролика) или бит (блочное шифрование).

Основным достоинством конвейерной обработки является легкость адаптации к параллельной обработке. Ситуация взаимной блокировки (deadlock) в такой модели практически исключена.

В самом простом случае каждый блок обработки информации обрабатывается одним и тем же процессором, а количество используемых процессоров равно количеству блоков. В этом случае говорят о статической привязке процессоров к этапам конвейера (static mapping).

Самый первый и самый последний блок, как правило, занимаются операциями ввода/вывода, тогда как в других блоках такого рода операций стараются избегать.

В качестве примера рассмотрим конвейер из четырех последовательных этапов, изображенный на рис. 3.1.

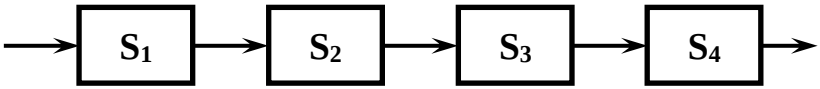


Рис 3.1. Модель конвейера с четырьмя этапами обработки данных

Предположим, что последовательность данных, поступающая на вход этого конвейера, разделена на порции d_1, d_2, …, d_N. Можно синхронизировать обмен данными между всеми процессорами, в этом случае работа всего конвейера будет состоять из циклически повторяющейся последовательности двух шагов:

– обработка очередной порции данных каждым из процессоров;

– передача обработанной порции данных на следующий процессор и получение порции данных от предыдущего процессора.

Именно по такой схеме работают сборочные конвейеры на реальном производстве. Для упрощения представим, что обработка очередной порции данных на каждом этапе занимает T_1, T_2, T_3 и T_4 соответственно и не зависит от самих данных. Следовательно, время обработки очередной порции информации на конвейере будет равно максимальному из этих значений, обозначим его через $T = \max(T_1, T_2, T_3, T_4)$. Также предположим, что время передачи данных между различными стадиями конвейера мало по сравнению с T и им можно пренебречь.

Соответствующая данной модели временная диаграмма обработки данных представлена на рис. 3.2.

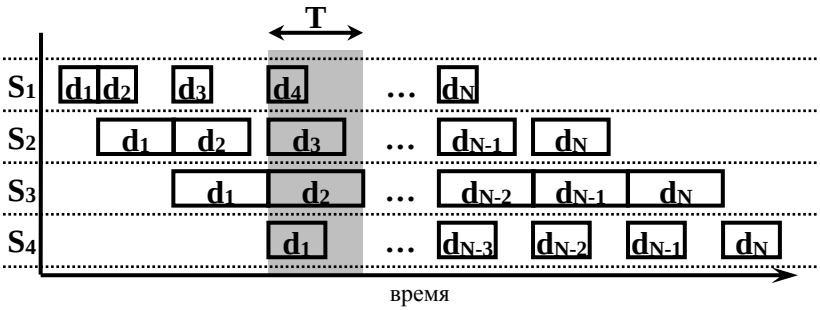


Рис 3.2. Временная диаграмма работы конвейера в модели с синхронным обменом данными между всеми этапами

Общее время вычислений в этом случае составит приблизительно T(N+M), где M – число этапов конвейера.

Если N достаточно велико, то простоями отдельных процессоров при запуске и остановке работы конвейера можно пренебречь.

В этом случае коэффициент использования всей системы (utilization) составит:

$$utilization = \frac{\sum_{i=1,M} T_i}{M \cdot \max\limits_{i=1,M} T_i}.$$

Если время обработки на каждом этапе соотносится между собой как числа 2:4:5:3, то легко подсчитать, что коэффициент использования всей системы составит 0,7 или 70%. Чем больше различается между собой время обработки данных на каждом этапе, тем ниже получится коэффициент (при данной модели организации конвейера). Достоинствами такой модели являются ее простота и отсутствие дополнительных затрат памяти, так как мы храним в памяти только те порции данных, которые обрабатываем в данный момент.

На практике время работы каждого этапа конвейера зависит от входных данных, а сами данные могут поступать на вход конвейера неравномерно. В качестве примера такой ситуации может выступать программа, выполняющая индексирование страниц интернет сайтов, – время загрузки каждой страницы зависит от загруженности сервера и линий связи, размера страницы и т.д.

Поэтому для передачи информации от одного блока к другому часто используют кольцевые буферы фиксированной длины или очереди. Применение кольцевых буферов предпочтительнее, так как в этом случае мы сможем приостановить работу конвейера, если данные начнут поступать быстрее, чем будут успевать обрабатываться (когда буфер заполнится целиком). К выбору размера кольцевого буфера следует относится внимательно – слишком маленький размер негативно скажется на производительности конвейера, а слишком большой размер приведет к неоправданному расходу используемой оперативной памяти.

Рассмотрим, как изменится временная диаграмма работы при использовании кольцевых буферов на 4 элемента каждый (рис. 3.3).

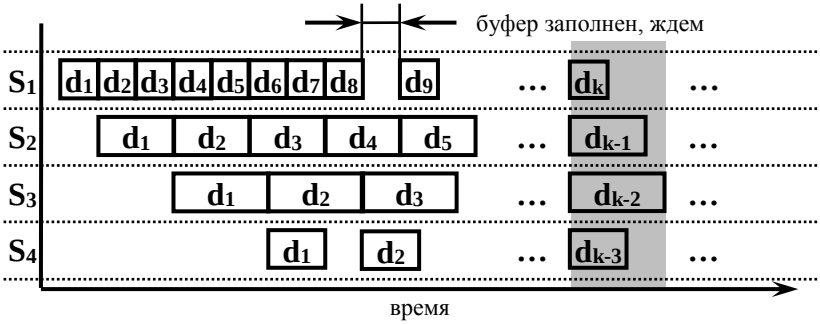


Рис 3.3. Временная диаграмма работы конвейера в модели с кольцевыми буферами на 4 элемента

После обработки порции данных d_8 на этапе S_1 мы вынуждены ждать, так как в буфере уже находятся четыре порции данных: d_5, d_6, d_7, d_8. Продолжить обработку данных мы сможем после того, как порция d_5 будет извлечена из буфера. Если продолжить работу конвейера, то буфер между этапами S_2 и S_3 также полностью заполнится. С другой стороны, в буфере между этапами S_3 и S_4 никогда не будет более одной порции данных, а большую часть времени он будет пустой. Через некоторое время диаграмма работы станет похожей на ту, что была при синхронном обмене данными.

Таким образом, кольцевые буферы улучшают работу конвейера при неравномерном поступлении (или неравномерной обработке) данных, но не позволяют улучшить коэффициент использования системы.

На практике можно столкнуться с ситуацией, когда количество доступных процессоров не совпадает с количеством этапов конвейера. В этом случае процессоры не привязываются к конкретному этапу конвейера, а назначаются планировщиком исходя из текущей ситуации (dynamic mapping) для выполнения операции S_i над порцией данных d_j.

Если алгоритм обработки очередной порции данных d_j на этапе S_i не зависит от результатов обработки предыдущих пор-

ций данных на этом же этапе, то такой этап можно выполнять параллельно над разными порциями данных. Это позволяет еще более эффективно использовать вычислительные ресурсы.

Допустим, что в нашем примере этапы S_2 и S_3 могут выполняться параллельно, а этапы S_1 и S_4 должны сохранять порядок выполнения. На временной диаграмме (рис. 3.4) слева теперь отображаются доступные процессоры, а для обработки данных дополнительно указывается производимая над ними операция.

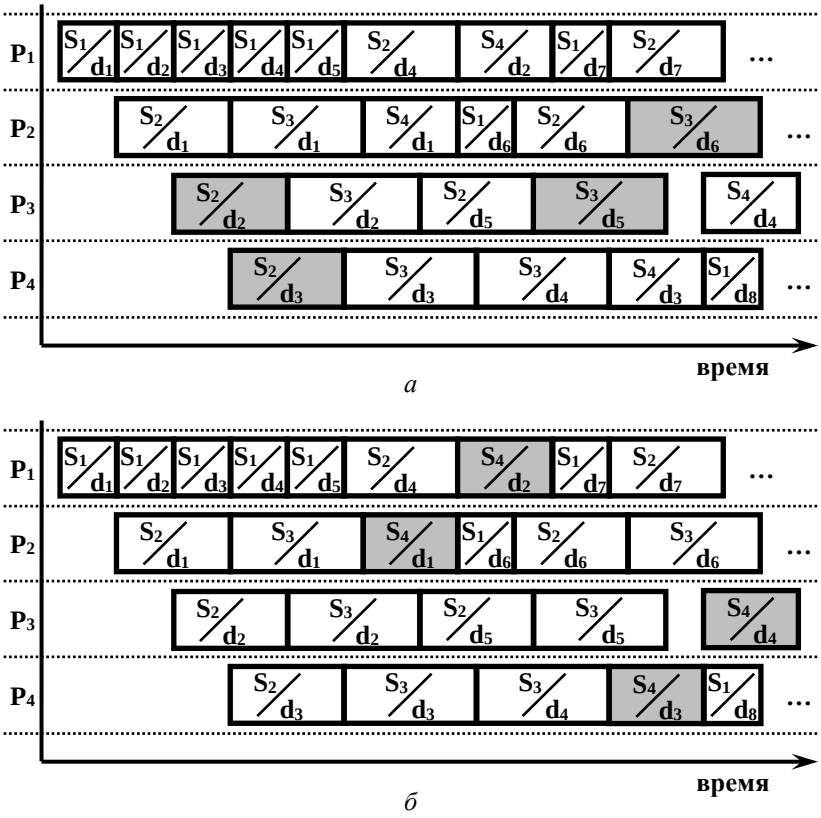


Рис 3.4. Временная диаграмма работы конвейера с динамическим планированием: *а* – параллельное выполнение этапов S_2 и S_3 над разными блоками данных; *б* – сохранение порядка выполнения задач S_4

На рис. 3.4, *а* можно отметить наличие одновременно выполняющихся этапов S_2 и S_3, а на рис. 3.4, *б* видно, что в любой момент времени существует не более одного выполняющегося блока S_4 и порядок их выполнения над блоками данных сохранен, несмотря на выполнение на различных процессорах. Пропуски между блоками обработки почти отсутствуют, а значит, коэффициент использования системы близок к 1 (100 %).

Такое решение тоже не идеально, так как этапы конвейера, отвечающие за операции ввода-вывода, обычно не могут выполняться параллельно сами с собой. Сюда же можно отнести некоторые алгоритмы обработки данных, например, межкадровое сжатие видеопотока, блочные шифры с обратной связью и т.п. Если продолжительность таких этапов значительно больше остальных, то общий выигрыш от параллелизма может оказаться невысоким. В случае, когда дело касается операций ввода-вывода, стоит подумать об их оптимизации (кэширование операций чтения/записи, использование носителей информации с более высоким быстродействием и т.п.).

Построение конвейера в Intel TBB

Для построения конвейера в библиотеке Intel TBB имеется алгоритм **tbb::parallel_pipeline** со следующим синтаксисом:

```
void parallel_pipeline(
   size_t max_number_of_live_tokens,
   const filter_t<void,void>&filter_chain
   [, task_group_context&group]);
```

Параметр **max_number_of_live_tokens** определяет, сколько блоков данных может находиться в обработке одновременно. Этот параметр непосредственно влияет на производительность конвейера, рекомендуется использовать значение больше или равное количеству ядер процессора, умноженное на 2, но не меньше числа этапов конвейера.

Второй параметр **filter_chain** определяет последовательность операций, производимых над данными. Каждая операция, производимая над данными, называется фильтром (filter). В Intel TBB последовательность фильтров должна быть линейной

(ветвления и слияния не допускаются). Фильтр принимает на вход данные и отдает данные на выходе. Тип данных на выходе фильтра предыдущего этапа и на входе следующего должен совпадать. Первый фильтр в цепочке не принимает данные на вход, а последний не отдает данные на выходе. Для каждого фильтра можно указать его тип, всего существует три типа фильтров:

– **parallel**, одновременно может выполняться несколько таких фильтров над разными блоками данных, порядок выполнения не важен;

– **serial_out_of_order**, одновременно может выполняться только один такой фильтр, порядок выполнения не важен, т.е. следующий по порядку блок данных может быть обработан раньше предыдущего;

– **serial_in_order**, одновременно может выполняться только один такой фильтр, при этом сохраняется порядок выполнения, т.е. следующий по порядку блок данных будет обработан только после того, как завершится обработка предыдущего.

Третий параметр **group** не обязательный и служит для указания контекста группы задач, на котором будет выполняться конвейер.

Рассмотрим пример применения конвейера для обработки видеофайла.

Пример 3.1. Использование конвейера для обработки видео

Задача состоит в следующем: требуется открыть видеофайл, применить к нему фильтр изображений и сохранить результат в выходной файл. Фильтр будет применяться к каждому кадру по отдельности (без использования связи между кадрами).

Проанализировав задачу, мы приходим к выводу, что нам потребуется три этапа конвейера для выполнения следующих операций:

– чтение отдельного кадра из видеофайла (последовательно, с сохранением порядка операций);

– применение фильтра (параллельно);

– запись данных в выходной видеофайл (последовательно, с сохранением порядка операций).

Для работы с видео мы воспользуемся библиотекой OpenCV (Open Computer Vision). Перед началом работы конвейера нам необходимо открыть соответствующие файлы на чтение и запись.

```
// открываем входной файл
VideoCapture sourceVideo(argv[1]);
if (!sourceVideo.isOpened())
{
   cout << "Could not open source video file "
        << argv[1] << endl;
   return -1;
}

// получаем четырехсимвольный тип кодека,
// размер кадра и частоту кадров

int fourcc = static_cast<int>(
   sourceVideo.get(CV_CAP_PROP_FOURCC));

int frame_width  = static_cast<int>(
   sourceVideo.get(CV_CAP_PROP_FRAME_WIDTH));

int frame_height = static_cast<int>(
   sourceVideo.get(CV_CAP_PROP_FRAME_HEIGHT));

double fps = sourceVideo.get(CV_CAP_PROP_FPS);

// создаем выходной поток
VideoWriteroutputVideo;
outputVideo.open("output.avi", fourcc, fps,
   Size(frame_width, frame_height), true);

if (!outputVideo.isOpened())
{
   cout  << "Could not open the output video"
         << endl;
   return -1;
}
```

Имя входного файла мы получаем в качестве первого аргумента командной строки, для выходного файла мы используем имя “output.avi”. Все параметры выходного файла (тип используемого кодека, размер и частоту кадров) мы используем такие же, как и у входного файла.

Каждый фильтр конвейера проще оформить в виде отдельного класса с перегруженным оператором (). Рассмотрим фильтр для чтения кадра из потока.

```
class ReadFrameFunc
{
public:
   // конструктор
   ReadFrameFunc(VideoCapture& source)
      : m_source(source) {}

   // конструктор копирования
   ReadFrameFunc(const ReadFrameFunc& f)
      : m_source(f.m_source) {}

   Mat operator()(tbb::flow_control& fc) const
   {
      Mat frame;
      m_source >> frame;

      // если достигнут конец файла,
      // останавливаем работу конвейера
      if (frame.empty()) fc.stop();

      return frame.clone();
   }

private:
    VideoCapture& m_source;
};
```

Примечательным здесь является параметр fc, который передается в перегруженный оператор (). Он используется для остановки работы конвейера. После того, как последний кадр из файла будет прочитан, мы сигнализируем о том, что данные больше не будут поступать на вход конвейера. Реальная работа

конвейера завершится, когда все обрабатываемые блоки пройдут все оставшиеся этапы конвейера. Также можно обратить внимание на тип переменной, где будет храниться прочитанный кадр. Это внутренний тип OpenCV, который при копировании использует технику подсчета ссылок (reference counting), т.е. сам видеокадр при этом скопирован не будет. Для полного копирования необходимо вызвать метод clone.

В качестве фильтра изображения рассмотрим преобразование в оттенки серого, это простая операция, и она не требует дополнительных комментариев:

```
class ImageFilter
{
public:
   Mat operator()(Mat frame) const
   {
      Mat_<Vec3b> frame_ = frame;
      for( int i = 0; i < frame.rows; ++i)
      {
         for( int j = 0; j < frame.cols; ++j )
         {
            float luma = 0;
            luma += 0.2126f * frame_(i,j)[0];
            luma += 0.7152f * frame_(i,j)[1];
            luma += 0.0722f * frame_(i,j)[2];

            unsigned char l =(unsigned char)luma;

            frame_(i,j)[0] = l;
            frame_(i,j)[1] = l;
            frame_(i,j)[2] = l;
         }
      }
      return frame;
   }
};
```

Заключительный этап конвейера (сохранение в видеофайл) тоже довольно прост для понимания. Так же, как и при чтении входного кадра из потока, всю работу за нас выполняет библио-

тека OpenCV. Нам достаточно лишь направить кадр в выходной поток, используя оператор потокового вывода.

```
classWriteFrameFunc
{
public:
   // конструктор
   WriteFrameFunc(VideoWriter&destination)
      : m_destination(destination) {}

   void operator()(Mat frame) const
   {
      m_destination << frame;
   }

private:
   VideoWriter& m_destination;
};
```

Теперь соберем все фильтры в единую цепочку и запустим конвейер:

```
// инициализация планировщика задач TBB
tbb::task_scheduler_initinit;

tbb::filter_t<void, Mat> f1(
tbb::filter::serial_in_order,
   ReadFrameFunc(sourceVideo));

tbb::filter_t<Mat,  Mat> f2(
   tbb::filter::parallel,
   ImageFilter());

tbb::filter_t<Mat,  void> f3(
   tbb::filter::serial_in_order,
   WriteFrameFunc(outputVideo));

tbb::filter_t<void, void> f = f1 & f2 & f3;
tbb::parallel_pipeline(10, f);
```

Тут следует обратить внимание на инициализацию планировщика задач TBB, которая должна быть выполнена до запуска конвейера. В качестве необязательного параметра можно ука-

зать число потоков, которые будут созданы планировщиком. По умолчанию количество создаваемых потоков равно количеству доступных процессоров.

Для первого и последнего фильтра мы указали тип **serial_in_order**, что обеспечит нам правильный порядок кадров в выходном видеофайле. Преобразование в оттенки серого может выполняться параллельно для разных кадров.

И последний момент, который хотелось выделить, – это использование числа 10 в качестве значения параметра **max_number_of_live_tokens**. Таким образом, в данной программе в любой момент времени будут обрабатываться не более 10 кадров.

Задание для лабораторной работы

Требуется модифицировать приведенную выше программу конвейера для последовательного применения двух фильтров изображений. Варианты фильтров выбираются из табл. 3.1 в соответствии с порядковым номером студента в списке по модулю 5.

Таблица 3.1

Варианты фильтров для индивидуального задания

Номер варианта	**Фильтр 1**	**Фильтр 2**
1	Поворот изображения на 90°	Медианный фильтр
2	Коррекция насыщенности	Нерезкое маскирование
3	Коррекция яркости	Размытие по Гауссу
4	Получение негатива изображения	Повышение резкости изображения
5	Гамма-коррекция	Обнаружение краев

Исходный видеофайл выбирается студентом по своему усмотрению исходя из следующих требований:

– разрешение не менее 640x480 точек;

– длительность ролика должна быть выбрана таким образом, чтобы время параллельной обработки со всеми включенными фильтрами составляло около одной минуты;

– должен использоваться распространенный формат сжатия видео, допускающий воспроизведение распространенными мультимедиа-проигрывателями;

– содержание видеоролика не должно нарушать чьих-либо прав или противоречить действующему законодательству.

Имя входного и выходного файлов, а также параметры фильтров (если они необходимы) должны задаваться либо через командную строку, либо при помощи конфигурационного файла в произвольном формате, допускающем редактирование настроек при помощи текстового редактора. Рекомендуется предусмотреть возможность включать/отключать фильтры через конфигурационный файл (без компиляции программы).

С программой необходимо провести следующие эксперименты:

1. Заполнить таблицу 3.2 по результатам работы различных версий программы (при этом следует установить значение параметра **max_number_of_live_tokens** равное числу доступных ядер, умноженному на 2. Объяснить полученные результаты.

Все дальнейшие эксперименты следует проводить с версией программы, где этапы-фильтры выполняются параллельно.

2. Измерить время работы программы с различными значениями параметра алгоритма **max_number_of_live_tokens**: 4, 8, 16, 64, 256. Объяснить полученные результаты и выбрать оптимальное значение параметра.

3. Поменять местами порядок фильтров и определить, изменилось ли время работы программы. Влияет ли порядок применения фильтров на итоговый результат? Объяснить полученные результаты.

4. При наличии возможности проверить, как изменяется время работы программы при использовании 1, 2, 4 и 8(6) потоков. Для этого можно вручную ограничить количество потоков при инициализации планировщика. Построить график зависимости времени работы программы от числа создаваемых потоков.

Таблица 3.2

Результаты выполнения модифицированной программы

Версия программы	**Время выполнения, с**
Последовательная версия программы (без использования IntelTBB)	
Параллельная версия программы, все этапы конвейера выполняются последовательно с сохранением исходного порядка обработки кадров (serial_in_order)	
Параллельная версия программы, этапы конвейера, реализующие фильтры, выполняются последовательно (serial_out_of_order)	
Параллельная версия программы, этапы конвейера, реализующие фильтры, выполняются параллельно (parallel)	

5. Уменьшить длительность видеоролика (числа кадров) до 3 с (примерно 75-90 кадров в зависимости от частоты кадров в секунду). Для этого можно воспользоваться самой программой с отключенными фильтрами и ограничением количества записываемых кадров.

6. Измерить время обработки короткого ролика. Как оно соотносится со временем обработки файла целиком?

7. При наличии параметров у фильтров определить их влияние на время работы алгоритма (2-3 различных значения). Для этого пункта разрешается использование короткого ролика.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Характеристики используемого аппаратного и программного обеспечения.

4. Характеристики выбранного видеофайла: используемый видеокодек, размер кадра (точек), частота кадров (кадры/с), продолжительность (с), общее количество кадров.

5. Заполненная таблица временных затрат для различных версий программы.

6. Таблица и график зависимости времени работы программы от параметра алгоритма **max_number_of_live_tokens**.

7. Время работы программы при изменении порядка фильтров.

8. Таблица и график зависимости времени выполнения программы от количества потоков (при наличии такой возможности).

9. Время работы программы при уменьшении длительности ролика.

10. Таблица и график зависимости времени работы программы от параметров фильтров.

11. Выводы.

12. Приложение. Примеры кадров (с одинаковыми номерами), поступающих на вход каждого из фильтров и результирующего кадра. Подсказка: в библиотеке OpenCV есть необходимые функции для сохранения кадра в растровое изображение.

Вопросы для контроля знаний

1. Приведите примеры использования конвейерной обработки данных.

2. Что является основным достоинством конвейера при параллельной обработке?

3. Какими недостатками обладает модель конвейера со статической привязкой (где каждому этапу конвейера назначается отдельный процессор)?

4. Для чего в конвейрах используются буферы? В каких случаях это может улучшить производительность?

5. За счет чего достигается рост производительности в модели конвейера с динамическим планированием?

6. Как происходит инициализация планировщика задач IntelTBB?

7. Какие ограничения накладывает IntelTBB на последовательность фильтров?

8. Перечислите типы фильтров для построения конвейера, имеющиеся в библиотеке IntelTBB.

9. За что отвечает параметр max_number_of_live_tokens? Как рекомендуется выбирать значение для этого параметра?

10. Как в IntelTBB происходит остановка работы конвейера по окончании обработки последней порции данных?

Лабораторная работа 4

ПОТОКИ В ОС WINDOWS (WINAPI THREADS)

Цели работы

Знакомство с особенностями реализации потоков в ОС Windows и предоставляемыми средствами синхронизации. Получение практического опыта по созданию многопоточных программ для ОС Windows с использованием WinAPI.

Изучение проблем взаимной блокировки и голодания на примере задачи об обедающих философах, причин их возникновения и методов по их предотвращению.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с многоядерным процессором (желательно наличие не менее четырех ядер);

– установленная ОС Windows XP или более поздняя версия;

– среда разработки MS Visual Studio 2008 с поддержкой языка C/C++ или более поздняя версия.

Процессы и потоки

В ОС Windows любое приложение состоит из одного или нескольких процессов. Упрощенно можно сказать, что процесс – это работающая программа. В свою очередь, в рамках одного процесса могут работать один или более потоков. Таким обра-

зом, любой процесс имеет, как минимум, один поток и любой поток принадлежит какому-либо процессу.

Процесс обеспечивает ресурсы, необходимые для выполнения программы. С каждым процессом связаны:

– уникальный идентификатор процесса (целое число);
– приоритет процесса (priority class);
– виртуальное адресное пространство;
– исполняемый код;
– дескрипторы (handles) открытых системных объектов;
– контекст безопасности;
– переменные окружения;
– как минимум, один поток выполнения.

Поток – это сущность внутри процесса, которую можно поставить в очередь на выполнение. Количество одновременно выполняющихся потоков определяется количеством процессоров. Потоки одного процесса разделяют адресное пространство и ресурсы процесса. Каждому потоку соответствуют:

– уникальный идентификатор потока;
– приоритет потока (priority level);
– локальное хранилище потока (TLS = Thread Local Storage);
– контекст выполнения (регистры процессора, стек и т.п.);
– (дополнительно) контекст безопасности.

Вопросы, связанные с контекстом безопасности (правами пользователей, делегирования прав и т.д.), мы рассматривать не будем. Тем не менее в большинстве рассматриваемых нами функций в качестве одного из аргументов будет присутствовать указатель на атрибуты безопасности. В наших примерах мы будем всегда передавать пустой указатель (NULL), что будет означать использование настроек по умолчанию (тех же прав, что и у процесса). В этом случае объекты синхронизации не будут наследоваться дочерними процессами.

Диспетчеризация потоков и приоритеты

В ОС Windows используется вытесняющая многозадачность. Это означает, что при появлении потока с более высоким приоритетом выполнение текущего потока будет приостановлено и начнет выполняться поток с максимальным приоритетом.

Всего существует 32 уровня приоритетов: от 0 (самый низкий) до 31 (самый высокий). Для каждого уровня приоритетов существует отдельная очередь задач. Приоритет потока определяется двумя значениями – приоритетом процесса (priority class) и приоритет потока (priority level) (табл. 4.1).

Таблица 4.1

Определение базового (статического) приоритета потока

	THREAD_PRIORITY_IDLE	THREAD_PRIORITY_LOWEST	THREAD_PRIORITY_BELOW_NORMAL	THREAD_PRIORITY_NORMAL	THREAD_PRIORITY_ABOVE_NORMAL	THREAD_PRIORITY_HIGHEST	THREAD_PRIORITY_TIME_CRITICAL
IDLE_PRIORITY_CLASS	1	2	3	4	5	6	15
BELOW_NORMAL_PRIORITY_CLASS	1	4	5	6	7	8	15
NORMAL_PRIORITY_CLASS	1	6	7	8	9	10	15
ABOVE_NORMAL_PRIORITY_CLASS	1	8	9	10	11	12	15
HIGH_PRIORITY_CLASS	1	11	12	13	14	15	15
REALTIME_PRIORITY_CLASS	16	22	23	24	25	26	31

Как видно из таблицы, приоритет “обычного” потока равен 8. Приоритет 0 имеет только системная задача «Бездействие системы» (System Idle).

Приоритеты потоков, перечисленные в таблице, называются статическими или базовыми. Планировщиком при определении задачи для выполнения на процессоре используются динамиче-

ские приоритеты. По умолчанию динамический приоритет равен базовому. В некоторых случаях (срабатывание блокировки, получение оконного события и т.п.) динамический приоритет может быть временно повышен, но не более чем до 15. После выполнения на процессоре динамический приоритет вновь снижается до базового значения. Приоритеты потоков, относящиеся к классу задач реального времени, не изменяются. Управление динамическими приоритетами (разрешение/запрет изменения приоритетов) осуществляется при помощи специальных функций и может затрагивать как отдельные потоки, так и все потоки, принадлежащие определенному процессу.

Начиная с версий Windows Vista / Windows Server 2008, появились две дополнительные службы, которые также оказывают влияние на диспетчеризацию потоков:

– служба управления потоками (Thread Ordering Service);

– служба планировщика мультимедиа (MultiMedia Class Scheduler Service).

Первая из них предназначена для гарантированного выполнения клиентских потоков, как минимум, один раз за указанный интервал времени. Вторая обеспечивает более приоритетный доступ к процессорному времени для критичных по времени мультимедиа-приложений.

В дополнение к этому в 64-разрядных версиях, начиная с Windows 7 / Windows Server 2008 R2, появилась возможность более гибкого управления выполнением потоков приложения. Эта возможность реализуется через механизм User-Mode Scheduling (UMS) и полезна в первую очередь для высоконагруженных приложений, выполняющих большое количество потоков на разных процессорах.

Средства синхронизации

В ОС Windows доступны разнообразные средства синхронизации межпоточного взаимодействия (табл. 4.2). Некоторые из средств синхронизации потоков могут использоваться также и при синхронизации между процессами (потоками разных процессов). Они выделены в отдельный класс – объекты синхронизации.

Таблица 4.2

Средства синхронизации межпоточного взаимодействия

Версия операционной системы		Windows XP	Windows Vista	Windows 7	Windows 8	Windows Server 2003	Windows Server 2008	Windows Server 2012
Объекты синхронизации	События	+	+	+	+	+	+	+
	Мьютексы	+	+	+	+	+	+	+
	Семафоры	+	+	+	+	+	+	+
	Таймеры ожидания	+	+	+	+	+	+	+
Атомарные операции		+	+	+	+	+	+	+
Критические секции		+	+	+	+	+	+	+
Блокировки чтения-записи		–	+	+	+	–	+	+
Условные переменные		–	+	+	+	–	+	+
Барьеры синхронизации		–	–	–	+	–	–	+

Объекты синхронизации являются объектами ОС, имеют связанный с ними дескриптор и имя (в виде строки), чтобы на них можно было ссылаться из разных процессов. В примерах мы будем использовать пустое имя, так как все взаимодействие будет выполняться в рамках одного процесса.

Для работы с объектами синхронизации используются функции ожидания. Функция ожидания приостанавливает работу вызвавшего его потока до тех пор, пока не выполнится связанное с ней условие. В примерах мы будем использовать в основном функции ожидания одного или нескольких объектов. Эти функции могут также использоваться для ожидания завершения работы потока (потоков), поскольку они являются системными объектами.

События

Событие – это объект синхронизации с методами для явного изменения статуса. События бывают двух типов:

– с ручным сбросом состояния (manual-reset event), в этом случае статус события не изменится до вызова соответствующей функции сброса статуса события;

– с автоматическим сбросом состояния (auto-reset event), в этом случае статус события сбросится после того, как один из потоков, ожидающих этого события, будет разблокирован; в случае, если события ожидали сразу несколько потоков, нельзя однозначно сказать, какой из них будет разблокирован.

Мьютексы и семафоры

Мьютексы и семафоры являются широко распространенными средствами синхронизации, поэтому рассмотрим только особенности их реализации в ОС Windows.

Если поток, захвативший мьютекс, по каким-либо причинам завершился, не освободив его, то мьютекс будет разблокирован автоматически, но при этом функция ожидания вернет специальное значение WAIT_ABANDONED. Как правило, такая ситуация сигнализирует об ошибке и о неопределенном состоянии общего ресурса.

Поток может многократно пытаться захватить мьютекс, и при этом все повторные попытки захвата мьютекса будут успешными (поток не будет переведен в режим ожидания). При каждой попытке захвата семафора его счетчик будет уменьшаться независимо от того, захватывает его этот же поток или другой.

Только поток, захвативший мьютекс, может его освободить. Функция освобождения семафора может быть вызвана из любого потока.

Если в функцию ожидания нескольких объектов передать несколько раз дескриптор одного и тоже семафора, при его захвате счетчик уменьшиться только на 1.

Таймеры ожидания

Таймеры ожидания – это объекты синхронизации, позволяющие заблокировать поток на определенное время. Бывают двух типов:

– таймер с ручным сбросом (статус таймера не изменится до вызова функции SetWaitableTimer);

– синхронизирующий таймер (статус таймера сбросится автоматически после разблокировании потока).

Любой из этих двух типов таймеров может быть таймером с периодическим или однократным срабатыванием.

Атомарные операции

Можно считать атомарными операции чтения и записи правильно выравненных в памяти 32-разрядных чисел на 32-битных версиях Windows и 64-разрядных чисел на 64-битных версиях. Помимо этого есть набор функций для выполнения атомарных операций, объединенных в Interlocked API.

Как правило, каждая функция в Interlocked API существует в нескольких вариантах для работы с данными разной разрядности и разными вариантами использования барьеров памяти. В именах функций, работающих с размерами данных, отличных от 32 бит, присутствует суффикс, обозначающий разрядность операнда (8, 16, 64, 128).

Барьеры памяти реализуются компилятором на уровне инструкций процессора и служат гарантией того, что при параллельном выполнении все операции до барьера завершатся перед началом выполнения операций после барьера. Например, для процессора с поддержкой SSE2 это будут инструкции mfence (memory fence), lfence (load fence) и sfence (store fence).

Функции в Interlocked API можно разделить на следующие группы:

– операции инкремента/декремента: InterlockedIncrement и InterlockedDecrement;

– логические операции: InterlockedAnd, InterlockedOr и InterlockedXor;

– операции установки нового значения: InterlockedExchange и InterlockedExchangePointer;

– арифметические операции: InterlockedExchangeAdd и InterlockedExchangeSubtruct;

– битовые операции: InterlockedBitTestAndComplement, InterlockedBitTestAndSet и InterlockedBitTestAndReset;

– операции “сравнить и установить”: InterlockedCompareExchange и InterlockedCompareExchangePointer.

Критические секции

Критическая секция может использоваться только потоками одного процесса и гарантирует, что только один из потоков может находиться внутри критической секции.

Критическая секция может быть заменена на мьютекс, однако в целях повышения производительности рекомендуется использовать критическую секцию там, где это возможно. Отличие состоит в том, что у критической секции есть механизм, позволяющий избежать блокировки потока в случае, если другой поток покинет критическую секцию в течение определенного количества циклов ожиданий.

Блокировки чтения-записи

Являются готовым механизмом для решения проблемы доступа из разных потоков к одной и той же области памяти на чтение или запись (задача читателей-писателей). Более подробно задача о читателях и писателях будет рассмотрена в следующей лабораторной работе. Справедливое обслуживание так же, как и обслуживание в порядке очереди поступления заявок не гарантируется. Преимущество отдается потокам, желающим произвести запись (обновить значение).

Одну и ту же блокировку чтения-записи можно использовать в двух режимах:

– режим совместного доступа (shared mode), когда несколько потоков получают одновременно доступ на чтение;

– режим эксклюзивного доступа (exclusive mode), когда только один поток получает доступ на запись.

Условные переменные

Условная переменная используется совместно либо с критической секцией, либо с блокировкой чтения записи. Фактически условную переменную можно рассматривать как реализацию совместного использования события и критической секции (или блокировки чтения-записи).

Представим себе такую ситуацию: поток вошел в критическую секцию и, находясь в ней, ожидает наступления какого-либо события, например, освобождения места в кольцевом бу-

фере. Если ожидание может продлиться довольно долго, то в целях повышения эффективности работы программы следует выйти из критической секции и дождаться наступления события, после чего вновь войти в критическую секцию. Условная переменная позволяет выполнить эти операции как единое целое. Критическая секция будет освобождена до наступления события, а поток будет переведен в режим ожидания. После наступления события вход в критическую секцию будет выполнен автоматически.

Барьеры синхронизации

Барьер является простым механизмом синхронизации, он приостанавливает дальнейшее выполнение потоков до момента, когда необходимое число потоков достигнет барьера.

Потоки в ожидании прохождения барьера могут продолжать выполняться (проверять в цикле условие барьера) либо могут быть переведены в режим ожидания, тогда они больше не будут потреблять процессорное время. Перевод потоков у барьера в режим ожидания рекомендуется в случаях, когда ожидание прохождения барьера может продлиться долго.

На сегодняшний день барьеры поддерживаются только в версиях Windows 8 и Windows Server 2012. В более ранних версиях ОС схожее поведение можно реализовать при помощи использования других средств синхронизации.

Приостановление работы потока

Поток может приостановить свою работу на определенное время (в мс) при помощи функций Sleep или SleepEx.

Если в качестве значения времени передать 0, то оставшаяся часть от выделенного планировщиком временного интервала (time slice) будет выделена другому потоку. А в следующей итерации работы планировщика поток будет конкурировать за процессорное время наравне с другими потоками, как и при обычном исполнении.

Для приостановления работы потока из другого потока может использоваться функция SuspendThread. Поскольку в этом случае нет возможности контролировать точку в программе, где

исполнение потока будет прервано, ее рекомендуется использовать только в целях отладки.

Завершение работы потока

Поток прекратит свою работу в следующих случаях:

– при вызове функции ExitThread из самого потока;

– при вызове функции ExitProcess из любого потока данного процесса (в этом случае все потоки этого процесса будут завершены);

– поток возвратит управление из главной функции потока (выполнит оператор return);

– любой поток вызовет функцию TerminateThread с дескриптором данного потока;

– любой поток вызовет функцию TerminateProcess с дескриптором данного процесса (в этом случае все потоки этого процесса будут завершены).

В двух последних случаях ресурсы, связанные с потоком или процессом, не будут освобождены корректно. Поэтому эти функции рекомендуется использовать только в экстренных случаях.

Для нормального завершения работы рекомендуется создать событие и отслеживать состояние этого события во всех потоках при помощи функции WaitForSingleObject с нулевым временем ожидания. В случае наступления этого события функция вернет значение WAIT_OBJECT_0, после чего поток должен самостоятельно завершить свою работу.

При завершении работы потока выполняются следующие действия:

– освобождаются все ресурсы, связанные с потоком (окна и т.п.);

– устанавливается код возврата потока;

– сигнализируется о завершении работы потока;

– если поток является единственных потоком процесса, завершается работа процесса и выполняются все связанные с этим действия.

Пример 4.1. Подсчет количества простых чисел в указанном диапазоне

Мы уже рассматривали данную задачу ранее, посмотрим, как будет выглядеть ее реализация на WinAPI.

Теперь нам придется самим распределить задачу между работающими потоками. Для передачи в поток границ интервала воспользуемся следующей структурой:

```
// Структура передаваемая в поток
// в качестве параметра
struct SearchParams
{
   unsigned long start;  // начало интервала
   unsigned long end;    // конец интервала
};
```

Нам потребуется массив таких структур с количеством элементов, равным числу создаваемых потоков N_THREADS. Также нам понадобятся массивы для сохранения дескрипторов и идентификаторов создаваемых потоков:

```
// данные для работы потоков
SearchParamsdata[N_THREADS];

// дескрипторы потоков
HANDLE hThreadArray[N_THREADS];

// идентификаторы потоков
DWORD  dwThreadIdArray[N_THREADS];
```

Функция потока незначительно будет отличаться от кода, который мы использовали в прошлый раз. Основное отличие в том, как передаются аргументы в функцию потока.

Из прошлого опыта мы знаем, что необходимо синхронизировать доступ из разных потоков к переменной, где будет храниться количество найденных простых чисел. Здесь мы применим критическую секцию. В отличие от OpenMP критические секции в WinAPI приостанавливают выполнение потока не сразу, а лишь по истечении заданного количества циклов ожидания (указывается при ее инициализации).

```
DWORD WINAPI ThreadFunction(LPVOID lpParam)
{
   // Приведение указателя к нужному типу
   SearchParams* pInterval =
      reinterpret_cast<SearchParams*>(lpParam);

   // Выводим информацию о параметрах поиска
   printf_s("Thread %6d, Interval [%5d, %5d]\n",
      GetCurrentThreadId(),
      pInterval->start,
      pInterval->end);

   // Подсчитываем количество чисел в диапазоне
   for (unsigned long number = pInterval->start;
      number <= pInterval->end; ++number)
   {
      bool is_prime = true;
      long last =
         (long)floor(sqrt((double)number));

      for (long j=2; j<=last && is_prime; ++j)
      {
         if ((number % j) == 0)
            is_prime = false;
      }
      if (is_prime)
      {
         EnterCriticalSection(&cs);
         ++nPrimeNumbers;
         LeaveCriticalSection(&cs);
      }
   }
   return 0;
}
```

Функция GetCurrentThreadId вернет идентификатор текущего потока. Использование критической секции тривиально и дополнительных пояснений не требует.

Критическая секция должна быть объявлена и проинициализирована ранее в основном потоке программы. Одной тысячи

циклов ожидания будет более чем достаточно для такой простой операции как инкремент. В случае, если по каким-либо причинам инициализировать критическую секцию не удалось, то завершаем работу всего приложения.

```
// Критическая секция
CRITICAL_SECTION  cs;

// Инициализируем критическую секцию
if (!InitializeCriticalSectionAndSpinCount(
   &cs,   // указатель на критическую секцию
   1000   // количество циклов ожидания
   ))
{
   printf("InitializeCriticalSectionAndSpinCount
failed (%d)\n", GetLastError());
   ExitProcess(1);
}
```

Теперь, когда все предварительные приготовления сделаны, можно приступить непосредственно к созданию потоков. Для создания потоков воспользуемся функцией CreateThread. Первый параметр этой функции – это указатель на структуру с атрибутами безопасности; мы будем использовать те же настройки, что и у текущего процесса, поэтому передаем NULL. Размер стека мы также не указываем явно, по умолчанию размер стека будет равен 1 Мб. Следующие два параметра – это основная функция потока и передаваемый в нее аргумент, в нашем случае это указатель на структуру SearchParams. Использование флага CREATE_SUSPENDED означало бы, что поток не следует запускать на выполнение сразу после создания, а только после вызова функции ResumeThread. При резервировании памяти под стек указанного размера следовало бы указать флаг STACK_SIZE_PARAM_IS_A_RESERVATION. Так как мы не указываем размер стека явно, этого нам не требуется. Таким образом, при передаче в качестве флагов 0 поток начнет свою работу сразу после создания. Наконец, последний параметр функции – это указатель на переменную, куда мы сохраняем идентификатор потока. В случае, когда он нам не требуется, можно передать NULL.

```
// min_num - нижняя граница интервала
// max_num - верхняя граница интервала

// количество чисел для проверки одним потоком
const long N = (max_num - min_num)/N_THREADS;

// Создание требуемого (N_THREADS) числа потоков
for (unsigned int i = 0; i < N_THREADS; ++i)
{
   // распределение задач между потоками
   data[i].start = min_num + i*N;
   data[i].end = (i == N_THREADS-1) ? max_num :
                 (data[i].start + N - 1);

   // Создание нового потока
   hThreadArray[i] = CreateThread(
      NULL,              // настройки безопасности
      0,                 // размер стека
      ThreadFunction,    // имя функции потока
      &data[i],          // аргумент функции потока
      0,                 // флаги
      &dwThreadIdArray[i]);

   // Если поток не был создан - завершаем работу
   if (hThreadArray[i] == NULL)
      ExitProcess(1);
}
```

После создания дополнительных потоков основной поток программы продолжит работу. Мы приостановим его работу до завершения остальных потоков, а потом освободим все связанные с ними ресурсы.

```
// Ожидаем завершения всех потоков
WaitForMultipleObjects(
   N_THREADS,     // количество объектов (потоков)
   hThreadArray,  // дескрипторы потоков
   TRUE,          // ждем завершения всех потоков
   INFINITE);     // ждем бесконечно
```

```
// Освобождаем дескрипторы потоков
for (int i = 0; i < N_THREADS; ++i)
   CloseHandle(hThreadArray[i]);

// удаляем критическую функцию
DeleteCriticalSection(&cs);
```

Наибольший интерес здесь представляет собой функция ожидания нескольких объектов, в данном фрагменте в качестве объектов выступают сами потоки. Первый параметр – это количество объектов, которые мы ждем, а второй – это указатель на первый элемент массива дескрипторов этих объектов. Третий параметр определяет, следует ли ждать сигналов от всех объектов, либо достаточно сигнала от одного из них (любого). Четвертый параметр используется для указания времени ожидания. Если ожидание небесконечно (как в нашем случае), то необходимо проверить значение, возвращаемое функцией, таким образом, можно будет узнать, были ли получены сигналы от указанных объектов или истекло время ожидания. В качестве длительности ожидания можно передать 0, фактически это будет означать проверку статуса объекта без какого-либо дополнительного ожидания.

Собрав все части программы вместе, запустим ее для поиска чисел в интервале [2, 1000] с числом потоков N_THREADS = 3. Ниже приведен результат работы.

```
Searching prime numbers in interval [2, 1000]
Thread    728, Interval [    2,   333]
Thread   3276, Interval [  334,   665]
Thread   2544, Interval [  666,  1000]
Total number of prime numbers found: 168
```

Здесь видно, как именно была поделена вся работа между потоками. Результат работы соответствует ожидаемому.

Вместо критической секции для предотвращения “гонок” можно было воспользоваться атомарной операцией инкремент – вот как в этом случае выглядел бы фрагмент кода:

```
      if (is_prime)
         InterlockedIncrement(&nPrimeNumbers);
```

Измерение временных интервалов

Для измерения временных интервалов с высокой точностью в WinAPI имеется функция QueryPerformanceCounter. Пример ее использования (без проверки ошибок) приведен ниже.

```
LARGE_INTEGER frequency; // разрешение таймера
LARGE_INTEGER t0;        // начало расчетов
LARGE_INTEGER t1;        // окончание расчетов

QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&t0);

// участок кода для оценки времени выполнения
...

QueryPerformanceCounter(&t1);

double seconds = t1.QuadPart - t0.QuadPart;
seconds = seconds/frequency.QuadPart;

// вывод затраченного времени
printf("Execution time   : %.5f s\n", seconds);
```

С помощью этой функции можно получить доступ к высокоточному таймеру; частота этого таймера может быть получена при помощи метода QueryPerformanceFrequency. Это позволяет перевести время из тактов таймера в секунды.

Ошибки синхронизации

Мы уже рассмотрели одну ошибку, связанную с отсутствием синхронизации, – ситуацию “гонок”. Существует еще несколько ситуаций, вызванных неправильной синхронизацией взаимодействия параллельно выполняющихся задач.

Взаимная блокировка (deadlock)

Данная ситуация возникает, когда две или более параллельно выполняющиеся задачи пытаются одновременно получить доступ к двум неразделяемым ресурсам. Представим, что для успешного выполнения необходимо захватить оба ресурса. Если

первая задача успела захватить первый ресурс, а вторая – второй, то обе задачи будут бесконечно ждать взаимного освобождения ресурсов.

Примером из реального мира может служить руководство для машинистов поездов (начало 20 века): “Когда оба поезда одновременно приближаются к перекрестку, оба поезда должны остановиться до тех пор, пока другой поезд не проедет”. Другим примером из литературы может служить “Уловка 22” из одноименного романа американского писателя Джозефа Хеллера.

Обнаружение взаимной блокировки – непростая задача, так как данная ситуация проявляется лишь при определенном стечении внешних обстоятельств. Тем не менее существует несколько техник по выявлению взаимных блокировок. Например, одна из таких техник основана на модели конечного автомата и выявлении тупиковых состояний.

Также разработано множество алгоритмов для предотвращения взаимных блокировок, некоторые из них мы рассмотрим ниже.

Livelock

Существует особая разновидность взаимной блокировки – livelock. В этой ситуации задачи не остаются в состоянии ожидания, а циклически меняют свое состояние, однако при этом также не могут получить необходимые для выполнения задачи ресурсы. Например, если при неудачном захвате второго ресурса первый ресурс освобождается и предпринимается попытка вначале захватить другой ресурс, то обе задачи могут оказаться в ситуации, когда они будут попеременно обладать первым и вторым ресурсом, но не обоими ресурсами сразу.

Примером из реальной жизни может служить ситуация, когда два человека сталкиваются в узком коридоре и из вежливости пытаются пропустить друг друга, при этом оба они делают шаги в одну и ту же сторону, чтобы уступить дорогу другому.

Голодание (starvation)

Еще одна ситуация, связанная с распределением ресурсов, – голодание. При этой ситуации задача не может получить желаемый ресурс по причине того, что этот ресурс постоянно выделя-

ется другим задачам, но не выделяется текущей. В результате так же, как и в предыдущих случаях, задача не может быть завершена.

Решением этой проблемы является увеличение приоритета задачи, долго ожидающей ресурса. Таким образом, рано или поздно она будет обладать максимальным приоритетом среди других задач и получит доступ к ресурсу.

Задача об обедающих философах

Рассмотренные выше ошибки синхронизации и алгоритмы их разрешения проще рассмотреть на примере классической задачи об обедающих философах. Пять безмолвных философов сидят вокруг круглого стола. Перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов (рис. 4.1). Каждый философ может либо есть, либо размышлять.

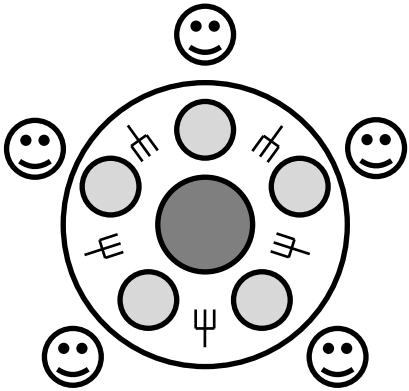


Рис. 4.1. Иллюстрация задачи об обедающих философах

Приём пищи не ограничен количеством оставшихся спагетти (подразумевается бесконечный запас). Тем не менее философ может есть только тогда, когда держит две вилки, взятые справа и слева. Каждый философ может взять ближайшую вилку (если она доступна) или положить (если он уже держит её). Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим. Суть

проблемы заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

Можно предложить следующий алгоритм действия философам:

– размышлять в течение заданного времени;
– ждать пока освободится левая вилка и взять ее;
– ждать пока освободится правая вилка и взять ее;
– есть в течение заданного времени;
– положить левую вилку;
– положить правую вилку;
– повторить все сначала.

Такое решение потенциально опасное и может привести к взаимной блокировке – ситуации, когда все философы будут иметь по одной вилке и ждать, когда освободится вторая. Тем не менее, попробуем реализовать этот вариант и посмотрим, как возникает взаимная блокировка. Итак, нам понадобится создать 5 задач-философов и 5 мьютексов-вилок. Кроме этого, мы будем использовать два события: одно для одновременного запуска потоков, второе – для завершения работы потоков. В качестве аргумента в функцию потока мы будем передавать структуру, где будет указан порядковый номер задачи и дескрипторы правой и левой вилок.

```
// Сообщение об окончании создания всех потоков
HANDLEhEventStart;

// Сообщение об окончании работы всех потоков
HANDLEhEventStop;

// структура, передаваемая в поток
structForksInfo
{
   int      index;  // порядковый номер философа
   HANDLE   left;   // дескрипотор левой вилки
   HANDLE   right;  // дескрипотор правой вилки
};
```

Код создания событий, мьютексов и потоков довольно типичен и не представляет большого интереса. Наибольший интерес для нас представляет функция самого потока-философа.

```
DWORD WINAPI PhilosopherThread(LPVOID lpParam)
{
   ForksInfo* pInfo =
      reinterpret_cast<ForksInfo*>(lpParam);
   write_log(pInfo->index, "thread is created");

   // ждем завершения запуска остальных задач
   DWORDdwWaitResult = WaitForSingleObject(
      hEventStart, INFINITE);

   while (true){
      // размышляем в течении заданного времени
      write_log(pInfo->index, "is thinking");
      Sleep(THINKING_TIME_MS);
      write_log(pInfo->index, "became hungry");

      // пытаемся взять левую вилку
      if (take_fork(pInfo, LEFT)) return 0;
      Sleep(FORK_PAUSE_MS);

      // пытаемся взять правую вилку
      if (take_fork(pInfo, RIGHT)) return 0;

      // едим в течении заданного времени
      write_log(pInfo->index, "iseating");
      Sleep(EATING_TIME_MS);
      write_log(pInfo->index, "is not hungry");

      // освобождаем левую вилку
      ReleaseMutex(pInfo->left);
      write_log(pInfo->index, "put left fork");
      Sleep(FORK_PAUSE_MS);

      // освобождаем правую вилку
      ReleaseMutex(pInfo->right);
      write_log(pInfo->index, "put right fork");
   }
}
```

Функция write_log носит вспомогательный характер и служит для вывода сообщений в файл журнала. Рассмотрим подробнее, что происходит внутри функции взятия вилки take_fork:

```
enum { LEFT = 0, RIGHT = 1};

// Функция взятия вилки
bool take_fork(ForksInfo* info, int fork)
{
   // дескриптор мьютекса-вилки
   HANDLE hFork = (fork == LEFT) ?
      info->left : info->right;

   while (true)
   {
      // проверка на завершение работы потоков
      DWORDdwWaitResult = WaitForSingleObject(
         hEventStop, 0);
      if (dwWaitResult == WAIT_OBJECT_0)
         return true;

      // ожидание вилки
      dwWaitResult = WaitForSingleObject(
         hFork, MUTEX_WAIT_TIME_MS);
      if (dwWaitResult == WAIT_OBJECT_0)
         break;

      if (dwWaitResult == WAIT_TIMEOUT)
         write_log(info->index, (fork == LEFT) ?
            "can`t take left fork" :
            "can`t take right fork");
   }

   write_log(info->index, (fork == LEFT) ?
      "has taken left fork" :
      "has taken right fork");
   return false;
}
```

Перечисляемый тип мы используем только для того, чтобы придать коду выразительность. Ожидание вилки представляет

собой бесконечный цикл, из которого можно выйти в двух случаях: произошло событие “завершение работы” или если удалось взять вилку. Для определения причины, по которой мы вышли из этой функции, будем использовать возвращаемое значение: в первом случае мы вернем true (в этом случае поток должен будет завершить свою работу), а во втором – false (в этом случае поток продолжит свою работу).

Ждать вилку в течение бесконечного времени было бы плохой идеей: в этом случае мы не смогли бы нормально обработать событие “завершение работы” и не смогли бы обнаруживать состояние, когда философ в течение уже продолжительного времени не может взять вилку.

Вот что мы получим в результате запуска программы:

```
0.000 [2200] Philosopher 0 thread is created
0.000 [ 724] Philosopher 2 thread is created
0.000 [3972] Philosopher 1 thread is created
0.000 [ 716] Philosopher 4 thread is created
0.000 [2712] Philosopher 3 thread is created
0.099 [3972] Philosopher 1 is thinking
0.099 [2200] Philosopher 0 is thinking
0.100 [ 716] Philosopher 4 is thinking
0.100 [2712] Philosopher 3 is thinking
0.100 [ 724] Philosopher 2 is thinking
0.141 [2200] Philosopher 0 became hungry
0.141 [3972] Philosopher 1 became hungry
0.141 [2712] Philosopher 3 became hungry
0.141 [ 716] Philosopher 4 became hungry
0.141 [ 724] Philosopher 2 became hungry
0.141 [2200] Philosopher 0 has taken left fork
0.141 [3972] Philosopher 1 has taken left fork
0.141 [2712] Philosopher 3 has taken left fork
0.141 [ 716] Philosopher 4 has taken left fork
0.141 [ 724] Philosopher 2 has taken left fork
1.161 [2200] Philosopher 0 can`t take right fork
1.161 [ 716] Philosopher 4 can`t take right fork
1.161 [ 724] Philosopher 2 can`t take right fork
1.161 [3972] Philosopher 1 can`t take right fork
1.161 [2712] Philosopher 3 can`t take right fork
```

Невозможность взять вилку одним конкретным философом еще не является взаимной блокировкой, а только подозрением на нее. С другой стороны, такая ситуация явно свидетельствует об “аномальном” поведении и, как минимум, является признаком “голодания”. Признаком взаимной блокировки является невозможность взять вилку всеми философами, что мы и увидели в данном случае.

Существует несколько вариантов решения этой задачи, позволяющих избежать взаимной блокировки. Помимо этого возможна и ситуация “голодания” – когда в результате несправедливого распределения ресурсов (вилок), один (или более) из философов всегда будет голодать, в то время как остальные будут успешно чередовать размышления с приемами пищи.

Решение с использованием семафора

Это достаточно простое решение заключается в том, чтобы использовать семафор с начальным значением счетчика 4. Семафор используется для того, чтобы ограничить число философов, владеющих вилками (одной или двумя). Счетчик семафора уменьшается, когда философ берет первую вилку, и увеличивается, когда он кладет обе вилки на стол. Для простоты можно договорится, что вначале всегда берется левая вилка, а затем правая.

Решение с использованием случайного ожидания

При этом варианте решения задачи философ должен положить уже взятую вилку, если с момента ее взятия прошло больше заданного времени и вторую вилку взять не удалось. После этого философ ждет некоторое время в заданном интервале и предпринимает попытку снова.

Решение с использованием иерархии ресурсов

Другое простое решение заключается в назначении иерархии ресурсам: вилки нумеруются от 1 до 5. Философ, который собирается начать есть, должен вначале взять вилку с меньшим номером и лишь затем – с большим. По окончании еды философ вначале кладет на стол вилку с большим номером, а затем с меньшим.

Решение с использованием иерархии потоков

Данное решение очень похоже на предыдущее, но в данном случае нумеруются философы, а не вилки. Философы с четными номерами вначале берут правую вилку, а затем левую и освобождают их в обратном порядке. Тогда как философы с нечетными номерами поступают наоборот: они вначале берут левую вилку, а затем правую.

Решение с использованием монитора

При этом способе решения философ ждет, когда обе вилки будут свободны, и берет их только в случае, если обе вилки свободны. Поскольку по условию задачи взятие вилок должно быть раздельными действиями, то требуется монитор для отслеживания состояния вилок и синхронизации операции по взятию обеих вилок со стола. Таким образом, в мониторе определены две операции: “ждать и взять вилки, как только они освободятся” и “положить вилки на стол”.

Задание для лабораторной работы

Требуется модифицировать приведенную программу для решения задачи обедающих философов одним из рассмотренных вариантов решения. Вариант выбирается в соответствии с номером студента по списку (по модулю 5):

1) решение с использованием семафора;
2) решение с использованием случайного ожидания;
3) решение с использованием иерархии ресурсов;
4) решение с использованием иерархии потоков;
5) решение с использованием монитора.

Программа должна вести журнал всех действий философов с временными отметками для возможности анализа ситуации. По окончании работы программы файл журнала будет использоваться для анализа полученного решения.

Используя собранную в журнале информацию, необходимо для каждой из задач философов:

1) определить количество приемов пищи;
2) определить суммарное время ожидания приема пищи (с момента, как он стал голоден, до момента начала приема пищи);

3) вычислить среднее значение длительности ожидания;

4) построить гистограмму распределения длительности ожидания.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Индивидуальное задание.
4. Описание произведенных модификаций.
5. Результаты работы программы: количество приемов пищи, длительность ожидания (суммарное, среднее и гистограмма распределения) для каждого философа.
6. Выводы.
7. Приложение 1. Фрагмент журнала работы программы.
8. Приложение 2. Текст программы.

Вопросы для контроля знаний

1. Опишите характеристики процессов и потоков в ОС Windows. Как процессы и потоки связанны межу собой?
2. Как происходит диспетчеризация потоков в ОС Windows?
3. Перечислите средства синхронизации межпоточного взаимодействия в ОС Windows.
4. Какие из средств синхронизации могут использоваться потоками разных процессов?
5. Каковы отличительные особенности реализации мьютексов и критических секций?
6. Опишите назначение параметров функции CreateThread.
7. Что такое функции ожидания? Как функцию ожидания можно использовать для ожидания завершения работы потоков?
8. Дайте определения понятиям “взаимная блокировка” (deadlock) и livelock. Приведите примеры.
9. В чем заключается решение задачи об обедающих философах?
10. Что такое “голодание” (starvation)? Как можно выявить данную ситуацию в программе? Какие существуют способы для борьбы с голоданием?

Лабораторная работа 5

POSIX THREADS

Цели работы

Знакомство с особенностями реализации потоков в POSIX и предоставляемыми средствами синхронизации. Получение практического опыта по созданию многопоточных программ с использованием POSIX Threads.

Изучение проблем организации совместного доступа к разделяемому ресурсу на примере задач о производителях-потребителях и читателях-писателях.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с многоядерным процессором (желательно наличие не менее четырех ядер);

– установленная ОС Windows или Linux (рекомендуется);

– компилятор для языка C/C++ с поддержкой pthreads (рекомендуется использовать GCC).

Стандарт POSIX

POSIX (Portable Operating System Interface (for UniX)) представляет собой набор стандартов, обеспечивающих переносимость прикладных программ на уровне исходного кода. Первоначально стандарт состоял из нескольких частей, текущая версия IEEE Std 1003.1-2008 объединяет все части. Стандарт описывает интерфейс служб ядра (core services), оболочку (shell), утилиты (utils), потоки (threads extensions) и расширение реального времени (real-time extensions). Мы в данной работе затронем только ту часть стандарта, которая непосредственно связана с созданием многопоточных приложений.

Реализация потоков в ОС Linux основана на данном стандарте. Также существуют реализации этого стандарта под другие операционные системы. Например, в MinGW имеется частичная поддержка этого стандарта для ОС Windows.

Состояния потока

В POSIX поток может находиться в одном из четырех состояний:

– готовый к выполнению (ready) – поток ожидает своей очереди для доступа к процессору, он мог быть только что запущен, разблокирован или вытеснен другим потоком;

– выполняющийся (running) – поток выполняется в данный момент времени, количество выполняющихся одновременно потоков определяется количеством процессоров (ядер);

– заблокированный (blocked) – поток ждет наступления какого-либо события (условной переменной, освобождения мьютекса, окончания операции ввода-вывода) и не может выполняться в данный момент;

– завершенный (terminated) – поток завершился или его выполнение было остановлено, но данные, связанные с потоком, не были удалены, ожидается вызов либо для слияния с другим потоком, либо для отвязывания потока от процесса.

Условия перехода от одного состояния к другому отображены на рисунке 5.1.

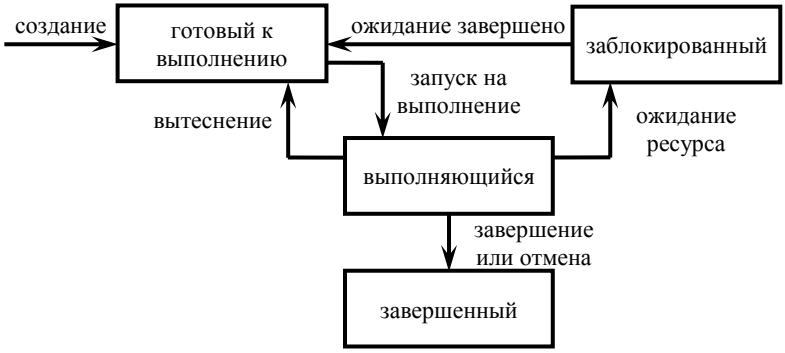


Рис. 5.1. Граф состояний потока

Диспетчеризация потоков

Модель диспетчеризации основана на очередях с приоритетами. Для каждого уровня приоритета существует отдельная

очередь. Общее количество уровней приоритета зависит от конкретной реализации, обычно их не менее 32.

Потоки в очереди обслуживаются в соответствии с указанной для них политикой диспетчеризации. Таким образом, в один и тот же момент в системе могут существовать два потока (или более), обслуживающихся по разным политикам диспетчеризации. Система может поддерживать разные политики диспетчеризации (зависит от реализации). В стандарте определены четыре базовые политики:

– SCHED_FIFO – обслуживание в порядке очереди поступления;

– SCHED_RR – карусельная диспетчеризация;

– SCHED_SPORADIC – диспетчеризация в режиме спорадического сервера (обработка непериодических задач);

– SCHED_OTHER – зависящая от реализации.

Отметим, что очереди на выполнение для всех потоков общие (не зависят от политики).

Обслуживание в порядке очереди

При данной политике задачи будут обслуживаться в том же порядке, в каком они поступили на выполнение. При выполнении на одном процессоре второй поток не начнет выполняться до тех пор, пока первый не завершит свою работу или не будет заблокирован.

Ниже приведен полный список правилами, в соответствии с которыми осуществляется диспетчеризация потоков при этой политике:

– при вытеснении, выполняющийся поток помещается в начало очереди;

– при разблокировании, поток добавляется в конец очереди;

– при изменении приоритета при помощи вызова функции pthread_setschedprio, поток добавляется в начало очереди, если его приоритет вырос, или в конец очереди, если его приоритет понизился;

– при изменении приоритета потока другими способами, он добавляется в конец очереди;

– при вызове функции sched_yield, поток добавляется в конец очереди.

Карусельная диспетчеризация

В отличие от предыдущей политики, здесь гарантируется, что ни один из потоков с одинаковыми приоритетами не сможет монополизировать процессор. Отличие состоит в том, что если текущий поток выполнялся в течение времени, большем, чем возвращает функция sched_rr_get_interval, то он будет вытеснен и помещен в конец очереди, а поток, находящийся в начале очереди, будет отправлен на выполнение.

Таким образом, гарантируется, что все готовые к выполнению потоки, выполнятся по одному разу, прежде чем первый поток продолжит свое выполнение.

Диспетчеризация в режиме спорадического сервера

Если первые две политики реализуются практически всегда, то реализация режима спорадического сервера встречается довольно редко (преимущественно в ОС реального времени). Чаще всего данный режим используется в тех случаях, когда надо гарантировать минимальное время выполнения потока за указанный интервал времени. При этом реальное время работы потока может оказаться как меньше, если поток закончил вычисления раньше этого времени, так и больше, если ему удалось выполнить часть работы в фоновом режиме.

Рассмотрим общий принцип управления выполнением потоков в этом режиме (рис. 5.2).

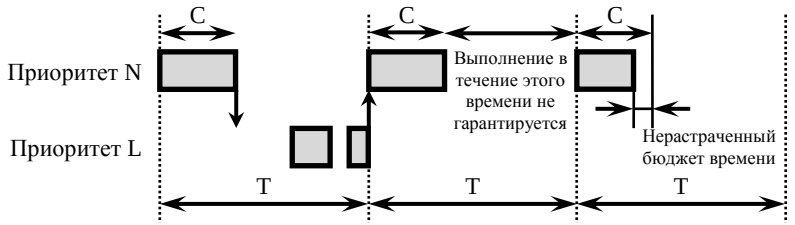


Рис. 5.2. Выполнение потока в режиме спорадического сервера: N – нормальный приоритет потока; L – фоновый приоритет потока; T – интервал пополнения времени; C – первоначальный бюджет времени

В этом режиме потоку назначается два приоритета: обычный (sched_priority) и фоновый (sched_ss_low_priority). Также потоку назначается первоначальный бюджет времени (sched_ss_init_budget), интервал пополнения времени (sched_ss_repl_period) и количество пополняемого времени (replenish_amount).

Когда бюджет времени больше нуля, поток выполняется с обычным приоритетом, в других случаях приоритет потока понижается до фонового, пока бюджет времени не восполнится. В остальном диспетчеризация выполняется аналогично обслуживанию в порядке очереди.

Создание потоков и атрибуты потока

Поток создается путем вызова функции pthread_create, вот как она объявлена:

```
int pthread_create(
   pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine) (void *),
   void *arg);
```

Как мы видим, у нее всего четыре параметра. Первый параметр – это указатель на связанную с потоком переменную. Эта переменная позволяет в последствии однозначно идентифицировать поток и используется в большинстве функций работы с потоками. Второй параметр позволяет управлять атрибутами создаваемого потока. Если передать нулевой указатель, то будет создан поток с атрибутами по умолчанию. Третий и четвертый параметры – это главная функция потока и передаваемые в нее аргументы соответственно.

В качестве дополнительных атрибутов потока при создании можно указать:

– состояние потока: отвязанный от текущего потока (detached) или объединяемый (joinable);

– наследование параметров диспетчеризации (scheduling inheritance);

– политика диспетчеризации (scheduling policy);

– параметры диспетчеризации (scheduling parameters);

– контекст диспетчеризации (scheduling contention scope);
– размер стека (stack size);
– адрес стека (tack address);
– размер защитной области стека (stack guard size).

Отличие между состояниями потоков заключается в том, как будут освобождены связанные с потоком ресурсы. Если поток отвязан от других потоков, то ресурсы освобождаются сразу после его завершения. Ресурсы объединяемого потока освобождаются только после вызова функции pthread_join из другого потока, с которым он объединяется. В большинстве случаев рекомендуется создавать объединяемые потоки, что позволит легко убедиться в их завершении перед завершением работы программы.

Поток может наследовать параметры диспетчеризации или переопределить их. Также можно указать контекст диспетчеризации: системный (PTHREAD_SCOPE_SYSTEM) или контекст процесса (PTHREAD_SCOPE_PROCESS). В случае системного контекста поток будет конкурировать за ресурсы со всеми другими потоками всех процессов, работающих в этом же контексте. В случае контекста процесса ему придется конкурировать только с другими потоками этого же процесса. По стандарту в системе требуется реализация, как минимум, одного из контекстов. В ОС Linux реализован и используется системный контекст (единый для всех потоков всех процессов).

Дополнительно можно указать размер стека и размер защитной области стека – специальной области, располагаемой непосредственно за стеком и используемой для борьбы с переполнением стека. Также можно предварительно выделить память под стек и указать ее адрес непосредственно. В ОС Linux/x86-32 по умолчанию выделяется стек в 2 Мб, а размер защитной области равен одной странице памяти.

Завершение работы потока

Существует несколько причин, по которым поток может прекратить свою работу:

– было возвращено управление из главной функции потока (оператор return);

– поток вызвал функцию завершения потока pthread_exit;

– работа потока была прервана из другого потока при помощи функции pthread_cancel;

– работа основного потока (функции main) была завершена раньше окончания работы других потоков;

– работа процесса была приостановлена вызовом функции exec или exit.

Наибольший интерес для рассмотрения представляет прерывание работы одного потока из другого. По ходу работы прерываемый поток мог выполнить ряд операций, которые могут повлиять на дальнейшую работу программы, если они не будут правильно обработаны. Например, поток мог выделить память или захватить какой-либо разделяемый ресурс. Если мы прерываем работу такого потока, то необходимо освободить выделенную память и занимаемый ресурс.

Управлять поведением прерываемого потока можно при помощи вызова функций pthread_setcancelstate и pthread_setcanceltype. Первая функция определяет, можно ли прерывать работу потока (PTHREAD_CANCEL_ENABLE) или нет (PTHREAD_CANCEL_DISABLE). Во втором случае вызов функции pthread_cancel будет просто проигнорирован. При помощи второй функции можно указать, как именно будет прервано выполнение потока. Выполнение можно прервать асинхронно (PTHREAD_CANCEL_ASYNCHRONOUS). В этом случае выполнение будет прервано в любом месте и никакой дополнительной обработки произведено не будет. Поэтому этот режим используют крайне редко. А в случае режима PTHREAD_CANCEL_DEFERRED выполнение потока будет приостановлено в строго определенных местах – точках отмены. Точка отмены представляет собой вызов одной из определенных функций, например, pthread_testcancel. Таким образом, можно указать места в программе, где выполнение потока может быть безопасно остановлено. Также при помощи функций pthread_cleanup_push, pthread_cleanup_pop можно управлять стеком функций-обработчиков, которые будут автоматически вызваны при прерывании выполнения.

Средства синхронизации

В POSIX определены следующие средства синхронизации:
– барьер синхронизации;
– мьютекс;
– блокировка чтения-записи;
– условная переменная;
– однократная инициализация.

За исключением мьютексов и однократной инициализации, остальные средства синхронизации довольно типичны и идеологически мало чем отличаются от рассмотренных ранее.

Также можно обратить внимание на отсутствие среди стандартных средств синхронизации семафоров, которые доступны через расширения реального времени.

Мьютексы

Функционирование мьютекса в POSIX определяется несколькими независимыми друг от друга параметрами:
– типом мьютекса;
– способом взаимодействия с потоками других процессов;
– поведением при завершении захватившего его потока;
– политикой поведения и приоритетом.

Все характеристики мьютекса задаются через специальную структуру с описанием атрибутов, которая затем указывается при его создании.

Тип мьютекса определяет его поведение при повторном захвате мьютекса этим же потоком, освобождении незахваченного мьютекса и освобождении мьютекса, захваченного другим потоком. Всего существует четыре типа мьютекса:

– PTHREAD_MUTEX_NORMAL – попытка захвата уже захваченного мьютекса этого типа приведет к взаимной блокировке, при освобождении незахваченного мьютекса или мьютекса, захваченного другим потоком, поведение не определено;

– PTHREAD_MUTEX_ERRORCHECK – функция захвата уже захваченного мьютекса вернет ошибку; функция освобождения незахваченного мьютекса или мьютекса, захваченного другим потоком, такжевернут ошибку;

– PTHREAD_MUTEX_RECURSIVE – захват уже захваченного мьютекса будет успешным, при этом функция освобождения мьютекса должна быть вызвана соответствующее число раз; функция освобождения незахваченного мьютекса или мьютекса, захваченного другим потоком, вернут ошибку;

– PTHREAD_MUTEX_DEFAULT – во всех случаях поведение не определено; в зависимости от реализации в качестве типа по умолчанию может быть назначен один из трех вышеперечисленных типов.

По отношению к взаимодействию с потоками других процессов мьютексы бывают:

– PTHREAD_PROCESS_SHARED, доступные любому потоку, имеющему доступ до участка памяти, где мьютекс был создан;

– PTHREAD_PROCESS_PRIVATE, доступные только потокам того же процесса.

В случае, если поток, захвативший мьютекс, завершился, не освободив его, и при этом имеются другие потоки, ожидающие этот же мьютекс, дальнейшее поведение определяется устойчивостью мьютекса:

– PTHREAD_MUTEX_STALLED, никаких дополнительных действий не производится, возможна ситуация взаимной блокировки;

– PTHREAD_MUTEX_ROBUST, ожидающему мьютекс вернется значение EOWNERDEAD.

Политику поведения и приоритеты мьютексов мы рассмотрим чуть ниже, в главе “Инверсия приоритетов”.

Однократная инициализация

Некоторые операции в программе должны быть выполнены однократно (инициализация мьютексов, создание ключей для доступа к данным других потоков и т.п.).

При инициализации приложения это условие легко выполнимо: все операции можно выполнить из функции main перед созданием потоков.

В случае создания библиотеки выполнить эти требования сложнее. Альтернативным решением может стать использование булевой переменной совместно с мьютексом со статической

инициализацией, но использование специального контекста управления однократной инициализации является более простым решением.

Инверсия приоритетов

Пусть у нас имеются три потока, которые выполняются на одном процессоре.

Поток Thread 1 имеет самый низкий приоритет из трех, его выполнение было прервано после того, как он захватил разделяемый ресурс (мьютекс).

Поток Thread 2 с наиболее высоким приоритетом, который прервал выполнение потока Thread 1. Для продолжения работы ему потребовался разделяемый ресурс, захваченный потоком Thread 1. В результате данный поток был заблокирован до освобождения ресурса.

Поток Thread 3 со средним приоритетом, в результате сложившейся ситуации получил все процессорное время.

Сложившаяся ситуация называется инверсией приоритетов – на выполнение был отправлен поток с более низким приоритетом, а поток с максимальным приоритетом был заблокирован. Для распространения этой ситуации на многопроцессорные системы достаточно создать количество потоков со средним приоритетом (Thread 3), равное или большее числу процессоров.

Для борьбы с инверсией приоритетов в POSIX используются приоритеты мьютексов. Существует три политики, определяющие поведение потока при захвате мьютекса.

Атрибут PTHREAD_PRIO_NONE не оказывает никакого воздействия на приоритет захватившего его потока.

Поток, захвативший мьютекс (или несколько мьютексов) с атрибутом PTHREAD_PRIO_INHERIT, будет выполняться с приоритетом, который определяется как максимум из приоритета текущего потока и максимальным приоритетом потока, ожидающим мьютекс.

Поток, захвативший мьютекс (или несколько мьютексов) с атрибутом PTHREAD_PRIO_PROTECT, будет выполняться с приоритетом, который определяется как максимум из приоритета текущего потока и приоритета захваченного мьютекса. Для

избежания инверсии приоритетов, приоритет мьютекса в этом случае должен быть выше или равен максимальному приоритету потока, использующему данный мьютекс.

Попытаемся создать программу, которая наглядно бы продемонстрировала ситуацию с инверсией приоритетов. Поскольку нам надо прервать работу потока с низким приоритетом в строго определенном месте, воспользуемся для этого барьером синхронизации.

Также это гарантирует нам, что все потоки уже будут созданы и запущены к этому моменту. Количество потоков, встречающихся у барьера, равно 4, потому что в дополнение к трем потокам добавляется основной поток программы. Дополнительные атрибуты барьера нам не нужны, поэтому мы их не указываем и передаем вместо них нулевой указатель.

Всегда необходимо проверять результат вызова любой функции POSIX: если вызов прошел успешно, то функция вернет 0, иначе будет возвращен код ошибки.

В результате мы получим следующий фрагмент кода:

```
// объявляем барьер
pthread_barrier_t barrier;

// инициализируем барьер
int res = pthread_barrier_init(
   &barrier,   // указатель на барьер
   NULL,       // указатель на атрибуты
   4);         // количество потоков

if (res != 0)
{
   printf("pthread_barrier_init failed (%d)\n",
      res);
   return 0;
}
```

Теперь рассмотрим, как реализованы основные функции потоков.

Начнем с потока с низким приоритетом. Здесь следует обратить внимание на то, что барьер синхронизации расположен после захвата мьютекса. Это гарантирует захват мьютекса низ-

кориоретным потоком раньше, чем это сможет сделать поток с высоким приоритетом.

```
// функция низкоприоритетного потока
void * low_priority_thread(void * )
{
   printf("Low priority thread started\n");

   // захватываем мьютекс
   int res = pthread_mutex_lock(&mutex);
   if (res != 0)
      printf("pthread_mutex_lock failed (%d)\n",
         res);
   else
   {
      printf("Low priority thread succsessfully
locked mutex\n");

      // ждем пока другие потоки будут созданы
      pthread_barrier_wait(&barrier);

      // эмулируем вычислительный процесс
      process_data(1000);

      // разблокируем мьютекс
      printf("Low priority thread unlocking
mutex\n");

      res = pthread_mutex_unlock(&mutex);
      if (res != 0)
         printf("pthread_mutex_lock failed
(%d)\n", res);
   }

   printf("Low priority thread ended\n");
   pthread_exit(0);
}
```

Далее рассмотрим поток с высоким приоритетом. Он отличается лишь расположением барьера синхронизации.

```
// функция высокоприоритетного потока
void * high_priority_thread(void * )
{
   // ждем у барьера пока
   // низкоприоритетныйпоток захватит мьютекс
   // и все остальные потоки будут созданы
   pthread_barrier_wait(&barrier);

   printf("High priority thread started\n");

   // захватываем мьютекс
   int res = pthread_mutex_lock(&mutex);
   if (res != 0)
      printf("pthread_mutex_lock failed (%d)\n",
         res);
   else
   {
      printf("High priority thread succsessfully
lock mutex\n");

      // эмулируем вычислительный процесс
      process_data(1000);

      // разблокируем мьютекс
      printf("High priority thread unlocking
mutex\n");
      res = pthread_mutex_unlock(&mutex);
      if (res != 0)
         printf("pthread_mutex_lock failed
(%d)\n", res);
   }

   printf("High priority thread ended\n");
   pthread_exit(0);
}
```

И, наконец, рассмотрим функцию низкоприоритетного потока. Она отличается отсутствием кода для работы с мьютексом и является самой простой из всех.

```
// функция потока со средним приоритетом
void * mid_priority_thread(void * )
{
   // ждем у барьера пока
   // низкоприоритетный поток захватит мьютекс
   // и все остальные потоки будут созданы
   pthread_barrier_wait(&barrier);

   int i;
   printf("Mid priority thread started\n");

   // эмулируем вычислительный процесс
   process_data(2000);

   printf("Mid priority thread ended\n");

   pthread_exit(0);
}
```

Во всех трех потоках нам необходимо эмулировать некоторый вычислительный процесс. Для этого мы используем созданную нами функцию process_data. Отметим, что данная функция очень похожа на фрагмент кода, который мы использовали ранее для проверки чисел на простоту.

```
//////////////////////////////////////////////////
// функция эмулирования вычислительного процесса
//////////////////////////////////////////////////
void process_data(unsigned int n)
{
    for (unsigned int i = 0; i < n; ++i)
    {
       long test = 1;
       for (long l = 1; l < 1000000; ++l)
             test = test % l;
    }
}
```

Для определения приоритетов дополнительных потоков воспользуемся несложными расчетами. Все потоки должны выполняться в режиме карусельной диспетчеризации.

```
// выбираем карусельную политику диспетчеризации
intpolicy = SCHED_RR;

// получаем минимальный и максимальный приоритеты
// потков для выбранной политики диспетчеризации
min_priority = sched_get_priority_min(policy);
max_priority = sched_get_priority_max(policy);

// вычисляем приоритеты для создаваемых потоков
mid_priority  = (min_priority + max_priority)/2;
low_priority  = (mid_priority + min_priority)/2;
high_priority = (mid_priority + max_priority)/2;
```

Теперь можно перейти к созданию мьютекса с определенным набором атрибутов. В следующем фрагменте кода обработка ошибок пропущена для ясности. В реальных программах всегда следует проверять код возврата любой POSIX функции!

```
// мьютекс
pthread_mutex_tmutex;

// атрибуты мьютекса
pthread_mutexattr_t mutex_attr;

// инициализируем атрибуты  мьютекса
res = pthread_mutexattr_init(&mutex_attr);

// указываем протокол работы мьютекса
res = pthread_mutexattr_setprotocol(
   &mutex_attr,
   PTHREAD_PRIO_NONE);

// указываем приоритет мьютекса
res = thread_mutexattr_setprioceiling(
   &mutex_attr,
   mid_priority);

// инициализируем мьютекс
res = pthread_mutex_init(&mutex, &mutex_attr);
```

При создании потоков также необходимо указать дополнительные атрибуты: политику диспетчеризации и приоритеты, – обработка ошибок пропущена для ясности:

```
// идентификаторы потоков
pthread_t  threads[3];

// аттрибуты потоков
pthread_attr_t   pthread_attr;

// инициализируем аттрибуты потоков
res = pthread_attr_init(&pthread_attr);

// разрешаем дочерним потоки иметь собственную
// дисциплину диспетчеризации, которая может
// отличатся от дисциплины основного потока
res = pthread_attr_setinheritsched(
   &pthread_attr,
   PTHREAD_EXPLICIT_SCHED);
// устанавливаем дисциплину диспетчеризации
// дочерних потоков
res = pthread_attr_setschedpolicy(
   &pthread_attr,
   SCHED_RR);

// параметры диспетчеризации
sched_param param;
param.sched_priority = low_priority;

// устанавливаем приоритет
res = pthread_attr_setschedparam(
   &pthread_attr,
   &param);

res = pthread_create(
   &threads[0],          // идентификатор потока
   &pthread_attr,        // аттрибуты потока
   &low_priority_thread, // функция потока
   NULL);        // функция потока без аргумента
```

Атрибуты других потоков не отличаются ничем, кроме приоритета. В приведенном примере мы рассмотрели только созда-

ние низкоприоритетного потока, остальные потоки создаются сходным образом. Потоки начнут работать сразу после создания и выполниться до барьера, где будет ожидать других потоков.

Основной поток также ждет у барьера, а после его прохождения ждет завершения работы всех других потоков. Ожидание других потоков осуществляется при помощи функции **pthread_join**. Идентификатор потока передается в качестве первого аргумента, а указатель на возвращаемое значение – в качестве второго. В нашем случае функции потока ничего не возвращают, поэтому мы передаем нулевой указатель. Остается лишь удалить структуры и завершить работу:

```
// ждем у барьера
pthread_barrier_wait(&barrier);

// ожидание завершения потоков
for (int i = 0; i< 3; ++i)
   res = pthread_join(threads[i], NULL);

// удаление структур
pthread_attr_destroy(&pthread_attr);
pthread_mutexattr_destroy(&mutex_attr);
pthread_mutex_destroy(&mutex);
pthread_barrier_destroy(&barrier);

// завершение работы программы
return 0;
```

Собрав программу вместе и выполнив ее, мы получим следующие результаты:

```
Low priority thread started
Low priority thread succsessfully locked mutex
High priority thread started
Mid priority thread started
Mid priority thread ended
Low priority thread unlocking mutex
High priority thread succsessfully lock mutex
High priority thread unlocking mutex
High priority thread ended
Low priority thread ended
```

Как видно из результатов, поток со средним приоритетом завершился раньше высокоприоритетного. Нам удалось успешно воспроизвести ситуацию инверсии приоритетов.

Теперь поменяем протокол работы мьютекса на PTHREAD_PRIO_INHERIT и посмотрим, как это повлияет на результат работы. Теперь приоритет низкоприоритетного потока должен временно увеличиться до максимального приоритета потока, ожидающего этот мьютекс. Это обеспечит его выполнение до тех пор, пока он не освободит мьютекс. Это, в свою очередь, разблокирует высокоприритетный поток, который и завершит свою работу первым. Вот, что мы в итоге получим:

```
Low priority thread started
Low priority thread succsessfully locked mutex
High priority thread started
Low priority thread unlocking mutex
High priority thread succsessfully lock mutex
High priority thread unlocking mutex
High priority thread ended
Mid priority thread started
Mid priority thread ended
Low priority thread ended
```

Последовательность, в которой потоки завершили свою работу, соответствует их приоритетам.

Изучение политики PTHREAD_PRIO_PROTECT остается для самостоятельного рассмотрения читателя. При предыдущих двух политиках приоритет мьютекса фактически не использовался, но он является важным для работы в третьем режиме.

Классическая задача «производитель-потребитель»

Две задачи – производитель и потребитель используют общий кольцевой буфер фиксированного размера. Производитель генерирует данные и помещает их в буфер, а потребитель извлекает данные из буфера по одной порции данных за раз. Сложность заключается в том, что производитель не должен помещать новую порцию данных, если буфер уже полон, а потребитель не должен пытаться извлечь данные из пустого буфера.

Кольцевой буфер реализуем с помощью массива и двух индексов – указателей на начальный и конечный элемент соответственно.

Напомним, что для правильной работы этого буфера нам необходимо наличие в нем, как минимум, одного пустого элемента. Без этого невозможно различать ситуации полного и пустого буфера, когда оба указателя указывают на один и тот же элемент. Таким образом, для определения заполненности буфера мы будем сравнивать его размер с BUFFER_SIZE - 1.

```
// размер циклического буфера
const int BUFFER_SIZE = 8;

// циклический буфер
int buffer[BUFFER_SIZE];

// индекс начала буфера
unsigned int queue_start = 0;

// индекс конца буфера
unsigned int queue_end   = 0;
```

Количество элементов в буфере определяется следующим образом:

```
// функция для определения размера очереди
unsigned int get_queue_size()
{
   return (queue_start <= queue_end) ?
      queue_end - queue_start :
      queue_end + BUFFER_SIZE - queue_start;
}
```

Поскольку кольцевой буфер является разделяемым ресурсом, то доступ к нему следует защищать мьютексом. Функция определения количества элементов также не должна вызываться параллельно с другими операциями (добавлением/удалением элементов), так как в этом случае она может вернуть неправильное значение.

Тогда работа потока-производителя могла бы состоять из последовательности следующих шагов:

– генерировать новый элемент (прочитать из входного файла новую порцию данных и т.п.);

– захватить мьютекс;

– проверить размер буфера;

– если буфер полон (нет свободного места), освободить мьютекс и ждать появления свободного места, после чего вновь захватить мьютекс и продолжить работу;

– поместить новый элемент в буфер;

– освободить мьютекс;

– продолжить с начала, пока не все данные будут обработаны (не достигнут конец файла и т.п.).

В случае полного буфера нам приходиться ждать появления свободного места, это может произойти только когда поток-потребитель извлечет очередной элемент из буфера. В этот же момент он может известить поток-производитель (просигнализировать) о возможности продолжить работу. В свою очередь, мы должны проинформировать поток-потребитель о наличии в буфере элементов, после помещения нового элемента в буфер.

Для выполнения операции “освободить мьютекс, ждать появления свободного места, после чего вновь захватить мьютекс и продолжить работу” идеально подойдет условная переменная. Нам понадобятся две условные переменные: одна из них будет связана с условием “в буфере есть свободное место для элемента”, а вторая – с условием “в буфере есть необработанные элементы”.

Для условных переменных и мьютекса можно использовать статическую инициализацию. Это избавляет от необходимости вызывать соответствующие методы create и destroy, но не позволяет указать дополнительные атрибуты. В данном случае нам этого и не требуется:

```
// условная переменная для непереполненого буфера
pthread_cond_t queue_not_full_cond =
    PTHREAD_COND_INITIALIZER;

// условная переменная для непустого буфера
pthread_cond_t queue_not_empty_cond =
    PTHREAD_COND_INITIALIZER;
```

```
// мьютекс для доступа к очереди
pthread_mutex_tqueue_mutex =
   PTHREAD_MUTEX_INITIALIZER;
```

Рассмотрим функции потоков производителя и потребителя, которые во многом схожи.

```
// функция потока-производителя
void * producer(void *)
{
   write_log("PRODUCER", "Thread started");

   // Цикл производства
   while (true) {
      // проверка на остановку производства
      if (stop_production) break;

      pthread_mutex_lock(&queue_mutex);

      // если очередь переполнена - ждем
      while (get_queue_size() == BUFFER_SIZE - 1)
      {
         write_log("PRODUCER", "Waiting");
         pthread_cond_wait(
            &queue_not_full_cond,
            &queue_mutex);
      }

      // добавляем новый элемент в очередь
      buffer[queue_end] = ++item;
      queue_end = (queue_end + 1) % BUFFER_SIZE;
      write_log("PRODUCER", "New item ready");
      pthread_mutex_unlock(&queue_mutex);

      // посылаем сигнал потоку-потребителю
      pthread_cond_signal(&queue_not_empty_cond);

      Sleep (my_rand() % PRODUCER_SLEEP_TIME_MS);
   }
   write_log("PRODUCER", "Thread exit");
   pthread_exit(0);
}
```

```
// функция потока-потребителя
void * consumer(void *)
{
   write_log("CONSUMER", "Thread started");

   // Цикл потребления
   while (true) {
      pthread_mutex_lock(&queue_mutex);

      // проверка на окончание работы
      if (stop_production
         && get_queue_size() == 0) break;

      // если очередь пуста - ждем
      while (get_queue_size() == 0)
      {
         write_log("CONSUMER", "Waiting");
         pthread_cond_wait(
            &queue_not_empty_cond,
            &queue_mutex);
      }

      // удаляем обработанный элемент из очереди
      queue_start = (queue_start + 1) %
         BUFFER_SIZE;
      write_log("CONSUMER", "Item processed");
      pthread_mutex_unlock(&queue_mutex);

      // посылаем сигнал потоку-производителю
      pthread_cond_signal(&queue_not_full_cond);

      Sleep (my_rand() % CONSUMER_SLEEP_TIME_MS);
   }
   write_log("CONSUMER", "Thread exit");
   pthread_exit(0);
}
```

Следует обратить внимание на использование условной переменной внутри цикла проверки условия. Эта дополнительная проверка нужна для случая, когда по сигналу условной пере-

менной могут разблокироваться сразу несколько потоков. Такой прием является типовым при использовании условных переменных.

Функция write_log носит вспомогательный характер и используется для журналирования работы программы. Функция Sleep представляет собой обертку вокруг стандартной функции nanosleep для удобства работы. Меняя значения констант PRODUCER_SLEEP_TIME_MS (максимальная пауза между циклами производства) и CONSUMER_SLEEP_TIME_MS (максимальная пауза между циклами потребления), можно добиться ситуаций, когда будет простаивать поток-производитель или поток-потребитель.

Булевая переменная stop_production используется для остановки работы потоков из основного потока по запросу от пользователя. В этом случае поток-производитель сразу прекращает свою работу, а поток-потребитель продолжает работу до тех пор, пока в буфере остаются необработанные элементы.

Классическая задача о читателях-писателях

Данная задача является одной из важнейших среди задач параллельного программирования. Пусть имеется область памяти, позволяющая чтение и запись. Несколько потоков имеют к ней доступ. Потоки, читающие из памяти, называются читателями, а потоки, производящие запись, – писателями. В момент записи только один процесс-писатель имеет доступ к памяти, тогда как при чтении из памяти несколько процессов-читателей могут одновременно иметь доступ (на чтение). Самым простым решением будет использование мьютекса для разграничения доступа к ресурсу, в этом случае только один поток может получить доступ к ресурсу в любой момент времени.

Это порождает первую проблему читателей-писателей: процесс-читатель не должен ждать, если общий ресурс открыт на чтение. Другими словами, чтение могут выполнять несколько потоков одновременно. Для решения этой проблемы можно воспользоваться парой мьютексов – один для записи, второй для подсчета потоков, открывших ресурс на чтение. Когда первый поток-читатель пытается открыть ресурс на чтение, он блокиру-

ет запись. Разблокировка записи происходит, когда последний из потоков-читателей завершает доступ к ресурсу. Такое решение проблемы называется решением с преимуществом читателей (readers preference). В результате мы получим следующий псевдокод:

```
mutex rc, wr;
int readcount = 0;

reader()
{
   lock(rc);
      ++readcount;
      if (readcount==1) lock(wr);
   unlock(rc);

   // операция чтения

   lock(rc);
      --readcount;
      if (readcount==0) unlock(wr);
   unlock(rc);
}

writer()
{
   lock(wr);
      // операция записи
   unlock(wr);
}
```

Теперь потоки могут читать параллельно друг с другом, но если потоки-читатели будут интенсивно считывать данные, то может сложиться следующая ситуация. В любой момент времени будет существовать один или более потоков-читателей, открывших ресурс на чтение, в результате – поток-писатель никогда не получит доступ к ресурсу. Это порождает вторую проблему читателей-писателей: процесс-писатель не должен ждать более, чем это необходимо (кроме завершения текущих операций чтения).

Чтобы решить вторую проблему, можно добавить еще пару мьютексов – для чтения и для счетчика ждущих записи потоков. Ниже приведен пример псевдокода:

```
mutex rc, wc, rd, wr;
int readcount = 0, writecount = 0;

reader()
{
   lock(rd);
      lock(rc);
         ++readcount;
         if (readcount==1) lock(wr);
      unlock(rc);
   unlock(rd);

   // операция чтения

   lock(rc);
      --readcount;
      if (readcount==0) unlock(wr);
   unlock(rc);
}

writer()
{
   lock(wc);
      ++writecount;
      if (writecount==1) lock(rd);
   unlock(wc);

   lock(wr);
      // операция записи
   unlock(wr);

   lock(wc);
      --writecount;
      if (writecount==0) unlock(rd);
   unlock(wc);
}
```

Теперь первый поток-писатель будет блокировать доступ при новых запросах на чтение и останется дождаться только завершения текущих операций чтения. Сниматься такая блокировка будет при отсутствии других потоков-писателей после завершения записи. Данное решение получило название – решение с преимуществом писателей (writers preference).

В этом случае потоки-писатели могут монополизировать ресурс, так что потоки-читатели не получат к нему доступ. Однако в большинстве реальных систем на одну операцию записи приходится несколько операций чтения. К тому же часто не имеет смысл читать “старые” данные, которые скоро должны обновиться, поэтому часто останавливаются на этом решении.

Третья проблема читателей-писателей призвана обеспечить равноправное обслуживание: ни один из потоков не должен ждать слишком долго, т.е. поток должен получить доступ к разделяемому ресурсу в течение конечного времени. Псевдокод для ее решения выглядит следующим образом:

```
mutex rc, rd, wr;
int readcount = 0;

reader()
{
   lock(rd);
   {
      lock(rc);
         ++readcount;
         if (readcount == 1) lock(wr);
      unlock(rc);
   }
   unlock(rd);
   // операция чтения
   {
      lock(rc);
         --readcount;
         if (readcount == 0) unlock(wr);
      unlock(rc);
   }
}
```

```
writer()
{
   lock(rd);
      lock(wr);
   unlock(rd);
   // операция записи
    unlock(wr);
}
```

Если внимательно разобрать псевдокод, то можно заметить, что различие между потоком-читателем и потоком-писателем заключается преимущественно в операциях с мьютексом rd: это необходимо для обеспечения параллельного доступа на чтение.

В POSIX для решения схожих задач предусмотрен готовый механизм синхронизации – блокировка чтения-записи, которая работает с преимуществом писателей. Впоследствии ее использование позволит сильно упростить код и обойтись всего одним примитивом синхронизации.

```
// блокировка чтения-записи
pthread_rwlock_trwlock =
   PTHREAD_RWLOCK_INITIALIZER;

// функция потока-читателя
void * reader_thread(void * arg)
{
   unsigned int *n_read = (unsigned int *)arg;

   while (!abort_all_threads)
   {
      // блокируем на чтение
      int res = pthread_rwlock_rdlock(&rwlock);
      if (res == 0)
      {
         ++(*n_read);
         Sleep(READ_TIME_MS);
         res = pthread_rwlock_unlock(&rwlock);
      }
   }
   pthread_exit(0);
}
```

```
// функция потока-писателя
void * writer_thread(void * arg)
{
   unsigned int *n_write = (unsigned int *)arg;

   while (!abort_all_threads)
   {
      // блокируем на запись
      int res = pthread_rwlock_wrlock(&rwlock);
      if (res == 0)
      {
         ++(*n_write);
         Sleep(WRITE_TIME_MS);

         res = pthread_rwlock_unlock(&rwlock);
      }
   }
   pthread_exit(0);
}
```

Блокировку чтения-записи можно захватить в одном из двух режимов – на чтение (из потоков-читателей) или на запись (из потоков-писателей). Чтобы иметь возможность судить о справедливости распределения доступа между потоками, мы добавим код для подсчета количества операций чтения/записи из каждого потока. Для моделирования длительности операций чтения/записи мы будем использовать временную задержку при помощи функции Sleep.

В целях эксперимента выберем время чтения READ_TIME_MS и время записи WRITE_TIME_MS, равное 20 мс. Также ограничим общее время эксперимента 5000 мс. Тогда максимально возможное количество операций записи в течение эксперимента составит 5000 мс / 20 мс = 250 (суммарно для всех потоков). Теоретический максимум для операций чтения составит 250 операций на каждый поток, так как операции чтения могут выполняться параллельно.

При запуске 4 потоков-читателей и 4 потоков-писателей получим следующие результаты:

```
Reader thread 0 created
Reader thread 1 created
Reader thread 2 created
Reader thread 3 created
Writer thread 0 created
Writer thread 1 created
Writer thread 2 created
Writer thread 3 created
Number of reads by thread 0: 1
Number of reads by thread 1: 1
Number of reads by thread 2: 1
Number of reads by thread 3: 1
Total number of reads: 4
Number of writes by thread 0: 67
Number of writes by thread 1: 50
Number of writes by thread 2: 69
Number of writes by thread 3: 54
Total number of writes: 240
```

Как и ожидалось, писатели монополизировали доступ к ресурсу, при этом распределение доступа внутри класса задач-писателей можно считать равномерным. Количество операций записи составило 240, что очень близко к теоретическому максимуму. Не будем забывать, что данная ситуация была смоделирована специально, в большинстве реальных задач такое поведение (непрерывное обновление ресурса) возникает редко.

Задание для лабораторной работы

В зависимости от номера студента по списку (четный или нечетный) требуется реализовать решение одной из классических задач, рассмотренных ранее, с некоторыми модификациями.

Независимо от варианта требуется организовать запись в файл журнала информации об операциях, произведенных каждым потоком с указанием времени начала и окончания операции (подсказка – признаком окончания операции может быть начало следующей операции, производимой этим же потоком).

Помимо информации о действиях потоков, в файл журнала должно записываться состояние ресурса. Для задачи «производитель-потребитель» состоянием ресурса является количество

порций данных в кольцевом буфере, а для задачи «читатели-писатели» состоянием ресурса является текущий режим доступа к ресурсу (чтение/запись/простой). Каждое изменение состояния ресурса должно отражаться в файле журнала. Проще всего это реализовать в участках кода, изменяющих состояние ресурса.

По результатам работы программы (на основе файла журнала) построить график операций, производимых каждой задачей на общей временной оси для нескольких вариантов исходных данных. При построении графика следует ограничиться некоторым временным интервалом (примерно в 5-10 циклов работы каждого потока), на котором возникает интересующая нас ситуация (ожидание доступа к ресурсу). Примерный вид графика (для задачи о читателях-писателях) приведен на рис. 5.3.

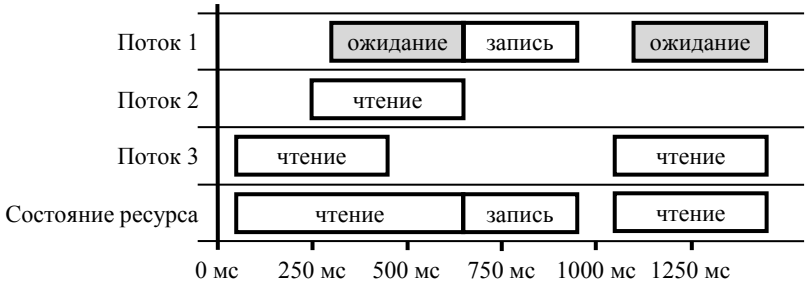


Рис. 5.3. Временной график операций, производимых потоками

Вариант 1 (нечетные номера по списку)

Используя программу решения задачи производитель-потребитель, произвести модификации и испытания в соответствии со следующими требованиями:

1. Реализовать запись обработанных элементов в выходной файл. Имя файла можно задать константой.

2. Реализовать возможность параллельной работы нескольких задач-потребителей с сохранением порядка обрабатываемых данных в выходном файле. Количество задач-потребителей рекомендуется выбирать равной числу доступных процессоров минус один, но не меньше трех.

3. Посчитать количество блоков данных, обработанных каждой задачей потребителем в отдельности. Это можно реализовать как в самой программе, так и путем анализа файла журнала. Время работы программы выбрать достаточно большим, чтобы данные были статистически достоверны (например, чтобы в среднем приходилось по 100 блоков данных на каждую задачу-потребителя). Сделать вывод о равномерности работы задач потребителей.

4. Подобрать диапазоны задержек для моделирования двух ситуаций – полного буфера (простой задачи-производителя) и пустого буфера (простой задач-потребителей). Убедиться в корректной работе программы в обоих случаях.

Вариант 2 (четные номера по списку)

Используя программу решения задачи о читателях-писателях, произвести модификации и испытания в соответствии со следующими требованиями:

1. Обеспечить справедливый доступ к ресурсу для любого количества задач писателей и читателей.

2. Реализовать подсчет количества обращений к ресурсу для каждой задачи и времени (максимального и среднего), которая каждая из задач провела в ожидании ресурса. Это можно реализовать как в самой программе, так и путем анализа файла журнала. Время работы программы выбрать достаточно большим, чтобы данные были статистически достоверны (более 100 обращений к ресурсу для каждой задачи).

3. Провести испытания работы программы с разным количеством задач (не менее четырех вариантов). Проанализировать результаты.

4. Определить загруженность ресурса (время в течении которого он был открыт на чтение и на запись). Это можно реализовать как в самой программе, так и путем анализа файла журнала.

5. Сравнить результаты работы измененной программы с исходной, которая использует блокировку чтения-записи с теми же временными задержками.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Индивидуальное задание.
4. Характеристики используемого оборудования и программных средств.
5. Описание произведенных модификаций в программе.
6. Результаты работы программы для нескольких случаев (включая временной график операций для каждого случая).
7. Анализ работы программы.
8. Выводы.
9. Приложение 1. Файлы журнала работы программ.
10. Приложение 2. Тексты программ.

Вопросы для контроля знаний

1. Опишите состояния, в которых может находиться поток в POSIX. Каковы условия перехода между этими состояниями?
2. Как происходит диспетчеризация потоков в POSIX? Опишите политики диспетчеризации.
3. Какое минимальное количество приоритетов должна иметь POSIX-совместимая операционная система?
4. Перечислите средства синхронизации потоков, предусмотренные стандартом POSIX.
5. Опишите особенности мьютексов в POSIX.
6. Как происходит завершение работы потока в POSIX?
7. Что такое инверсия приоритетов? Какие средства в POSIX предусмотрены для борьбы с ней?
8. Какие достоинства и недостатки имеет статическая инициализация примитивов синхронизации (мьютексов, условных переменных и т.д.)?
9. Для чего используется условная переменная?
10. Сформулируйте все три проблемы читателей-писателей. Что означает решение этой проблемы с преимуществом читателей / писателей?
11. Что такое блокировка чтения-записи? Гарантирует ли ее использование справедливое обслуживание?

Лабораторная работа 6

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ НА ГРАФАХ

Цели работы

Изучение параллельных версий алгоритмов на графах.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо:
– персональный компьютер с многоядерным процессором.

Вводная часть

Граф – это совокупность множества вершин и наборов связей между ними. Графы широко используются в различных науках: в математике, физике, географии, биологии и т.д.

При использовании графов многие сложные на первый взгляд процессы становятся наглядны и просты для понимания.

Существует несколько видов графов:
– неориентированный граф,
– ориентированный граф.

Ориентированный граф или орграф – это граф, у которого каждое ребро из всего множества ребер графа имеет направление. Графически ребра ориентированных графов показаны стрелками.

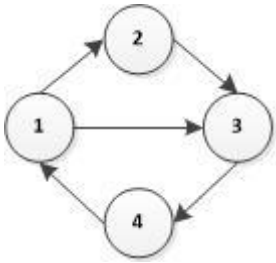


Рис. 6.1. Ориентированный граф

Неориентированный граф – это граф, у которого каждое ребро из всего множества ребер графа не имеет заданного

направления. Графически ребра неориентированных графов показываются в виде линий.

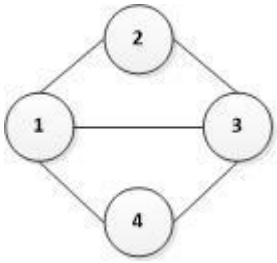


Рис. 6.2. Неориентированный граф

При представлении графов в компьютерных программах обычно используются либо матрицы смежности, либо списки смежности.

Пример: матрица доступа для неориентированного графа.

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Существует огромное количество последовательных алгоритмов для работы с графами, которые выполняют различные задачи (поиск кратчайшего пути, вычисление расстояния от одного узла до всех остальных, и т.д.). Многие из этих алгоритмов могут быть адаптированы к параллельному вычислению. Мы рассмотрим лишь наиболее простые и часто используемые алгоритмы.

Алгоритм Флойда-Уолшелла

Алгоритм Флойда–Уолшелла предназначен для нахождения кратчайших путей между всеми вершинами ориентированного графа. Этот алгоритм изобрели в 1962 г. Роберт Флойд и Стивен Уолшелл.

Входной информацией для алгоритма является любой взвешенный ориентированный граф, а также построенная по этому графу матрица смежности, в которой все неизвестные расстояния от одной вершины до другой обозначаются “бесконечностью”.

Основную идею данного алгоритма проще всего продемонстрировать в виде псевдокода:

```
for (k = 0; k < n; k++)
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         A[i,j] = min(A[i,j], A[i,k]+A[k,j]);
```

Здесь A – матрица смежности, I – начальный узел, J – конечный узел, n – количество узлов, k – промежуточный узел. Как видно из псевдокода, алгоритм Флойда-Уолшелла имеет сложность n3. После выполнения алгоритма матрица смежности A будет заменена матрицей кратчайших путей между всеми узлами графа.

Данный алгоритм можно легко преобразовать в параллельную версию.

Несмотря на то, что основные вычисления происходят при нахождении минимального из двух значений, распараллеливать по данному вычислению не имеет никакого смысла из-за простоты вычисления. В этом случае рекомендуется реализовать одновременное обновление нескольких значений матрицы A.

Алгоритм Дейкстра

Алгоритм Дейкстра предназначен для нахождения кратчайшего расстояния от одного из узлов графа до всех остальных. Этот алгоритм изобрел Эдсгер Вибе Дейкстра в 1959 г.

Данный алгоритм имеет большую популярность и решает огромное количество задач. Например – вычисление расстояния от столицы до всех городов области, или вычисление времени поездки от дома до всех библиотек города.

Входной информацией для алгоритма является любой граф, который не содержит ребра отрицательного веса (рис. 6.3). Рассмотрим алгоритм на примере задачи: найти кратчайшее расстояние от вершины 1 до всех остальных вершин графа.

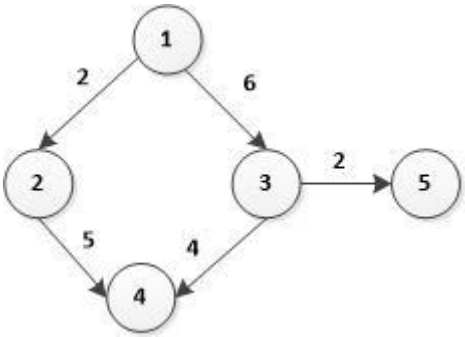


Рис. 6.3. Граф не содержащий ребра отрицательного веса

Суть этого алгоритма заключается в переборе всех узлов графа и назначении им меток, которые являются минимальными расстояниями от заданного узла до текущего.

1. Узлу 1 присвоим метку 0, всем остальным узлам присваиваем метки равные «бесконечности» (рис 6.4).

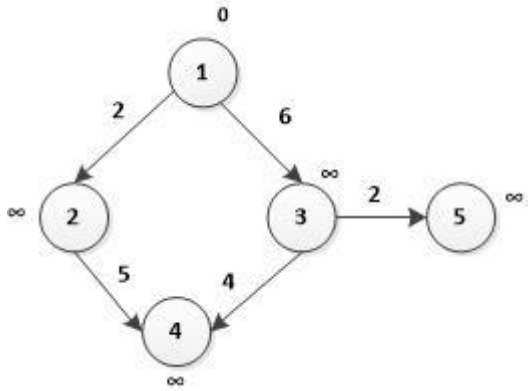


Рис. 6.4. Результат выполнения 1-го пункта

2. Рассматриваем все узлы, в которые из узла 1 есть путь. Каждому из рассмотренных узлов назначаем метки, равные длине пути из узла 1.

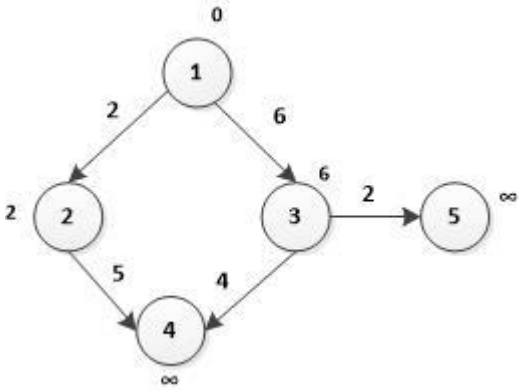


Рис. 6.5. Результат выполнения 2-го пункта

3. Выбираем узел S, который имеет минимальную метку и рассматриваем все узлы, в которые из узла S есть путь. Каждому из рассмотренных узлов присваиваем метки, равные сумме метки узла S и длины пути из узла S в рассматриваемый узел. В случае, если узлу уже присвоена метка, то необходимо сравнить значение существующей метки с суммой метки узла S и длины пути из узла S в рассматриваемый узел. Если значение суммы меньше значения метки рассматриваемого узла, необходимо перезаписать метку.

4. После прохождения всех узлов, в которые из узла S есть путь, узел S помечается как завершенный.

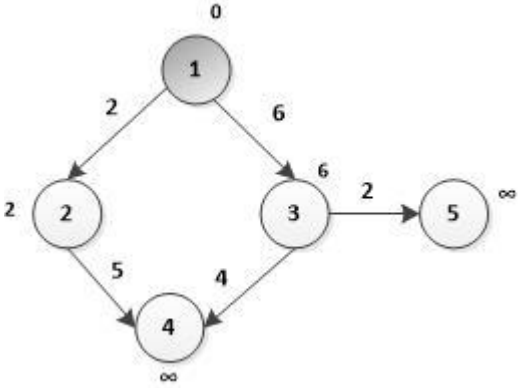


Рис. 6.6. Результат выполнения 4-го пункта

5. Выполняем пункты 2-4, пока есть незавершенные вершины.

После выполнения всех действий получаем результат:

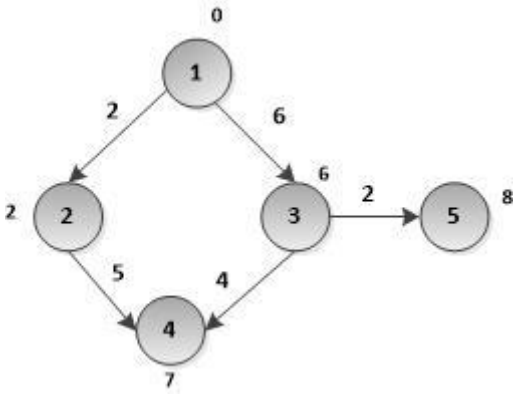


Рис. 6.7. Результат выполнения 5-го пункта

Алгоритм поиска в ширину (Bread-First Search)

Алгоритм поиска в ширину предназначен для поиска узла в графе, удовлетворяющего заданным условиям. Данный алгоритм является одним из самых простейших алгоритмов для работы с графами.

На вход данного алгоритма может подаваться любой граф.

Для более наглядного представления опишем алгоритм в виде блок-схемы (рис 6.8.):

Ввиду того, что алгоритм осуществляет только поиск узла, в процессе работы его можно изменять для сохранения и обработки дополнительной информации, необходимой для решения конкретной задачи.

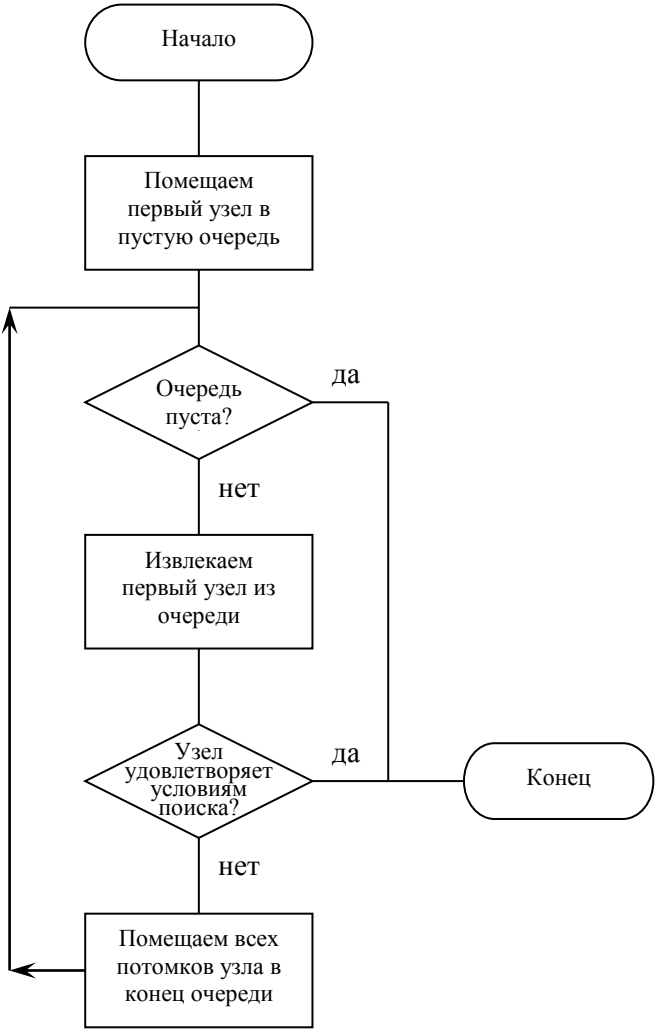


Рис. 6.8. Алгоритм поиска в ширину

Алгоритм Борувки

Алгоритм Борувки позволяет найти минимальное остовное дерево в графе. Данный алгоритм был изобретен Отакаром Борувкой в 1926 г. Иногда этот алгоритм называются алгоритмом Соллина.

На вход данного алгоритма подается любой граф.

Основная идея алгоритма заключается в последовательном объединении в группы узлов с минимальными весами связей между ними. Для наглядности рассмотрим следующий пример: необходимо найти минимальное остовное дерево для графа (рис. 6.9):

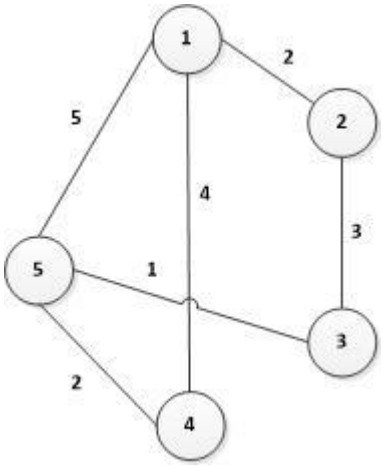


Рис. 6.9. Неориентированный граф

Распишем минимальные веса связей для каждого узла:

– для узла 1 минимальная связь равна 2 и связывает с узлом 2;

– для узла 2 минимальная связь равна 2 и связывает с узлом 1;

– для узла 3 минимальная связь равна 1 и связывает с узлом 5;

– для узла 4 минимальная связь равна 2 и связывает с узлом 5;

– для узла 5 минимальная связь равна 1 и связывает с узлом 3;

Как видно из списка, произойдут следующие объединения:

– узел 1 будет объединен с узлом 2;

– узел 5 будет объединен с узлом 3 и узлом 4.

В результате получаем следующую картину (рис. 6.10):

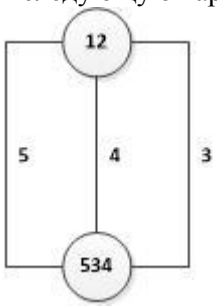


Рис. 6.10. Граф после объединения узлов

Как видно на рисунке, минимальный вес связи между этими узлами будет равен 3.

После этого необходимо раскрыть все объединенные узлы для получения минимального остовного дерева (рис. 6.11):

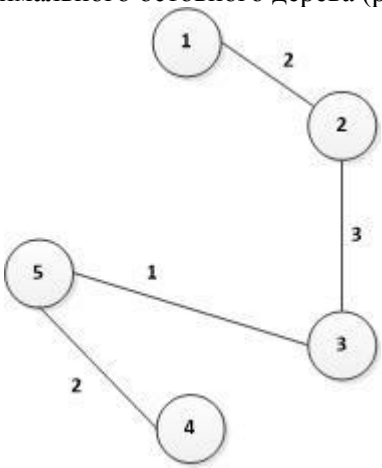


Рис. 6.11. Минимальное остовное дерево

Задание для лабораторной работы

Требуется реализовать параллельную версию алгоритмов рассмотренных в данной лабораторной работе. В качестве исходных данных можно использовать разнообразные данные, полученные из открытых источников, например:

– структуру социальной сети (или ее части);

– карту автомобильных/железных дорог;

– структуру перекрестных ссылок сайта;

– анализ текстов программы, распространяемой по свободной лицензии.

Для визуализации исходного графа и найденного решения можно воспользоваться Graphviz (Graph Visualization Software), официальный сайт – http://www.graphviz.org/.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Индивидуальное задание.
4. Описание параллельной версии алгоритма.
5. Описание исходных данных.
6. Результаты работы программы.
7. Анализ работы программы.
8. Выводы.
9. Приложение.

Вопросы для контроля знаний

1. Дайте определение графа.
2. Дайте обобщенное определение графа.
3. Какие виды графов существуют?
4. Опишите характеристики графов.
5. Опишите основные способы предсталвения графа.
6. Опишите алгоритм Флойда-Уолшелла.
7. Опишите алгоритм Дейкстра.
8. Опишите алгоритм поиска в ширину.
9. Опишите алгоритм Борувки.

Лабораторная работа 7
MESSAGE PASSING INTERFACE

Цели работы

Знакомство с особенностями параллельных вычислений в модели с распределенной памятью. Изучение интерфейса передачи сообщений MPI.

Получение практического опыта по разработке программ для выполнения на вычислительном кластере.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– сеть из восьми или более персональных компьютеров, объеденных в единый вычислительный кластер;

– установленная ОС Windows или Linux;

– средства компиляции и запуска программ с использованием MPI (рекомендуется MPICH или OpenMPI);

– компилятор для языка C/C++ под выбранную операционную систему.

Общие сведения

Рассмотренные в прежних главах технологии и алгоритмы были ориентированы на использование модели вычислений с разделяемой памятью. Это позволяло решать многие задачи более эффективно на системах с многоядерными процессорами и многопроцессорных системах. Однако количество ядер (процессоров) в таких системах обычно невелико и обычно не превышает 12. Существует довольно широкий круг задач, который требует значительно больших вычислительных ресурсов. На их решение на “обычных” персональных ЭВМ ушли бы годы или десятилетия. Поэтому в этой главе мы рассмотрим другой подход к организации параллельных вычислений – модель вычислений с распределённой памятью.

В модели с распределенной памятью основными элементами являются отдельные вычислительный узлы, объединенные с

помощью сети в единную вычислительную систему – кластер. В качестве отдельных узлов могут выступать обычные ЭВМ, а в качестве сети – Ethernet. Количество вычислительных элементов в такой системе может достигать нескольких десятков и тысяч узлов.

С точки зрения разработки программного обеспечения, основной отличительной особенностью кластерных вычислений является то, что коммуникация здесь обходится намного дороже, чем в системах с разделяемой памятью. Поэтому алгоритмы, требующие частой коммуникации между вычислительными узлами, плохо адаптируются к этой модели. Большинство используемых алгоритмов призваны свести коммуникацию между узлами к минимуму и учитывают физическую удаленность узлов друг от друга. Для этого все узлы в сети объединяются в сеть с учетом выбранной топологии. Соседние узлы в топологии сети, как правило, расположены рядом физически. На рис. 7.1 приведены наиболее популярные топологии для кластерных вычислений.

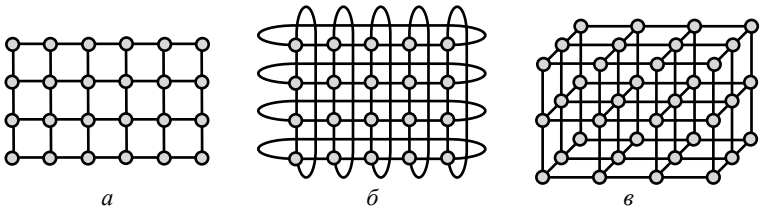


Рис. 7.1. Топологии сетей для кластерных вычислений: *а* – двухмерный массив; *б* – тор; *в* – куб

Также популярной топологией является гиперкуб (отсутствует на рисунке ввиду сложности отображения), который получается добавлением связей от первого к последнему узлу по всем трем направлениям, аналогично тому, как получается тор из двухмерного массива. Реже используются топологии “цилиндр” (отличается от тора отсутствием циклических связей по одному из направлений) и дерево.

Для организации взаимодействия отдельных узлов используется механизм передачи сообщений, наиболее популярным на сегодняшний день является стандарт MPI. Существует несколь-

ко версий стандарта MPI, последней на момент написания книги является MPI 3.0, выпущенная 21 сентября 2012 г. В стандарте прописана реализация для языков C/C++ и Fortran, но существует много вариантов привязки к языкам Java, Python, Ruby, R, Matlab и т.д. Среди реализаций с открытым исходным кодом наибольшей популярностью пользуются две реализации:

– MPICH (официальный сайт http://www.mpich.org/),

– OpenMPI (официальный сайт http://www.open-mpi.org/).

Обе реализации активно развиваются на момент написания книги и поддерживают ОС Windows, Unix, Linux и MacOS.

Стандарт MPI-1 описывает:

– коммуникацию между отдельными узлами (point-to-point communications);

– коллективные операции (collective operations);

– группы процессов (process groups);

– домены коммуникаций (communications domain);

– топологии процессов (process topologies);

– управление окружением (environmental management);

– интерфейс для профилирования (profiling interface);

– привязку для языков C и Fortran (Fortran and C bindings).

В дополнении к этому в стандарте MPI-2 имеются:

– динамическое управление процессами (dynamic process management);

– параллельный ввод-вывод (parallel input/output);

– односторонние операции (one-side operations);

– привязка для языка C++.

Пример 7.1. Простейшая программа на MPI

Данный пример демонстрирует минимальный набор из шести MPI-функций, необходимый и достаточный для создания полноценной MPI-программы, для решения реальной задачи.

К любой MPI-программе необходимо подключить заголовочный файл с определением MPI-функций.

Обязательными являются инициализация среды выполнения (вызов функции MPI_Init) и завершение работы среды выполнения (функция MPI_Finalize). Все остальные MPI-функции, за редким исключением, можно вызывать только между этими

двумя вызовами. Не рекомендуется вызывать какой-либо другой код (даже без использования MPI) до инициализации и после завершения работы (кроме return), так как поведение программы в этом случае по стандарту не определено.

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

int main( int argc, char *argv[])
{
   char message[20];
   int myrank, nprocs;
   MPI_Statusstatus;

   // инициализация MPI (создание процессов)
   MPI_Init(&argc, &argv);

   // идентификация текущего процесса
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   if (myrank == 0)
   { // выполняется нулевым процессом
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1,
         MPI_CHAR, 1, 99, MPI_COMM_WORLD);
   }
   else if (myrank == 1)
   { // выполняется первым процессом
      MPI_Recv(message, 20,
         MPI_CHAR, 0, 99, MPI_COMM_WORLD,
         &status);
      printf("received :%s:\n", message);
   }

   // завершение работы (ожидание процессов)
   MPI_Finalize();
   return 0;
}
```

Когда параллельная программа запускается на кластере, фактически происходит запуск нескольких копий программы (процессов). Все копии программы работают на основе одинакового исполняемого кода, такая организация параллельной программы называется SPMD (Single Program Multiply Data). Обычно на каждом вычислительном узле запускается один или более процессов. Несмотря на то, что, начиная с версии MPI 2.0, есть возможность динамического порождения процессов, пользуются ей редко. В большинстве случаев количество процессов определяется в момент запуска программы и не изменяется во время работы.

После прохождения инициализации мы можем определить место нашего процесса среди всех других процессов при помощи функции MPI_Comm_rank. Первым аргументом передается коммуникатор – некий объект, через который процессы группы могут обмениваться сообщениями. Предопределены два коммуникатора – MPI_COMM_WORLD и MPI_COMM_SELF. Первый коммуникатор объединяет все запущенные процессы, а во второй входит только текущий процесс. Метод MPI_Comm_size позволяет узнать сколько процессов связаны с данным коммуникатором.

Выполнение дальнейшего кода будет зависеть от номера (ранга) процесса:

– нулевой процесс отправит сообщение;

– первый процесс будет ожидать сообщение;

– остальные процессы ничего делать не будут.

Отправку и прием сообщений мы рассмотрим более подробно в следующей главе.

Для упрощения кода в примере мы убрали проверку на ошибки. Вызов любой MPI функции возвращает значение MPI_SUCCESS, если она завершилась успешно; или код ошибки, если произошла ошибка.

Также можно обратить внимание на указание типа MPI_CHAR в функциях приема и отправки данных. В стандарте MPI определены свои типы данных, что позволяет не привязываться к конкретному языку программирования при описании стандарта. Соответствия между типами данных для языка C приведены в табл. 7.1.

Таблица 7.1

Соответствие между типами данных языка С и стандарта MPI

Тип данных MPI	Тип данных С
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t

Коммуникация между отдельными узлами (“точка-точка”)

В коммуникациях “точка-точка” всегда участвуют два узла – отправитель и получатель данных. Прототипы функций отправки и приема данных выглядят следующим образом:

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

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

Первый параметр – это буфер источника и приемника соответственно. Второй параметр – это количество элементов для отправки и максимальное количество элементов для приема. Тип элемента указывается третьим параметром. Параметр dest используется для указания получателя сообщения, а параметр source указывает на отправителя (от кого мы ждем сообщение). В качестве отправителя можно указать MPI_ANY_SOURCE, в этом случае нам не важно, от кого мы получим сообщение. Следующие два параметра – tag и comm – позволяют задать тип сообщения и коммуникатор, они должны быть одинаковыми для соответствующих функций приема и отправки.

Существуют четыре различных режима для обмена сообщениями:

– стандартный (MPI_Send), функция завершит свою работу сразу после отправки сообщения; в этом случае нет гарантии, что функция приема сообщения была вызвана до окончания работы функции отправки сообщения;

– буферезированный (MPI_Bsend), функция завершит свою работу сразу после копирования сообщения в буфер;

– синхронный (MPI_Ssend), функции приема и отправки завершаться одновременно;

– с ожиданием готовности (MPI_Rsend), функция приема должна быть вызвана заранее, в противном случае функция отправки вернет ошибку.

Все приведенные выше функции коммуникации являются блокирующими, т.е. они приостановят дальнейшую работу программы до окончания отправки/приема сообщения. Существуют неблокирующие варианты этих функций (MPI_Irecv, MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend), которые принимают на

вход дополнительный параметр MPI_Request *request. Неблокирующие функции сразу возвращают управление, а параметр request в дальнейшем позволяет проверить завершение соответствующей операции при помощи функции MPI_Test или дождаться ее завершения при помощи функции MPI_Wait.

Коллективные операции

Операции коммуникации “точка-точка” являются достаточными для реализации сколь угодно сложных программ. Однако в большинстве случаев более эффективными окажутся операции коллективного взаимодействия, так как они могут учитывать специфику среды передачи данных (например, использовать широковещательные рассылки пакетов).

В MPI определено большое количество видов коллективного взаимодействия:

– барьер синхронизации (MPI_Barrier), позволяет синхронизировать выполнение процессов внутри группы;

– широковещательная рассылка копии данных среди всех процессов группы (рис. 7.2, *а*);

– распределение данных между процессами группы (рис. 7.2, *б*);

– сбор данных со всех процессов группы (рис. 7.2, *в*);

– сбор данных со всех процессов группы с сохранением результатов в каждом узле (рис. 7.2, *г*);

– редукция (рис. 7.2, *д*);

– редукция с сохранением результатов в каждом узле (рис. 7.2, *е*);

– редукция с последующим распределением результатов среди процессов группы (рис. 7.2, *ж*);

– вычисление частичной редукции (рис. 7.2, *з*);

– обмен данными между всеми процессами (рис. 7.2, *и*).

Большинство операций коллективного взаимодействия существует в двух вариантах: “простом”, когда размеры пересылаемых данных всем узлам одинаковы, и векторном, который предоставляет более широкие возможности по управлению размерами и расположением в памяти пересылаемых данных. Функции для векторного варианта имеют в названии букву “v”.

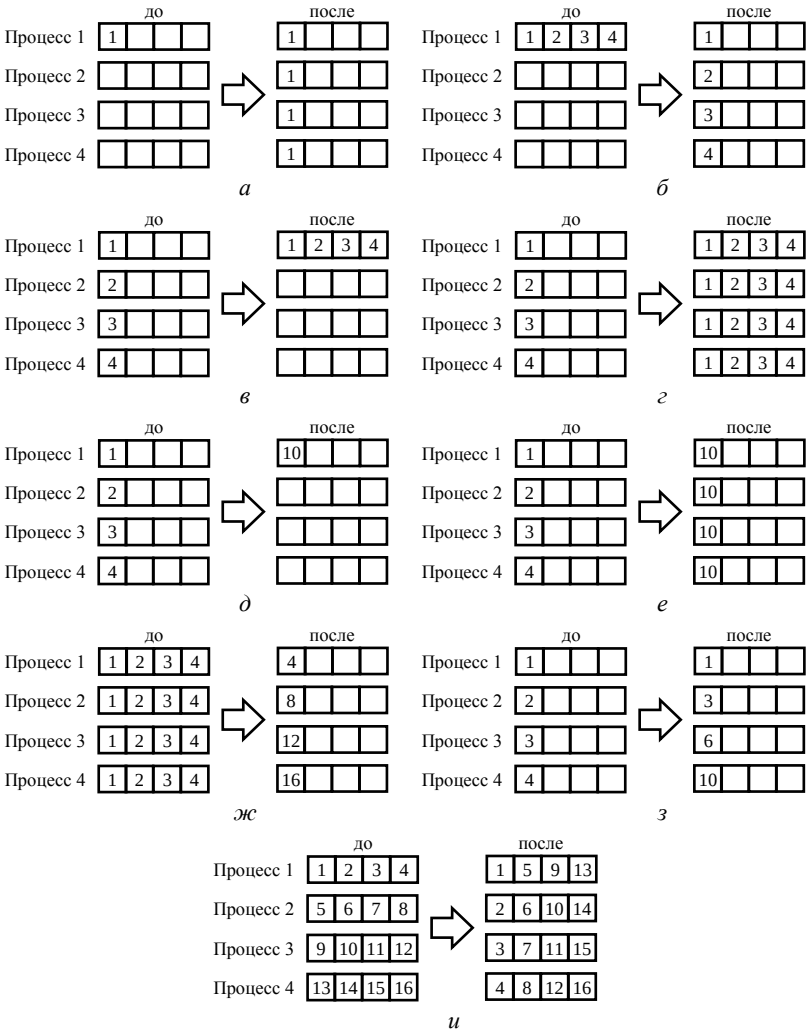


Рис. 7.2. Виды коллективного взаимодействия в MPI: *а* – MPI_Bcast; *б* – MPI_Scatter; *в* – MPI_Gather; *г* – MPI_Allgather; *д* – MPI_Reduce; *е* – MPI_Allreduce; *ж* – MPI_Reduce_scatter; *з* – MPI_Scan; *и* – MPI_Alltoall

В качестве операции для редукции можно использовать одну из предопределенных операций из табл. 7.2.

Таблица 7.2

Операции для редукции

Операция	Описание
MPI_MAX	максимальное значение
MPI_MIN	минимальное значение
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое И
MPI_BAND	побитовое И
MPI_LOR	логическое ИЛИ
MPI_BOR	побитовое ИЛИ
MPI_LXOR	логическое исключающее ИЛИ
MPI_BXOR	побитовое исключающее ИЛИ
MPI_MAXLOC	максимальное значение и позиция
MPI_MINLOC	минимальное значение и позиция

Группы процессов, коммуникаторы и виртуальные топологии

Группой называется пронумерованное множество процессов. Нумерация внутри группы начинается от 0 и заканчивается N-1, где N – число потоков в группе. Группа процессов всегда ассоциирована с коммуникатором – специальным объектом, через которого организуется обмен сообщений внутри группы. С точки зрения программиста, группа и коммуникатор являются одной сущностью.

Коммуникаторы позволяют использовать операции коллективного взаимодействия на выбранном подмножестве процессов. Группы и коммуникаторы являются динамическими объектами, которые могут создаваться и удаляться в ходе работы программы. При этом процесс может входить в несколько групп/коммуникаторов.

Коммуникаторы используются для построения виртуальных топологий. Слово “виртуальный” в данном контексте означает,

что логическая топология может отличаться от физической. MPI поддерживает два типа топологий: декартова топология и топология в виде графа.

С помощью декартовой топологии можно строить двухмерный массив, цилиндр, тор, куб и гиперкуб. Помимо размерности, эти топологии отличаются лишь отсутствием/наличием дополнительных связей для замыкания процессов в кольцо по соответствующим измерениям.

Применение виртуальных топологий решает две задачи: повышает удобство использования и эффективность коммуникации. Существует широкий набор алгоритмов, адаптированных к выполнению на определенной топологии.

На рис. 7.3 показано, как соотносится порядковый номер процесса с индексами в двухмерном массиве.

0 (0,0)	**1** (0,1)	**2** (0,2)	**3** (0,3)
4 (1,0)	**5** (1,1)	**6** (1,2)	**7** (1,3)
8 (2,0)	**9** (2,1)	**10** (2,2)	**11** (2,3)

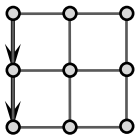
Рис. 7.3. Соотношение номера процесса с индексом в двухмерном массиве 3×4: наверху – порядковый номер процесса в глобальном коммуникаторе (MPI_COMM_WORLD); внизу – индекс процесса в топологии двухмерный массив

В следующих главах мы рассмотрим, как организуются вычисления на топологиях двухмерный массив и куб. В качестве примера задачи мы будем рассматривать задачу умножения матриц.

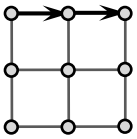
Умножение матриц на топологии двухмерный массив

В данном примере мы будем использовать двухмерный массив 3×3. Для упрощения предполагается, что размерность исходных матриц по каждому измерению кратна числу 3.

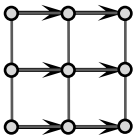
Весь процесс вычисления состоит из 6 шагов, как изображено на рис. 7.4.



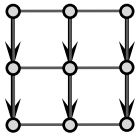
а – распределение матрицы А по строкам



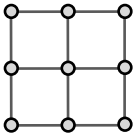
б – распределение матрицы В по строкам



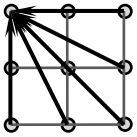
в – широковещательная рассыка строк матрицы В



г – широковещательная рассыка строк матрицы В



д – вычисление частичных произведений



е – сбор результатов частичных произведений

Рис. 7.4. Последовательность операций при умножении матриц на двухмерном массиве

На первом шаге исходная матрица А делится по строкам на равные части между всеми узлами в первой колонке. Сделать это можно при помощи вызова функции MPI_Scatter. На втором шаге такая же операция производится с исходной матрицей В, но разделение матрицы происходит по столбцам между всеми узлами в первом ряду. На третьем и четвертом шаге блоки элементов исходных матриц широковещательно рассылаются по строкам и столбцам вычислительного массива. В результате этого каждый процесс будет содержать часть строк матрицы А и часть столбцов матрицы В. Широковещательная рассылка может осуществляться при помощи функции MPI_Bcast. На пятом шаге непосредственно выполняется умножение элементов матриц между собой. В результате этого шага в каждом узле получится прямоугольный блок элементов результирующей матрицы. При этом каждый процесс будет вычислять ровно девятую часть элементов итоговой матрицы. Все, что остается сделать – собрать матрицу целиком в нулевом узле. Для этого пригодится функция MPI_Gather.

За исключением последней операции все остальные взаимодействия осуществлялись только между соседними узлами

сети, а значит, накладные расходы на коммуникацию были минимальны.

Наследуемые типы данных

Часто возникает потребность пересылать разнотипные данные (структуры с полями разных типов) или данные которые не располагаются в памяти последовательно (элементы матрицы, расположенные выше главной диагонали). В MPI для этого имеются два механизма:

– использование наследуемых типов данных (derived data types),

– упаковка данных в непрерывный буфер перед отправкой с последующей распаковкой после приема (pack/unpack data).

В большинстве случаев достичь желаемого результата можно использованием любого из предложенных механизмов. Объем пересылаемых данных в обоих случаях будет примерно одинаковым. Чаще всего выбор механизма обусловлен удобством использования, хотя имеются отличия в объеме используемой памяти и быстродействии. Мы рассмотрим подробно только первый.

Тип данных всегда указывается явно в любой функции взаимодействия. Помимо примитивных типов данных, перечисленных в табл. 7.1 можно указать наследуемый тип. Это справедливо как для коллективных операций взаимодействия, так и для операций “точка-точка”.

Наследуемый тип данных – это тип, построенный из стандартных типов. Наследуемый тип данных можно трактовать как объект, который определяет последовательность из примитивных типов данных и смещение (в байтах) до каждого из элементов последовательности относительного некоторого базового адреса. Наследуемый тип данных может также состоять из определенных ранее наследуемых типов.

Для каждого наследуемого типа можно определить его размер в байтах при помощи функции MPI_Type_size, фактически – это сумма размеров всех элементов, определенных в наследуемом типе. Помимо размера можно определить протяженность элемента в байтах, которая может быть больше из-за

выравнивания отдельных элементов в памяти. Протяженность элемента измеряется как разница между самым старшим байтом (upper bound) и самым младшим байтом (lower bound), занимаемым в памяти элементами последовательности. Протяженность элемента также может быть больше в случае, если элементы не лежат в памяти последовательно. Определить протяженность типа данных можно с помощью функции MPI_Type_extent.

Перед использованием наследуемого типа данных в функциях коммуникации его надо зафиксировать при помощи MPI_Type_commit. Если тип не участвует в коммуникации непосредственно (используется только для конструирования более сложного типа), то этого не требуется.

Существует несколько вариантов наследуемых типов данных.

Непрерывный тип (Contiguous)

Это наиболее простой тип данных, который позволяет определить непрерывную в памяти последовательность из заданного числа элементов однотипных элементов.

Для конструирования данного типа данных используется функция:

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

Первый аргумент count определяет количество элементов, второй аргумент oldtype определяет тип исходных элементов, а третий используется для определения нового типа данных. Пример использования данного типа с параметром count = 5 приведен на рис. 7.5.

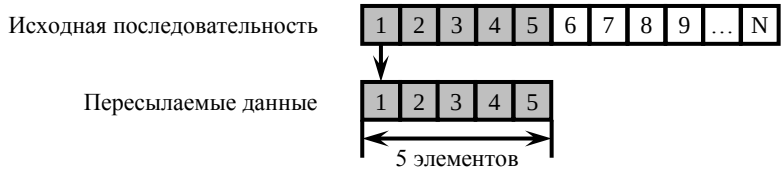


Рис. 7.5. Пересылка непрерывного типа данных длиной 5 элементов

Данный тип можно использовать, например, для выделения строки матрицы, при условии, что сама матрица лежит в непрерывном блоке памяти.

Тип “вектор” (Vector, Hvector)

Вектор позволяет определить последовательность из блоков однотипных элементов, расположенных в памяти на равном расстоянии друг от друга. Конструирование типа осуществляется при помощи вызова одной из функций:

```
int MPI_Type_vector(int count, int blocklength,
   int stride, MPI Datatype oldtype,
   MPI Datatype newtype)

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

Первый аргумент count определяет количество блоков, второй аргумент blocklength – количество элементов в блоке. Третий аргумент stride задает задает расстояние между блоками (может быть отрицательным), в первой функции расстояние измеряется в элементах, а во второй – в байтах. Второй вариант применим в случаях, когда в памяти последовательно хранятся данные разных типов (например, массив структур с полями типов int и double).

На рис. 7.6 показано использование типа вектор с параметрами count = 2, blocklength = 3, stride = 6.

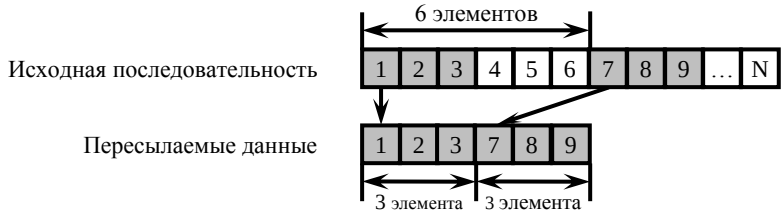


Рис. 7.6. Пересылка типа данных вектор из двух блоков длиной по 3 элемента каждый, с расстоянием в 6 элементов между блоками

С помощью этого типа данных можно выделить столбец матрицы, нечетные элементы последовательности или отдельный цветовой канал изображения, если оно хранится в памяти в формате RGBRGBRGB. Алгоритм сортировки Шелла, рассмотренный ранее, также хорошо сочетается с типом данных “вектор”.

Индексируемый тип данных (Indexed, Hindexed)

Отличие от предыдущего типа данных состоит в том, что теперь как сами блоки, так и расстояния между блоками могут быть различной длины. Точно так же, как и в предыдущем случае, имеются две функции:

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

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

Отличие аргументов от вектора состоит в том, что длины блоков и смещение теперь указываются в виде массивов размерностью count элементов.

Рассмотрим пример использования этого типа данных для формирования последовательности из элементов треугольной матрицы. Исходная матрица и пересылаемые элементы приведены на рис. 7.7.

Количество элементов будет равно количеству строк исходной матрицы n, а длины блоков будут уменьшаться с *n* до 1 с шагом 1. Остается только правильно рассчитать смещения до начала каждого блока. Если элементы матрицы хранятся в непрерывном блоке памяти построчно (за последним элементом строки j располагается первый элемент строки j+1), то смещение до начала строки с номером i равно номеру этой строки, умноженному на количество элементов в строке n. Также можно заметить, что блок в каждой строке смещен от ее начала на количество элементов, равное порядковому номеру строки.

исходная матрица

$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	…	$A_{1,n-1}$	$A_{1,n}$
$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	…	$A_{2,n-1}$	$A_{2,n}$
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	…	$A_{3,n-1}$	$A_{3,n}$
…	…	…	…	…	…
$A_{n-1,1}$	$A_{n-1,2}$	$A_{n-1,3}$	…	$A_{n-1,n-1}$	$A_{n-1,n}$
$A_{n,1}$	$A_{n,2}$	$A_{n,3}$	…	$A_{n,n-1}$	$A_{n,n}$

пересылаемые данные

$A_{1,1}$	…	$A_{1,n}$	$A_{2,2}$	…	$A_{2,n}$	…	$A_{n,n}$

Рис. 7.7. Пересылка треугольной матрицы при помощи типа данных Indexed (серым цветом отмечены пересылаемые элементы)

Ниже приведен фрагмент кода, соответствующий приведенной матрице из n×n элементов вещественного типа.

```
int len[n], disp[n];
for (size_t i = 0; i < n; ++i)
{
   len[i] = n -i;
   block_disp[i] = n*i + i;
}
MPI_Datatype newtype;
MPI_Type_indexed(n, len, disp, MPI_DOUBLE,
&new_type);
```

Тип структура (Struct)

Структура является наиболее общим типом наследуемых данных и может заменить любой из упомянутых выше типов. Для конструирования этого типа данных используется функция:

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

Первый аргумент count определяет количество блоков структуры и размерность массивов трех следующих аргументов. Аргумент array_of_blocklengths определяет сколько элементов входят в блок структуры, а аргумент array_of_types определяет типы этих элементов. Поскольку в структуры входят элементы разных типов, то смещение задается в байтах при помощи аргумента array_of_displacements.

В структуре могут использоваться два псевдо-типа MPI_LB, MPI_UB. Эти типы имеют нулевую длину и служат только для определения смещения до следующего элемента при пересылке нескольких однотипных элементов.

Рассмотрим пример. Пусть мы хотим разбить исходную матрицу на прямоугольные блоки (рис. 7.8). Данный тип разбиения нам пригодится при умножении двух матриц на топологии куб в следующей главе.

Самым простым вариантом на первый взгляд является использование наследуемого типа вектор:

```
// block_rows - кол-во рядов в блоке
// block_cols - кол-во колонок в блоке
// n_cols - кол-во колонок в матрице
MPI_Datatype vec_type;
MPI_Type_vector(block_rows, block_cols, n_cols,
   MPI_DOUBLE, &vec_type);
```

Однако при попытке переслать более одного объекта мы столкнемся с ошибкой (рис. 7.8, *а*): адрес следующего блока будет определен неправильно. Система считает, что следующий блок начинается в памяти со следующего байта после последнего байта текущего блока, но в нашем случае это не соответствует ожидаемому результату.

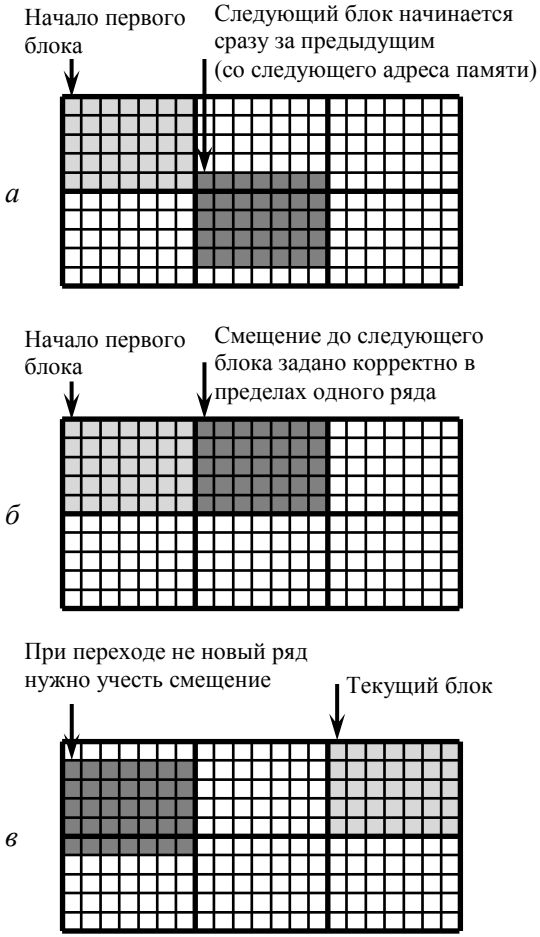


Рис. 7.8. Разбиение матрицы на прямоугольные блоки: *а* – смещение при использовании типа vector; *б* – при использовании дополнительной структуры; *в* – смещение при переходе на новый ряд матрицы

Выходом из сложившейся ситуации будет использование структуры, состоящей из двух полей – типа вектор, объявленного выше, и псевдо-типа MPI_UB, который лишь укажет смещение до следующего блока. Начало блоков в одном ряду смещены относительно друг друга на block_cols элементов. Нам лишь

необходимо перевести смещение в байты с учетом выравнивания типов. Сделать это можно при помощи функции MPI_Type_extent.

Оформим код в виде небольшой функции для объявления наследуемого типа “блок матрицы”.

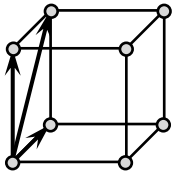
```
void CreateSubmatrixType(MPI_Datatype *type,
   size_t n_rows, size_t n_cols,
   size_t block_rows, size_t block_cols)
{
   // объем памяти занимаемый MPI_DOUBLE
   int double_extent;
   MPI_Type_extent(MPI_DOUBLE, &double_extent);

   // задаем типы элементов структуры
   MPI_Datatypetypes[2];
   MPI_Type_vector(block_rows, block_cols,
      n_cols, MPI_DOUBLE, &types[0]);
   types[1] = MPI_UB;

   // задаем длину элементов структуры
   Int blocklengths[2] = {1, 1};

   // задаем смещение до элементов структуры
   int displacements[2] =
      {0, double_extent*block_cols};

   // объявляем тип-структуру
   MPI_Type_struct(2, blocklengths,
      displacements, types, type);

   // фиксируем тип
   MPI_Type_commit(type);
}
```

Теперь смещение до следующего элемента определяется корректно в пределах одного ряда (рис. 7.8, *б*). При переходе на следующий ряд блоков (рис. 7.8, *в*), блок снова оказывается не на том месте, где ожидалось. К счастью, это можно исправить непосредственно в функции коммуникации.

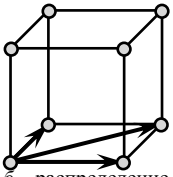
Умножение матриц на топологии куб

Схема умножения матриц с использованием топологии куб (или трехмерный массив) во многом сходна с умножением на двухмерном массиве. Последовательность шагов представлена на рис. 7.9 для размерности 2×2×2. Здесь имеется один дополнительный шаг – редукция. Для выполнения редукции можно воспользоваться функцией MPI_Reduce.

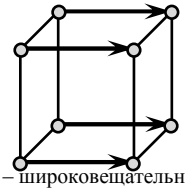
Имеется небольшое отличие в том, как происходит разбиение исходных матриц на блоки элементов: матрица делится на равные части по каждому измерению, в результате чего получаются блоки одинакового размера, которые затем распределяются между процессорами вдоль плоскости. Предполагается, что размерность исходных матриц кратна числу процессоров по используемому измерению (рис. 7.10).



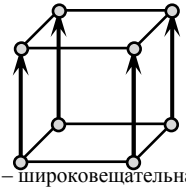
а – распределение элементов матрицы A



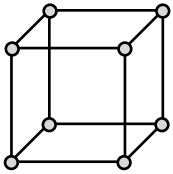
б – распределение элементов матрицы B



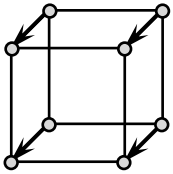
в – широковещательная рассылка элементов матрицы A



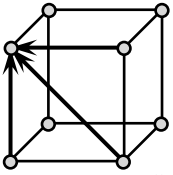
г – широковещательная рассылка элементов матрицы B



д – вычисление частичных произведений



е – редукция



ж – сбор результирующей матрицы

Рис. 7.9. Последовательность операций при умножении матриц на трехмерном массиве 2×2×2

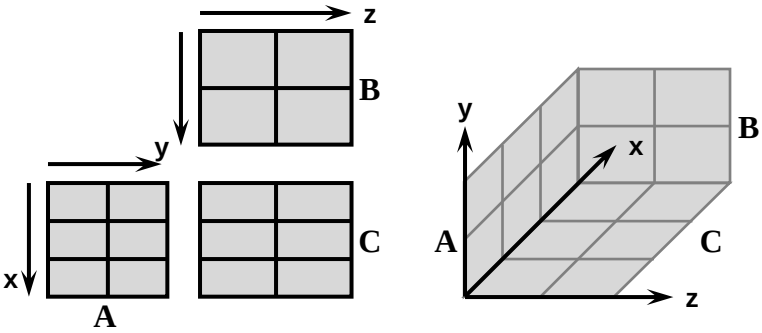


Рис. 7.10. Схема разбиения исходных матриц для распределения по вычислительному массиву 3×2×2: слева – представление на плоскости, справа – в пространстве

В остальном алгоритм похож на предыдущий. Попробуем реализовать его в виде программы.

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

const size_t GRID_DIM = 3;
const int DO_NOT_REORDER = 0;

int main( int argc, char *argv[])
{
   // инициализация MPI (создание процессов)
   MPI_Init(&argc, &argv);

   // идентификация текущего процесса
   int myrank, nprocs;
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   // создаем куб из имеющихся узлов
   int dims[GRID_DIM] = {0, 0, 0};
   int periods[GRID_DIM] = {0, 0, 0};
   MPI_Comm comm3D;
   MPI_Dims_create(nprocs, GRID_DIM, dims);
   MPI_Cart_create(MPI_COMM_WORLD, GRID_DIM,
      dims, periods, DO_NOT_REORDER, &comm3D);
```

Начало программы весьма типичное для любой MPI-программы и состоит в инициализации и идентификации процесса.

Вспомогательная функция MPI_Dims_create помогает автоматически рассчитать число процессов по каждому измерению, основываясь на количестве запущенных процессов. Количество измерений (3) указывается при помощи константы GRID_DIM. Для восьми процессов функция вернет распределение 2×2×2, а для пяти процессов – 5×1×1. Хотя программа будет корректно работать в любом случае, нам интересно, чтобы по каждому измерению было более одного процесса.

Функция MPI_Cart_create создает вычислительный массив заданной размерности из процессов глобального коммуникатора и выделяет под него отдельный коммуникатор comm3D. Параметр periods позволяет задать, будут связи по соответствующим измерениям зацикливаться или нет. В нашем примере этого не требуется.

Код определения размеров исходных матриц, выделения под них памяти и заполнения мы здесь приводить не будем, но отметим, что он должен вызываться только из нулевого процесса. После этого необходимо разослать размеры матриц всем участвующим в вычислении процессам:

```
double *A, *B, *C;
int matrix_dim[3];
if (myrank == 0)
{
   // запрос размеров матриц matrix_dim
   // выделение памяти под A, B, C
   // заполнение матриц A, B исходными данными
   ...
}

// рассылаем всем процессам размеры матриц
MPI_Bcast(matrix_dim, 3, MPI_INT, 0,
   MPI_COMM_WORLD);
```

Теперь обратим внимание на то, как у нас происходит коммуникация на разных шагах алгоритма. На шаге 1 (рис. 7.9, *а*) в коммуникации участвуют процессы в плоскости XY с нулевой *z* координатой, а на шаге 2 (рис. 7.9, *б*) – процессы в плоскости

XZ с нулевой *y* координатой. На шагах 3 и 4 (рис. 7.9, *в-г*) взаимодействуют все процессы по оси Z и все процессы по оси Y соответственно. Шаг, на котором вычисляется произведение матриц, не требует коммуникации, редукция (рис. 7.9, *е*) выполняется по оси X. И, наконец, на последнем шаге взаимодействуют все процессы в плоскости YZ с нулевой координатой *x*. Итого нам требуется 6 дополнительных коммуникаторов – три для плоскостей и три для осей.

Для декартовой топологии их легко создать путем понижения размерности пространства по соответствующей оси (осям) при помощи MPI_Cart_sub:

```
// коммуникаторы для плоскостей XY, XZ, YZ
MPI_Comm commXY, commXZ, commYZ;
int XY_dimensions[GRID_DIM] = {1, 1, 0};
int XZ_dimensions[GRID_DIM] = {1, 0, 1};
int YZ_dimensions[GRID_DIM] = {0, 1, 1};
MPI_Cart_sub(comm3D, XY_dimensions, &commXY);
MPI_Cart_sub(comm3D, XZ_dimensions, &commXZ);
MPI_Cart_sub(comm3D, YZ_dimensions, &commYZ);

// коммуникаторы для осей X, Y, Z
MPI_Comm commX, commY, commZ;
int X_dimension[GRID_DIM] = {1, 0, 0};
int Y_dimension[GRID_DIM] = {0, 1, 0};
int Z_dimension[GRID_DIM] = {0, 0, 1};
MPI_Cart_sub(comm3D, X_dimension, &commX);
MPI_Cart_sub(comm3D, Y_dimension, &commY);
MPI_Cart_sub(comm3D, Z_dimension, &commZ);
```

Несмотря на то что мы создаем всего 6 коммуникаторов, следует отметить, что для разных узлов это будут разные коммуникаторы. Например, узлы с координатами (1, 1, 1) и (1, 2, 1) будут принадлежать к различным коммуникаторам commX, так как они отличаются координатами *y*.

На рис. 7.11 показан результат работы функции для понижения размерности исключением координаты *x* для массива 4×3×3. Исключение координаты приводит к появлению четырех двухмерных коммуникаторов 3×3 узла в каждом. Также отме-

тим, что через эти коммуникаторы могут общаться только узлы с одинаковой координатой *x*.

В нашем примере для всех двухмерных коммуникаторов необходимо проверять дополнительное условие на равенство исключаемой координаты нулю, так как взаимодействие внутри других коммуникаторов не требуется для решения задачи.

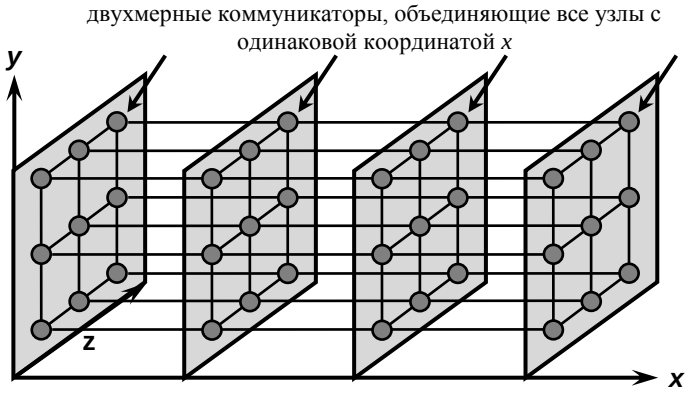


Рис. 7.11. Результат понижения размерности исключением координаты *x* для массива 4×3×3

Для того чтобы составить дополнительное условие для выбора конкретного коммуникатора, нам необходимо знать координаты узла в трехмерном массиве. Их можно получить следующим образом:

```
// получаем координаты процесса
int myrank3D;
int coords[GRID_DIM];
MPI_Comm_rank(comm3D, &myrank3D);
MPI_Cart_coords(comm3D, myrank3D, GRID_DIM,
   coords);

// вывод информации на экран
printf("Global rank: %2d, 3D rank : %2d, \
   coords = (%d, %d, %d)\n",
   myrank, myrank3D,
   coords[0], coords[1], coords[2]);
```

Вот что будет выведено на экран в результате работы этого кода для восьми процессов (сетка 2×2×2):

```
grid 2x2x2 created
Global rank:  5, 3D rank :  5, coords = (1, 0, 1)
Global rank:  1, 3D rank :  1, coords = (0, 0, 1)
Global rank:  3, 3D rank :  3, coords = (0, 1, 1)
Global rank:  7, 3D rank :  7, coords = (1, 1, 1)
Global rank:  4, 3D rank :  4, coords = (1, 0, 0)
Global rank:  2, 3D rank :  2, coords = (0, 1, 0)
Global rank:  6, 3D rank :  6, coords = (1, 1, 0)
Global rank:  0, 3D rank :  0, coords = (0, 0, 0)
```

Следующим шагом в нашей программе является выделение памяти под блоки элементов исходных матриц, это необходимо выполнить в каждом узле:

```
// определяем размеры блоков
size_t AA_rows = matrix_dim[0] / dims[0],
       AA_cols = matrix_dim[1] / dims[1],
       BB_rows = matrix_dim[1] / dims[1],
       BB_cols = matrix_dim[2] / dims[2],
       CC_rows = matrix_dim[0] / dims[0],
       CC_cols = matrix_dim[2] / dims[2];

// Выделяем память в каждом узле
double *AA, *BB, *CC, *CC_R;
AA = new double[AA_rows*AA_cols];
BB = new double[BB_rows*BB_cols];
CC = new double[CC_rows*CC_cols];
CC_R = new double[CC_rows*CC_cols];
```

Размер блока определяется как количество элементов по данному измерению в исходной матрице деленное на количество процессов в этом измерении. Дополнительная матрица CC_R нужна для редукции, так как нельзя указать одну и ту же область памяти при выполнении редукции в качестве источника и приемника одновременно.

Для рассылки исходных матриц мы воспользуемся функцией коллективного взаимодействия MPI_Scatterv. Это вариант функции, который позволяет указать длину и смещение данных в буфере для каждого блока. Мы используем смещение, чтобы

скорректировать позиции блока внутри матрицы при переходе на новую строку (рис. 7.8, *в*). Ниже приведен фрагмент кода регистрации типа и вычисления параметров функции MPI_Scatterv для матрицы A:

```
// создаем тип данных для блока матрицы A
MPI_Datatype typeA;
CreateSubmatrixType(&typeA, A_rows, A_cols,
   AA_rows, AA_cols);

// подготавливаем данные для рассылки матрицы A
int *sendcountsA = new int[dims[0]*dims[1]];
int *displsA = new int[dims[0]*dims[1]];
for (int i = 0; i < dims[0]; ++i)
{
   for (int j = 0; j < dims[1]; ++j)
   {
      sendcountsA[i*dims[1]+j] = 1;
      displsA[i*dims[1]+j] = i*dims[1]*AA_rows+j;
   }
}
```

Функция CreateSubmatrixType была описана нами выше (в предыдущей главе).

Для матриц B и C производятся аналогичные действия, при этом для матрицы B используются измерения dims[1] и dims[2], а для матрицы C – dims[0] и dims[2].

Теперь, когда все подготовления завершены, можно приступить непосредственно к выполнению алгоритма по шагам (рис. 7.9). После выполнения всех шагов результирующая матрица будет находиться в узле с индексом 0.

На первом шаге в коллективном взаимодействии должны участвовать только узлы с нулевой координатой *z*, на втором – узлы с нулевой координатой *x*, а на заключительном шаге – узлы с нулевой координатой *y*. Для того, чтобы учесть это в программе, достаточно добавить проверку на равенство нулю соответсвующей координаты непосредственно перед операцией коллективного взаимодействия.

```
// 1. Рассылка блоков матрицы A в плоскости X0Y
if (coords[2] == 0)
   MPI_Scatterv(A, sendcountsA, displsA, typeA,
     AA, AA_rows*AA_cols, MPI_DOUBLE, 0, commXY);

// 2. Рассылка блоков матрицы B в плоскости Y0Z
if (coords[0] == 0)
   MPI_Scatterv(B, sendcountsB, displsB, typeB,
     BB, BB_rows*BB_cols, MPI_DOUBLE, 0, commYZ);

// 3. Широковещательная рассылка блоков матрицы A
// вдоль оси Z
MPI_Bcast(AA, AA_rows*AA_cols, MPI_DOUBLE, 0,
   commZ);

// 4. Широковещательная рассылка блоков матрицы B
// вдоль оси X
MPI_Bcast(BB, BB_rows*BB_cols, MPI_DOUBLE, 0,
   commX);

// 5. Вычисление произведения блока матрицы A
// на блок матрицы B
for (size_t i = 0; i < CC_rows; ++i) {
   for (size_t j = 0; j < CC_cols; ++j) {
      CC[i*CC_cols+j] = 0.0;
      for (size_t k = 0; k < AA_cols; ++k)
         CC[i*CC_cols+j] +=
            AA[i*AA_rows+k] * BB[k*BB_rows + j];
   }
}

// 6. Редукция результатов перемножения
// вдоль оси Y
MPI_Reduce(CC, CC_R, CC_rows*CC_cols,
   MPI_DOUBLE, MPI_SUM, 0, commY);

// 7. Сборка результирующей матрицы в 0-м узле
// по плоскости X0Z
if (coords[1] == 0)
   MPI_Gatherv(CC_R, CC_rows*CC_cols, MPI_DOUBLE,
      C, sendcountsC, displsC, typeC, 0, commXZ);
```

Перед завершением работы программы остается освободить память из-под матриц и коммуникаторов. Следует обратить внимание на то, что память из-под матриц A, B, C следует высвобождать только в нулевом узле, так как в других процессах память под них не выделялась:

```
delete [] AA;
delete [] BB;
delete [] CC;
delete [] CC_R;

if (myrank == 0)
{
   delete [] A;
   delete [] B;
   delete [] C;
}

// освобождение ресурсов
MPI_Comm_free(&comm3D);
MPI_Comm_free(&commXY);
MPI_Comm_free(&commXZ);
MPI_Comm_free(&commYZ);
MPI_Comm_free(&commX);
MPI_Comm_free(&commY);
MPI_Comm_free(&commZ);

// завершение работы
MPI_Finalize();
return 0;
```

Измерение времени операций

В системе с распределённой памятью каждый процесс (узел) отсчитывает свое собственное время. И даже наличие специального метода MPI_Wtime само по себе не является решением проблемы. Согласно стандарту, переменная окружения MPI_WTIME_IS_GLOBAL будет иметь значение 1, если таймеры в различных узлах синхронизированы. В противном случае синхронизация таймеров перекладывается на разработчика прикладной программы.

Запуск программы на вычислительном кластере

Запуск программы на вычислительном кластере осуществляется при помощи команды mpirun или mpiexec:

```
mpirun --hostfile my_hosts -np 4 my_application
```

Ключевой параметр **--hostfile** позволяет указать имя файла, где перечислены ЭВМ, входящие в состав кластера (на которых будет выполняться программа). Параметр **--np** позволяет указать количество запускаемых процессов, в примере мы запускаем четыре процесса my_application.

Исполняемое приложение (исполняемый файл и все необходимые для запуска ресурсы) должно располагаться на всех узлах сети (желательно по одному и тому же пути), либо в общей сетевой директории.

Ниже приведен пример файла my_hosts с указанием списка ЭВМ для запуска приложения, следует обратить внимание на параметр slots, который позволяет указать количество процесссов, запускаемых на данной ЭВМ. Обычно оно равно количеству ядер (процессоров), доступных на данной ЭВМ. Это позволяет эффективно использовать все имеющиеся в наличии вычислительные ресурсы.

```
node0
node1 slots=2
node2 slots=4
foo.example.com slots=4
```

В качестве имени ЭВМ используется сетевое имя ЭВМ или IP-адрес.

В сборке MPICH под Windows можно воспользоваться вспомогательной утилитой с графическим интерфейсом для запуска приложений на кластере, ее внешний вид представлен на рис. 7.12. Данная утилита позволяет указать запускаемое приложение, задать количество процессов и список ЭВМ, на которых эти процессы будут запущены. Дополнительно можно настроить переменные окружения и сохранить журнал работы программы, где будут отражены все операции взаимодействия между процессами для последующего анализа.

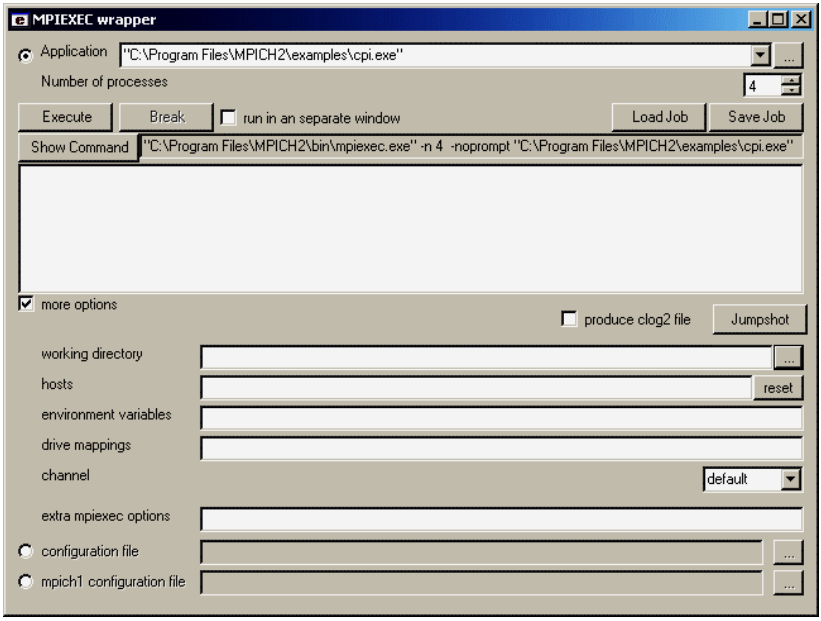


Рис. 7.12. Основное окно утилиты wmpiexec из сборки MPICH

Задание для лабораторной работы

В качестве задания предлагается реализовать решение задачи коммивояжера методом ветвей и границ для выполнения на вычислительном кластере.

Задача коммивояжера – это классическая задача комбинаторной оптимизации, заключающаяся в поиске оптимального маршрута, проходящего через все указанные города с возвратом в исходный город. В качестве исходных данных используется матрица проезда из одного города в другой. Задача называется симметричной, если стоимость проезда из города A в город B равна стоимости проезда из B в A для всех пар городов A и B. В противном случае она называется несимметричной.

Исходные данные (матрицу стоимости проезда) для работы можно взять с одного из следующих сайтов:

http://people.sc.fsu.edu/~jburkardt/f_src/tsp_io/tsp_io.html,
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/,

http://www.math.uwaterloo.ca/tsp/data/index.html.

Также можно построить матрицу самостоятельно, используя в качестве метрики расстояние в км между крупнейшими городами России (столицами европейских государств и т.п.).

Требуется исследовать время работы программы с входными данными различной размерности, для чего можно использовать первые N городов из списка. Построить график зависимости времени работы программы от размера задачи (количества городов). Провести не менее трех измерений с различным N, при этом максимальное время работы не должно превышать 10 мин.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Характеристики используемого оборудования (включая описание кластера) и программных средств.
4. Описание параллельного алгоритма, используемого при решении задачи.
5. Результаты работы программы для нескольких случаев (таблица и график).
6. Анализ результатов программы.
7. Выводы.
8. Приложение 1. Тексты программ.
9. Приложение 2. Журнал работы программы для решения задачи с максимальной размерностью N.

Вопросы для контроля знаний

1. Какие премущества и недостатки имеет параллельная программа с распределенной памятью?
2. Перечислите основные топологии, используемые при вычислениях на кластере. Какие преимущества дает использование топологий при вычислениях?
3. Какие топологии кластера поддерживает стандарт MPI?

4. Как происходит запуск параллельной программы на кластере? Можно ли запустить несколько копий программы на одном вычислительном узле?

5. Какие функции используются для инициализации и завершения работы параллельной программы с использованием MPI? Какие требования предъявляются к вызовам других функций?

6. Назовите основные функции, используемые для коммуникации “точка-точка”.

7. Являются ли функции коммуникации “точка-точка” достаточными для написания сложных программ?

8. Перечислите виды коллективных операций, предусмотренных стандартом MPI.

9. В чем состоит основное отличие векторых вариантов функций коллективного взаимодействия от их “обычных” аналогов?

10. Что такое коммуникатор? Какие коммуникаторы предопределены в MPI?

11. Что такое ранг процесса? При помощи какой функции его можно узнать?

12. Как можно определить количество запущенных процессов? Может ли оно меняться в течение работы программы?

13. Какие наследуемые типы данных описаны в стандарте MPI? Для чего используются псевдо-типы MPI_LB, MPI_UB?

14. Опишите основные шаги алгоритма при умножении матриц на топологии двухмерный массив.

15. Опишите основные шаги алгоритма при умножении матриц на топологии куб.

Лабораторная работа 8
ТЕХНОЛОГИЯ CUDA

Цели работы

Знакомство с особенностями параллельных вычислений с использованием видеоадаптеров. Определение оптимальной конфигурации программы для достижения максимальной эффективности вычислений.

Получение практического опыта по применению технологии CUDA для фильтрации изображений.

Необходимое оборудование и программное обеспечение

Для выполнения лабораторной работы необходимо следующее оборудование и программное обеспечение:

– персональный компьютер с видеоадаптером, поддерживающим технологию CUDA;

– установленная ОС Windows или Linux;

– установленный драйвер для видеоадаптера, позволяющий использовать технологию CUDA;

– набор инструментальных средств разработки для технологии CUDA (NVIDIA CUDA Toolkit);

– компилятор для языка C/C++ для разработки части программы, выполняемой на центральном процессоре.

Полный список видеоадаптеров с поддержкой технологии CUDA можно найти на сайте компании NVIDIA: https://developer.nvidia.com/cuda-gpus.

При использовании ОС Windows рекомендуется использовать интегрированную среду разработки Microsoft Visiaul Studio 2008 или более поздние версии.

Гетерогенные параллельные вычисления

Гетерогенными называются вычисления, которые выполняются с использованием различных видов вычислительных устройств. В качестве таких устройств могут выступать процессоры общего назначения (GPP – General Purpose Processors), процессоры цифровой обработки сигналов (DSP – Digital Signal Processors), графические процессоры (GPU – Graphical Processing Unit), арифметические сопроцессоры, процессоры специального назначения и т.д. Ключевым аспектом является то, что различные устройства построены на архитектурах с отличающимися наборами команд.

В данной работе мы будем рассматривать вычисления с использованием графических процессоров. Начало вычислений на видеоадаптерах тесно связано с программируемыми шейдера-

ми – небольшими программами, используемыми на одной из ступеней графического конвейера для окончательного формирования изображения. В большинстве ОС время работы шейдерных программ ограничено (не более 5 с), если устройство используется для формирования изображения (подключено к монитору).

В первом приближении можно сказать, что вычисление с использованием видеоадаптера состоит из следующих этапов:

– пересылка исходных данных из ОЗУ в видеопамять;

– вычисление при помощи графического процессора;

– пересылка результатов вычислений из видеопамяти в ОЗУ.

В некоторых случаях можно обойтись без пересылки данных (например, в случае обработки видео, результат вычислений может быть просто выведен на экран монитора), но в большинстве случаев это не так.

Рассматриваемая в данной главе технология CUDA разработана компанией NVIDIA и берет начало в потоковом мультипроцессоре, впервые появившемся в серии GeForce 8 в 2006 г.

Потоковый мультипроцессор серии GeForce 8

Архитектура видеоадаптера серии GeForce 8 схематично представлена на рис. 8.1.

Восемь одиночных процессоров способны параллельно выполнять 32 потока за 4 такта. Группа из 32 потоков называется **warp**. При вычислении сложных функций (например, тригонометрических) выполнением занимаются два специальных процессора, которые выполняют те же 32 потока за 16 тактов. В состав потокового мультипроцессора входят также 8192 32-битных регистра и 16 Кб разделяемой (общей) памяти. Один потоковый мультипроцессор поддерживал выполнение до 768 активных потоков. Потоковые мультипроцессоры объединялись в кластеры текстурных процессоров, где имелась дополнительная память для хранения текстур, которые затем объединялись в массив потоковых процессоров.

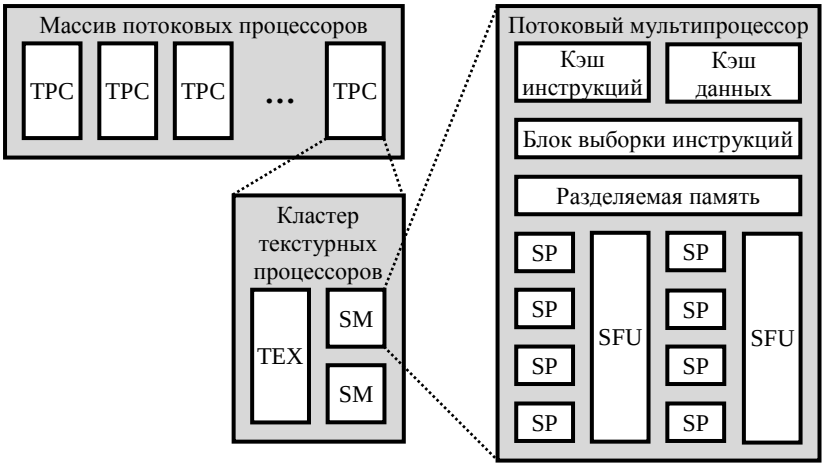


Рис. 8.1. Архитектура видеоадаптеров серии GeForce 8: SP – одиночный процессор (Single Processing core); SFU – специальный процессор (Super Function Unit); SM – потоковый мультипроцессор (Streaming Multiprocessor); TEX – текстурная память; TPC – кластер текстурных процессоров (Texture Processor Cluster)

Для сравнения: в состав мультипроцессора на базе ядра Kepler (версия CUDA 3.5) входят 192 одиночных процессора и 32 специальных процессора для вычисления сложных функций.

Модель выполнения

Потоки могут сгруппированы в одно-, двух- или трехмерные блоки (рис. 8.2). Блоки, в свою очередь, организуются в одно-, двух, или трехмерный массив.

Потоки одного блока могут взаимодействовать друг с другом, используя разделяемую память, и синхронизировать свое выполнение. Потоки из разных блоков не могут взаимодействовать друг с другом. При запуске каждой новой задачи может использоваться своя конфигурация блоков и потоков.

Блоки потоков распределяются между потоковыми мультипроцессорами, отдельный блок выполняется на отдельном мультипроцессоре. При этом на одном мультипроцессоре может одновременно выполняться более одного блока (если позволяют ресурсы).

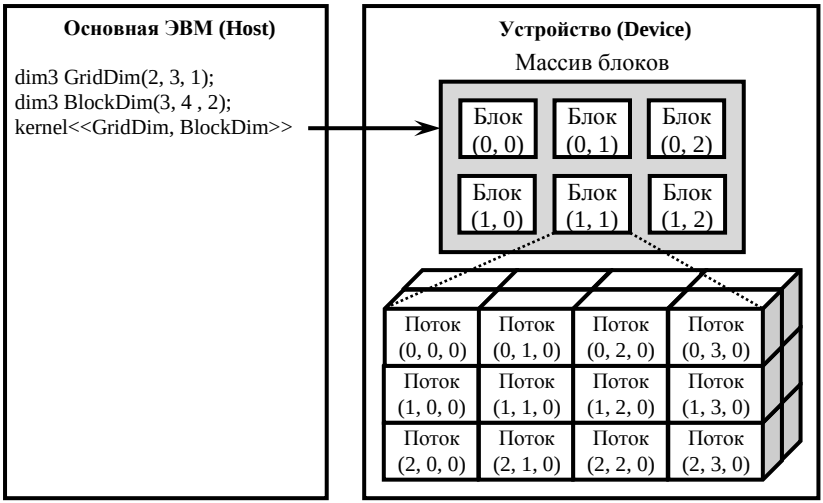


Рис. 8.2. Модель выполнения CUDA

Каждый поток может определить свое местоположение при помощи встроенных переменных, все они приведены в табл. 8.1.

Таблица 8.1

Встроенные переменные

Имя	Тип	Описание
gridDim	dim3	Размерность массива блоков (количество блоков по каждому измерению)
blockIdx	dim3	Индекс текущего блока внутри массива
blockDim	dim3	Размерность блока (количество потоков в блоке по каждому измерению)
threadIdx	dim3	Индекс потока внутри блока
warpSize	int	Количество потоков в одном warp (пока всегда 32)

Потоки объединяются в группы по 32 потока, называемые “warp” (число 32 связано с аппаратными особенностями и может быть изменено в будущем). Планировщик задач оперирует понятием “warp”, а не “отдельный поток”.

Максимальное количество warp, одновременно выполняющихся на одном мультипроцессоре, зависит от версии CUDA,

поддерживаемой устройством. В табл. 8.2 представлены основные характеристики устройств, в зависимости от версии CUDA.

Таблица 8.2

Вычислительные возможности устройства в зависимости от поддерживаемой версии CUDA (по данным NVIDIA)

Характеристика	**Номер версии CUDA**					
	1.x	**2.x**	**3.0**	**3.5**	**5.0**	**5.2**
Максимальное количество блоков в массиве по X-измерению	2^{16}-1		2^{31}-1			
Максимальное количество блоков в массиве по Y-измерению	2^{16}-1					
Максимальное количество блоков в массиве по Z-измерению	—	2^{16}-1				
Максимальное количество потоков в блоке по X-измерению и Y-измерению	512	1024				
Максимальное количество потоков в блоке по Z-измерению	64					
Максимальное количество потоков в блоке (суммарное, X*Y*Z)	512	1024				
Размер warp	32					
Максимальное количество выполняющихся warp на мультипроцессоре	24 или 32	48	64			
Максимальное количество выполняющихся потоков на мультипроцессоре	768 или 1024	1536	2048			
Максимальное количество выполняющихся блоков на мультипроцессоре	8		16		32	
Количество 32-битных регистров в мультипроцессоре	8K или 16K	32K	64K			

Продолжение таблицы 8.2

Характеристика	Номер версии CUDA					
	1.x	2.x	3.0	3.5	5.0	5.2
Максимальное количество 32-битных регистров на поток	128	63		255		
Максимальное количество разделяемой памяти в мультипроцессоре	16Кб	48Кб			64Кб	96Кб
Размер константной памяти	64Кб					
Максимальное количество инструкций в потоке	2 млн	512 млн				

Эти и другие характеристики для конкретного устройства можно получить во время выполнения программы с помощью функции **cudaGetDeviceProperties**.

Типы памяти

Время доступа к памяти играет немаловажное значение при вычислениях с помощью CUDA и оно сильно варьируется в зависимости от типа памяти. Типы памяти и взаимодействия потоков с памятью представлено на рис. 8.3.

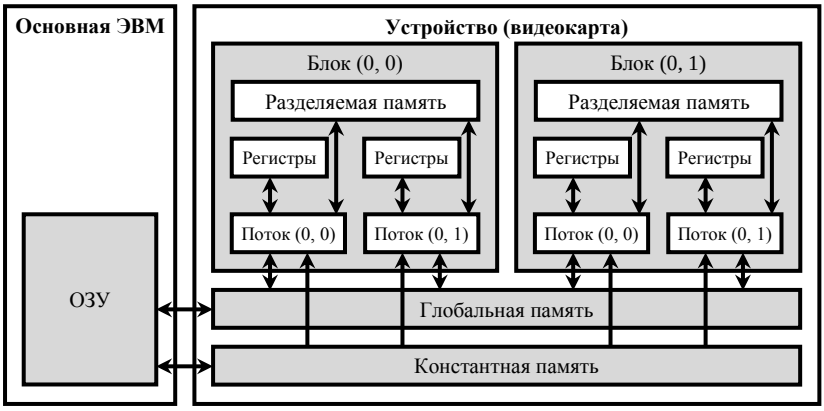


Рис. 8.3. Организация памяти в CUDA

Самыми быстрыми являются операции чтения и записи регистров, они выполняются примерно за один такт. Такое быстродействие обусловлено тем, что регистры располагаются непосредственно внутри потокового мультипроцессора и каждый мультипроцессор содержит тысячи таких 32-битных регистров (точное число указано в табл. 8.2). Однако, если учесть, что на одном мультипроцессоре может выполняться до 2048 потоков, то количество регистрок из расчета на поток оказывается не таким большим. Регистры используются для локальных переменных потока, и если суммарное число переменных для всех работающих потоков окажется больше, чем регистров, то в этом случае часть локальных переменных будет размещена во внешней памяти, что отрицательно скажется на производительности.

Помимо регистров, внутри мультипроцессора также располагается разделяемая память. Операции с ней проходят несколько медленнее, на их выполнение требуется порядка 5 тактов. Общая память используется для обмена данными между потоками одного блока и доступна как на чтение, так и на запись.

Все остальные типы памяти располагаются вне мультипроцессора. При этом наименьшим временем доступа обладает константная память, из потоков она доступна только на чтение. Объем константной памяти невелик – всего 64 Кб, но время операций сопоставимо с операциями с разделяемой памятью. Как и следует из названия, константная память используется для хранения данных, которые не меняются во время выполнения. Запись в константную память выполняется из основной ЭВМ.

В качестве глобальной памяти используется DDR – Double Data Rate память. Объем и тип именно этой памяти указывают в спецификации на видеоадаптер. Также в спецификации можно найти частоту памяти, разрядность шины и пропускную способность. Эти параметры оказывают серьезное влияние на время вычислений. Глобальная память используется для хранения исходных данных и сохранения результатов работы. Обмен информацией между глобальной памятью и основной ЭВМ осуществляется по шине PCI-E.

Помимо указанных на рисунке, имеется еще текстурная память, доступная потокам только на чтение. Данная память кэшируется особым образом для оптимизации операций с двумер-

ными данными: например, двухмерной сверткой. Это позволяет потокам одного warp достичь высокого быстродействия при работе с близко расположенными адресами текстуры по обоим измерениям.

Эффективная работа с памятью

Начиная с версии 2.x, для операций с глобальной памятью используется двухуровневый кэш. Кэш первого уровня L1 у каждого мультипроцессора свой, а кэш второго уровня L2 – общий для всех. Кэш первого уровня использует ту же встроенную в кристалл память, что и разделяемая память. С помощью вызова функции **cudaFuncSetCacheConfig** можно сконфигурировать, сколько памяти будет использоваться под кэш, а сколько под разделяемую память. Возможны два варианта: 16 Кб под кэш L1 и 48 Кб под разделяемую память, либо 48 Кб под кэш L1 и 16 Кб под разделяемую память. По умолчанию используется первый вариант. Максимальный размер кэша L2 может достигать 768 Кб для версии 2.x и 1,5 Мб для версии 3.x.

Рассмотрим, как выполняются операции обращения к памяти из потоков, входящих в состав одного warp. Для примера будем считать, что каждый поток выполняет 4-байтовую операцию доступа к памяти.

Кэш оперирует с блоками из 128 байт, выровненных по границе 128 байт. В первом случае (рис. 8.4*, а)* доступ из потоков последовательный и выровненный по границе блока в 128 байт. В этом случае все потоки смогут выполнить операции за одну транзакцию. Во втором случае (рис. 8.4, *б)* доступ не выравнен по границе блока и потребуется две транзакции.

Третий случай (рис. 8.4, *в)* – непоследовательный доступ внутри одного блока. При этом не важно, по каким именно адресам обращаются потоки, важно лишь, чтобы все эти адреса оставались внутри одного блока. В этом случае операция доступа также может быть выполнена за одну транзакцию.

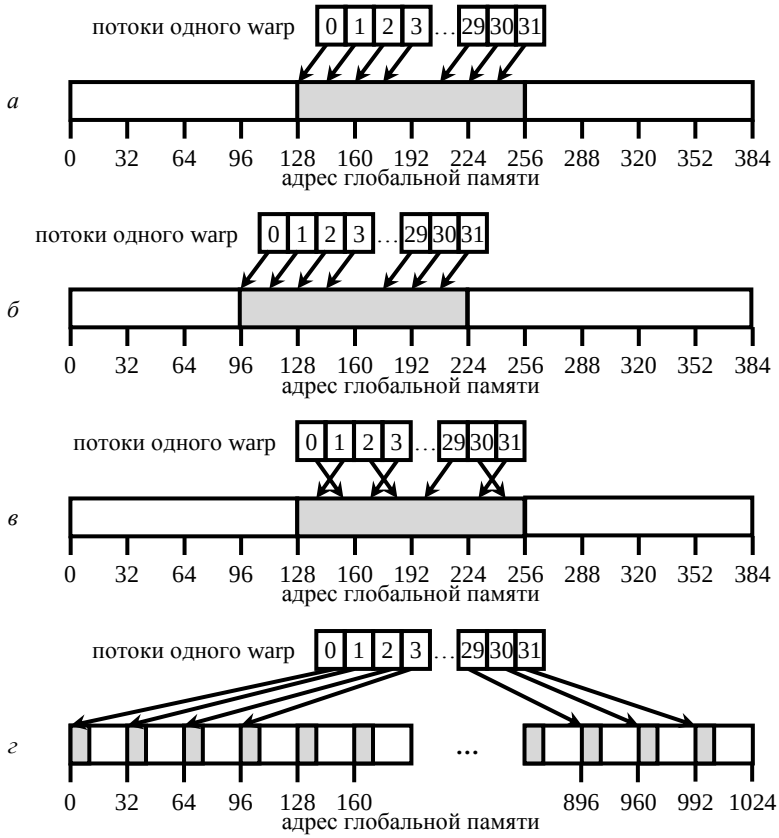


Рис. 8.4. Операции с памятью из потоков, входящих в состав одного warp: *а* – последовательный выровненный по границе блока доступ; *б* – последовательный невыровненный доступ; *в* – непоследовательный выровненный по границе блока доступ; *г* – доступ со смещением в 32 байта

Четвертый случай, худший вариант (рис. 8.4, *г*), – когда потоки обращаются по различным адресам в разных блоках. Для приведенного примера потребуется 8 транзакций. В этом случае потоки проведут значительную часть времени в ожидании завершения операции с памятью, что может свести на нет весь выигрыш от использования параллельных вычислений. Подобных ситуаций следует избегать.

Программирование на CUDA C

Для программирования в технологии CUDA используется синтаксис языка C. Для разработки программ необходим программный пакет CUDA Toolkit, а для их выполнения – драйвер устройства с поддержкой CUDA (обычно устанавливается в комплекте с CUDA Toolkit). В состав программного пакета входит специальный компилятор языка C – NVCC. Файлы предназначенные для компилирования при помощи NVCC обычно имеют расширение ".cu".

В состав программы входят участки кода (функции), которые выполняются на основной ЭВМ, и участи кода, которые выполняются на устройстве (видеоадаптере). Для того, чтобы компилятор различал, какие функции должны выполняться на основной ЭВМ, а какие на устройстве, используются ключевые слова (табл. 8.3).

Таблица 8.3

Модификаторы функций

Модификатор	**Откуда вызывается**	**Где выполняется**
__host__	основная ЭВМ	основная ЭВМ
__device__	устройство	устройство
__global__	основная ЭВМ	устройство

Модификаторы доступа __host__ и __device__ могут использоваться совместно, при этом будет создано две реализации функции – для выполнения на основной ЭВМ и устройстве соответственно.

Во время компиляции код, предназначенный для выполнения на ЭВМ, компилируется стандартными средствами, а код, предназначенный для выполнения на устройстве, преобразуется в PTX (Parallel Thread Execution), который обрабатывается непосредственно во время выполнения программы JIT-компилятором устройства (Just-In-Time Compiler) (рис. 8.5).

При написании исполняемых на устройстве функций можно указать модификатор переменной, который определит, в какой памяти она будет располагаться (а значит и время доступа к ней) (табл. 8.4).

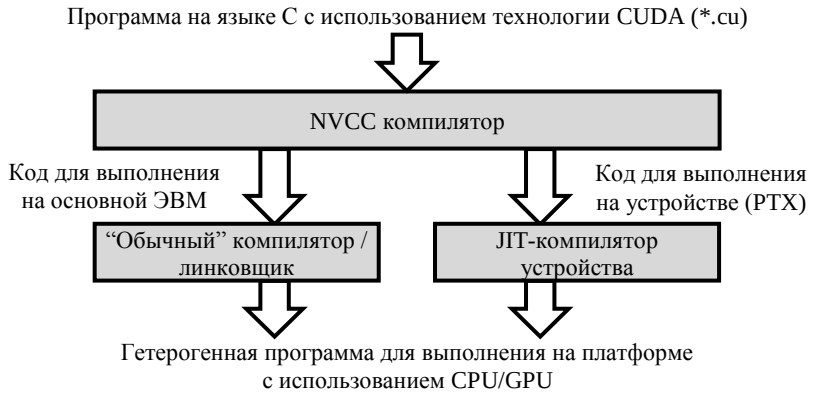


Рис. 8.5. Компиляция программы при помощи NVCC

Таблица 8.4

Модификаторы переменных

Варианты объявления переменных	**Тип памяти**	**Область видимости**	**Время жизни**
int i;	регистры	поток	поток
__device__ __shared__ int i; __shared__ int i;	разделяемая	блок	блок
__device__ int i;	глобальная	массив	прило-жение
__device__ __constant__ int i; __constant__ int i;	константная	массив	прило-жение

В состав CUDA Toolkit входит прикладной программный интерфейс среды выполнения – CUDA Runtime API, который представляет собой набор C/C++ функций для выполнения на основной ЭВМ и предназначенных для получения информации об устройстве, управления его конфигурацией, запуска вычислительных задач и т. п.

При успешном выполнении функции из CUDA Runtime API возвращают значение cudaSuccess; если же выполнение завершилось неудачно, то код ошибки можно получить при помощи вызова функции cudaGetLastError. В связи с этим полезным может оказаться следующий макрос:

```
#define CUDA_CHECK(call) \
if((call) != cudaSuccess) { \
cudaError_t err = cudaGetLastError(); \
   printf( "\"%s\" failed (%d)\n", #call, err); \
   exit(-1); }
```

Он проверяет результат выполнения и в случае ошибки выводит сообщение с кодом ошибки и завершает программу.

Дивергенция выполнения

При выполнении кода на устройстве используется концепция SIMD (Single Instruction Multiply Data), т.е. одна инструкция одновременно выполняется применительно к различным данным. Потоки внутри одного warp не могут одновременно выполнять разные инструкции, например:

```
if (threadIdx.x % 2)
{
   // четные потоки простаивают
}
else
{
   // нечетные потоки простаивают
}
```

В результате выполнения такого кода одна половина потоков будет простаивать во время выполнения ветки if, а другая – во время выполнения ветки else. Данная ситуация называется дивергенцией выполнения и негативно сказывается на общей производительности. Желательно, чтобы все потоки одного warp попадали либо в ветку if, либо в ветку else.

Пример 8.1. Умножение матриц

Рассмотрим, как можно организовать вычисления на устройстве для задачи перемножения двух матриц. Пусть размерность исходных матриц N×L и L×M, тогда размерность результирующей матрицы будет N×M.

При запуске программы мы должны убедиться в том, что необходимое оборудование установлено в системе и, если их

несколько, выбрать то, на котором будут выполнятся вычисления. Такое поведение реализовать достаточно просто, используя функции **cudaGetDeviceCount** и **cudaSetDevice**:

```
// заголовочные файлы CUDA
#include <cuda.h>
#include <cuda_runtime.h>

int main(int argc, char* argv[])
{
   // проверка на наличие устройств CUDA
   int deviceCount = 0;
   CUDA_CHECK(cudaGetDeviceCount(&deviceCount));
   if (deviceCount == 0)
   {
      printf("There are no CUDA device(s)\n");
      return 0;
   }

   // используем первое устройство
   cudaSetDevice(0);
```

Следующим шагом в программе будет выделение глобальной памяти на устройстве (видеопамяти) под хранение матриц. Следует обратить внимание на то, что при выделении памяти размер указывается в байтах:

```
// указатели на участки видеопамяти
float *dev_A, *dev_B, *dev_C;

// размеры матриц в байтах
size_t sizeA = N*L*sizeof(float);
size_t sizeB = L*M*sizeof(float);
size_t sizeC = N*M*sizeof(float);

// выделение видеопамяти
CUDA_CHECK(cudaMalloc((void**)&dev_A, sizeA));
CUDA_CHECK(cudaMalloc((void**)&dev_B, sizeB));
CUDA_CHECK(cudaMalloc((void**)&dev_C, sizeC));
```

Теперь можно выполнить пересылку исходных матриц из ОЗУ в видеопамять (глобальную память устройства):

```
// копирование исходных матриц в видеопамять
CUDA_CHECK(cudaMemcpy(dev_A, host_A, sizeA,
   cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(dev_B, host_B, sizeB,
   cudaMemcpyHostToDevice));
```

Мы копируем только матрицы A и B, так как в копировании матрицы C нет никакого смысла, поскольку она еще не содержит полезных данных. Копирование осуществляется при помощи функции **cudaMemcpy** с четырьмя аргументами: первый и второй аргументы – это приемник (куда копируем) и источник (откуда копируем), третий параметр задает размер копируемой памяти в байтах, а четвертый задает направление копирования, в нашем случае из ОЗУ в память устройства.

Теперь все готово для запуска вычислений на устройстве. Функция вычислений у нас находится в другом файле (MultMatrixKernel.cu), который будет обрабатываться компилятором NVCC, поэтому необходимо указать, что данная функция экспортируется при помощи extern “C” (она будет выполненяться на основной ЭВМ):

```
extern "C"
void multMatrix(float *A, float *B, float *C,
  unsigned int N, unsigned int L, unsigned int M)
{
   // размер блока
   dim3 BlockDim(16, 16, 1);

   // размер вычислительного массива
   dim3 GridDim(
      (N + BlockDim.x - 1)/BlockDim.x,
      (M + BlockDim.y - 1)/BlockDim.y, 1);

   // запуск вычислений
   mult_matrix_kernel<<<GridDim, BlockDim>>>(
       A, B, C, N, L, M);
}
```

Каждый отдельный поток будет вычислять отдельный элемент результирующей матрицы. Потоки сгруппируем в блоки по 16×16 элементов. Общее число потоков в блоке будет равно

256, что эквивалентно 8 warp. Исходя из размеров исходных матриц и размеров блока, определим количество блоков, необходимых для вычисления матрицы целиком.

Последняя строка в этой функции непосредственно запускает вычисления на устройстве (видеоадаптере) с указанной организацией блоков и потоков. Рассмотрим теперь код, который будет выполняться на самом устройстве:

```
__global__ void mult_matrix_kernel(
  float *A, float *B, float *C,
  unsigned int N, unsigned int L, unsigned int M)
{
   // определяем место потока в массиве
   unsigned int tx = threadIdx.x;
   unsigned int ty = threadIdx.y;
   unsigned int bx = blockIdx.x;
   unsigned int by = blockIdx.y;

   // определяем рассчетный элемент матрицы
   unsigned int row = by * blockDim.y + ty;
   unsigned int col = bx * blockDim.x + tx;

   if (row < N && col < M)
   {
      float sum = 0.0f;
      for (unsigned int i = 0; i < L; ++i)
         sum += A[row*L + i]*B[i*M + col];
      C[row*M + col] = sum;
   }
}
```

Поскольку данный код предназначен для выполнения на устройстве, но вызывается с основной ЭВМ, то используем модификатор функции **__global__**. При помощи встроенных переменных **threadIdx**, **blockIdx** и **blockDim** определяем место потока в вычислительном массиве и индекс элемента матрицы, который поток должен вычислить. Необходимо также проверить выход за границы матрицы, так как при делении результирующей матрицы C на блоки часть потоков может оказаться лишними (рис. 8.6).

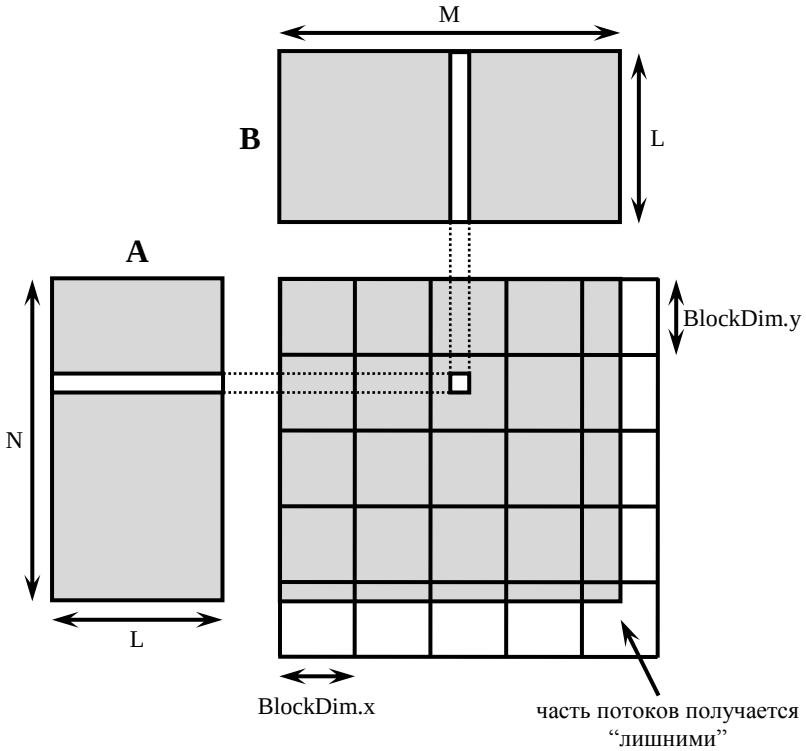


Рис. 8.6. Умножение матриц

При вычислении элемента результирующей матрицы каждый поток производит L чтений элементов матрицы A и L чтений элементов матрицы B, а также запись одного элемента результирующей матрицы. Итого мы получаем 2L+1 обращений к глобальной памяти для каждого потока.

После того, как вычисления завершатся, нам останется только скопировать результаты в ОЗУ и освободить память на устройстве:

```
// копирование результирующей матрицы в ОЗУ
CUDA_CHECK(cudaMemcpy(host_C, dev_C, sizeC,
   cudaMemcpyDeviceToHost));
```

```
// освобождение видеопамяти
CUDA_CHECK(cudaFree(dev_A));
CUDA_CHECK(cudaFree(dev_B));
CUDA_CHECK(cudaFree(dev_C));
```

Собрав все части воедино, мы получим рабочую программу перемножения матриц с использованием технологии CUDA.

Тем не менее разработанная нами программа не является оптимальной. Внутри блока каждый поток обращается за элементами исходных матриц независимо от других, хотя ряд исходной матрицы A используется для вычисления всех элементов блока в этом же ряде, а колонка исходной матрицы B для вычисления всех элементов в этой же колонке.

В рассмотренном примере каждый блок состоит из 256 потоков, каждый поток производит 2L чтений из глобальной памяти или 512L чтений на блок (тут надо отметить, что большинство таких вызовов тем не менее объединяется в транзакции).

Число чтений из глобальной памяти можно сократить. При работе блока нам требуются 16L элементов из матрицы A (16 рядов по L элементов) и 16L элементов из матрицы B (16 колонок по L элементов). При этом каждый элемент будет использоваться 16 раз (по одному разу в 16 различных потоках внутри блока). Таким образом, число чтений элементов из глобальной памяти сократится до 32L или в 16 раз.

К сожалению, мы не можем прочитать все необходимые нам элементы сразу, так как объем разделяемой памяти у нас ограничен. При этом не стоит также забывать, что число блоков, выполняющихся на мультипроцессоре, может быть больше одного и разделяемая память используется всеми блоками.

Вместо этого мы будет читать элементы исходных матриц блоками по 16×16 элементов и накапливать частичную сумму произведений элементов.

При этом работа потока будет теперь состоять из следующих шагов:

– копирование одного элемента из исходной матрицы A в разделяемую память;

– копирование одного элемента из исходной матрицы B в разделяемую память;

– ожидание, когда остальные потоки завершат копирование элементов в разделяемую память;

– накопление суммы произведений элементов исходных матриц;

– ожидание завершения вычислений другими потоками блока;

– повторение всех предыдущих шагов для следующих блоков элементов исходных матриц;

– сохранение результатов вычислений (элемента результирующей матрицы).

На рис. 8.7 приведена схема работы данного алгоритма.

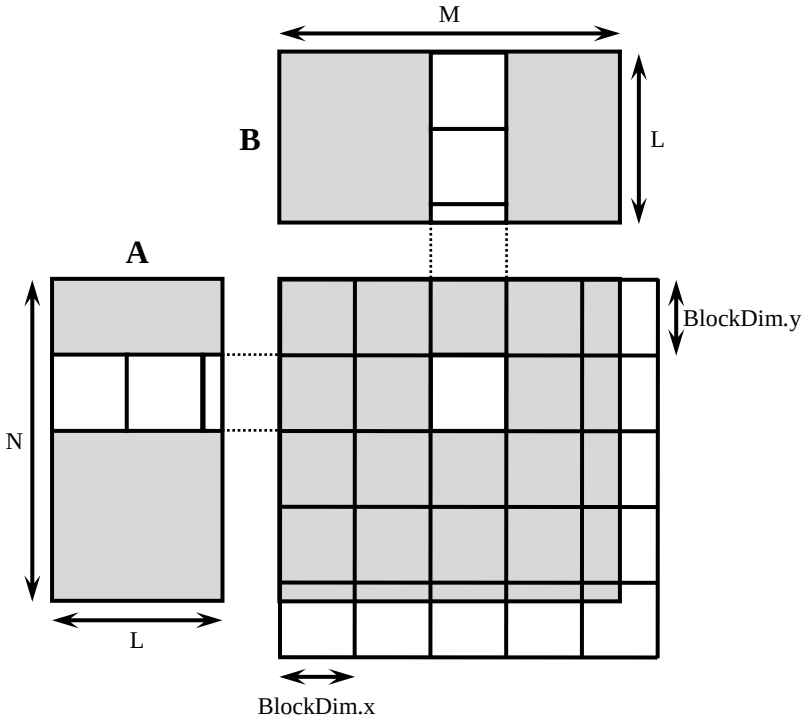


Рис. 8.7. Умножение матриц с использованием разделяемой памяти

При этом количество циклов составит ⌈L/16⌉, а функция будет выглядеть следующим образом:

```
__global__ void mult_matrix_kernel(
   float *A, float *B, float *C, unsigned int N,
   unsigned int L, unsigned int M)
{
   // выделение разделяемой памяти для хранения
   // блокоd элементов исходных матриц
   __shared__ float ds_A[16][16];
   __shared__ float ds_B[16][16];

   // определяем место потока в массиве
   unsigned int tx = threadIdx.x;
   unsigned int ty = threadIdx.y;
   unsigned int bx = blockIdx.x;
   unsigned int by = blockIdx.y;

   // определяем рассчетный элемент матрицы
   unsigned int row = by * blockDim.y + ty;
   unsigned int col = bx * blockDim.x + tx;

   float sum = 0;
   unsigned int nCycles = (L - 1)/16 + 1;
   for (int i = 0; i < nCycles; ++i) {
      // копируем элемент исходной матрицы A
      if (row < N && i*blockDim.x + tx < L)
         ds_A[ty][tx] = A[row*N + i*blockDim.x+ tx];
      else
         ds_A[ty][tx] = 0;

      // копируем элемент исходной матрицы B
      if (i*blockDim.x + ty < L && col < M)
         ds_B[ty][tx] = B[(i*blockDim.x + ty)*M+ col];
      else
         ds_B[ty][tx] = 0;

      __syncthreads();

      // накопление частичной суммы
      for (int j = 0; j < blockDim.x; ++j)
         sum += ds_A[ty][j] * ds_B[j][tx];

      __syncthreads();
   }

   // запись элемента результирующей матрицы
   if (row < N && col < M)
      C[row*M + col] = sum;
}
```

Следует обратить внимание на то, как используются барьеры синхронизации. Первый барьер не позволяет начинать потокам накопление частичной суммы, пока не все данные для этого готовы. Второй барьер не позволяет потокам начать обновлять данные в разделяемой памяти, пока все потоки не закончат расчет накопленной суммы.

Если сравнить время вычислений, то для матриц размером 200×200 элементом оно составило 63 мс для простого алгоритма и 49 мс для алгоритма с использованием разделяемой памяти. Таким образом, мы улучшили работу программы на 22%. Полученные результаты зависят от используемого оборудования и при проведении самостоятельных тестов могут значительно отличаться.

Студент может сравнить полученные результаты с программой перемножения матриц из главы, посвященной OpenMP.

Измерение времени операций

Большинство вызовов функций из CUDA Runtime API являются асинхронными, т.е. управление может быть сразу возвращено в основную программу. Поэтому стандартные средства для измерения временных интервалов здесь не подходят. Для измерений временных интервалов используется механизм событий.

Функция **cudaEventRecord** дает команду о записи времени события, когда дойдет очередь до выполнения этой команды на устройстве. Второй параметр этой функции – номер потока. Под потоком в данном случае понимается очередь команд, выполняющихся в заданном порядке.

Функция **cudaEventSynchronize** позволяет синхронизоровать выполнение программы на основной ЭВМ с наступлением события при вычислении на устройстве. Без подобной синхронизации нельзя гарантировать достоверность результатов измерений.

Функция **cudaEventElapsedTime** позволяет получить время в секундах между двумя событиями.

Приведенный ниже пример иллюстрирует использование описанных функций.

```
cudaEvent_tevt1, evt2;

// создание событий
cudaEventCreate(&evt1);
cudaEventCreate(&evt2);

// команда фиксирования времени первого события
cudaEventRecord(evt1, 0);

// участок кода для оценки времени выполнения
...

// команда фиксирования времени второго события
cudaEventRecord(evt2, 0);

// ждем наступление второго события
cudaEventSynchronize(evt2);

float seconds;
cudaEventElapsedTime(&seconds, evt1, evt2);

// вывод затраченного времени
printf("Execution time   : %.5f s\n", seconds);

// удаление событий
cudaEventDestroy(evt1);
cudaEventDestroy(evt2);
```

Рекомендации по достижению максимальной эффективности

Для достижения максимальной производительности необходимо выполнить три простых условия: максимально использовать доступные ядра, эффективно работать с памятью и обеспечить эффективное выполнение инструкций. В этой главе мы перечислим основные правила, позволяющие получить максимальную производительность в большинстве случаев.

1. Размер блока (количество потоков в блоке) следует выбирать кратным размеру warp.

2. Общее количество блоков должно быть больше, чем число потоковых мультипроцессоров (как минимум, в два раза).

3. Везде, где это возможно, следует не допускать дивергенции выполнения потоков внутри одного warp.

4. Необходимо минимизировать обращения к глобальной памяти и использовать разделяемую память.

5. При доступе к глобальной памяти следует добиваться меньшего числа транзакций.

6. Если устройство и алгоритм обработки данных позволяет, то следует совмещать копирование новой порции данных с обработкой текущей порции (рис. 8.8).

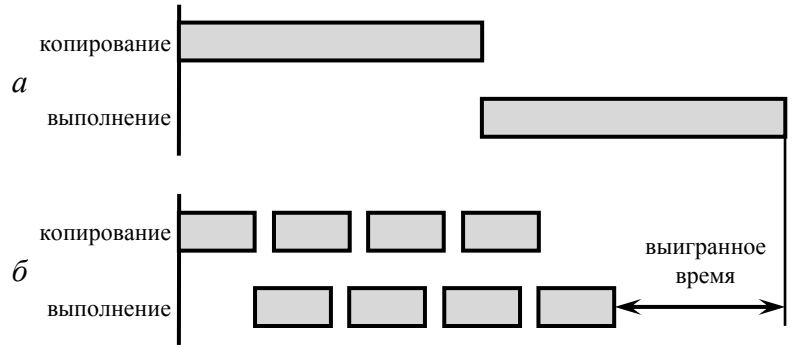


Рис. 8.8. Сравнение затраченного времени при последовательном и совмещенном выполнении операций копирования и выполнения: *а* – последовательное выполнение; *б* – совмещенное выполнение

Задание для лабораторной работы

Необходимо реализовать для обработки изображений. Исходное изображение должно загружаться из входного файла, результат тоже должен сохраняться в файл. Имена входного и выходного файлов должны задаваться через парметры командной строки. Формат входного и выходного файла может быть любым распространенным графическим форматом по выбору студента. Рекомендуется использовать формат BMP или PPM (из-за их простоты). Все обрабатываемые графические изображения должны иметь глубину 24 бита.

Вариант индивидуального задания выбирается из табл. 8.5 в соответствии с порядковым номером студента в списке группы по модулю 5.

Таблица 8.5

Варианты фильтров для индивидуального задания

Номер варианта	**Фильтр 1**	**Фильтр2**
1	Преобразование в оттенки серого	Медианный фильтр
2	Коррекция насыщенности	Нерезкое маскирование
3	Коррекция яркости	Размытие по Гауссу
4	Получение негатива изображения	Повышение резкости изображения
5	Гамма-коррекция	Обнаружение краев

Первый фильтр при обработке конкретного пикселя не требует значения соседних пикселей. Его надо реализовать в двух вариантах:

а) когда параллельно выполняющиеся потоки, входящие в состав одного warp, будут обрабатывать пиксели в соседних колонках одного ряда;

б) когда параллельно выполняющиеся потоки, входящие в состав одного warp, будут обрабатывать пиксели в соседних рядах одной колонки.

Второй фильтр при вычислении значения конкретного пикселя требует значения соседних пикселей. Его также необходимо реализовать в двух вариантах:

а) без использования разделяемой памяти;

б) с использованием разделяемой памяти – в этом случае потоки одного блока смогут многократно использовать значение одного пикселя при вычислении соседних к нему пикселей.

С разработанной программой необходимо провести ряд экспериментов:

1. Убедиться в том, что программа корректно обрабатывает изображения, чей размер не кратен размеру блока. Для этого взять изображение, где в качестве ширины и высоты используются простые числа больше 1000.

2. Отдельно для каждой реализации каждого фильтра измерить время следующих операций:

– выделения памяти на видеоадаптере;

– копирования исходного изображения в память видеоадаптера;

– обработки изображения при помощи фильтра;

– копирование обработанного изображения из памяти видеоадаптера.

По результатам измерений заполнить табл. 8.6 и объяснить полученные результаты.

Таблица 8.6

Время выполнения операций

Время операции	**1, а**	**1, б**	**2, а**	**2, б**
Выделение памяти				
Пересылка исходных данных из ОЗУ в видеопамять				
Вычисление результирующего изображения				
Пересылка результатов из видеопамяти в ОЗУ				

3. Выбрать наилучшую реализацию для каждого фильтра и определить, как изменится время операций в зависимости от размера изображения. Провести не менее 5 экспериментов для каждого фильтра. Свести данные в таблицу, аналогичную предыдущей, и построить график зависимости времени вычисления результирующего изображения от размера изображения в Мрх для каждого из фильтров. В качестве минимального размера изображения рекомендуется использовать изображения от 0.3 до 1 Мрх, а в качестве максимального – более 10 Мрх.

4. Для второго фильтра определить, как размер блока сказывается на производительности вычислений. Для временных оценок использовать изображение максимального размера из предыдущего пункта. Размер блока рекомендуется выбирать таким, чтобы общее число потоков в блоке было кратно количеству потоков в одном warp (т.е. кратно 32), например, 8×8, 8×16, 16×8, 16×16, 32×32 и т.д. Провести не менее пяти измерений с

различным размером блока. Для сравнения выполнить один дополнительный тест с блоком 7×7. Результаты свести в единую таблицу. Сделать выводы.

Содержание отчета

1. Титульный лист.
2. Цель работы.
3. Индивидуальное задание.
4. Характеристики видеоадаптера, используемого для выполнения работы (могут быть получены при помощи специальных функций).
5. Заполненная по результатам экспериментов таблица 8.4.
6. Таблица временных затрат на различные операции для изображений различных размеров для обоих фильтров.
7. Графики зависимости времени вычисления результирующего изображения от размера изображения в Мрх (по 1 графику для каждого фильтра).
8. Таблица времени вычисления результирующего изображения для второго фильтра в зависимости от размера блока.
9. Выводы.
10. Приложение 1. Тексты программы для фильтра 1 (только часть выполняющаяся на видеоадаптере).
11. Приложение 2. Тексты программы для фильтра 2 (только часть выполняющаяся на видеоадаптере).
12. Приложение 3. Примеры обработки изображений фильтрами (до и после обработки каждым из фильтров). Для большей наглядности лучше использовать увеличенный фрагмент изображения, а не изображение целиком.

Вопросы для контроля знаний

1. Опишите назначение основных компонентов, входящих в состав потокового мультипроцессора серии G80.
2. Что означает термин “гетерогенные вычисления”?
3. Какая модель параллельных вычислений используется в технологии CUDA?
4. Что такое warp и дивергенция выполнения?

5. Как можно узнать характеристики устройства с поддержкой технологии CUDA?

6. Какие ограничения накладываются на размерность структур для группировки потоков?

7. Как поток может определить свое место относительно других потоков?

8. Перечислите основные виды памяти, доступные потоку. Какие из них доступны на запись? Как указать в какой памяти должна располагаться переменная?

9. Оцените время доступа к каждому виду памяти относительно друг друга.

10. Как происходит пересылка данных из ОЗУ в видеопамять и обратно?

11. Почему эффективная работа с памятью является одним из ключевых моментов при написании программ с использованием технологии CUDA?

12. Какие рекомендации необходимо соблюдать при написании эффективных CUDA-программ?

13. Как происходит компиляции CUDA-программ?

14. Что произойдет, если при объявлении функции указать два модификатора: __host__ и __device__?

15. Что означает термин “асинхронная” применительно к функциям из CUDA Runtime API?

16. Как можно измерить время выполения секции CUDA-программы?

17. Как можно синхронизировать выполнение потоков? Какие ограничения накладываются на синхронизируемые потоки?

ЗАКЛЮЧЕНИЕ

Мы рассмотрели несколько различных технологий создания параллельных программ. Некоторые из них специфичны для конкретной операционной системы, некоторые требуют наличия специального оборудования. Тем не менее большинство из них могут использоваться совместно в одной программе. Такое совместное использование технологий не всегда позволит повысить производительность системы, но в ряде случаев выигрыш от этого будет значительным. Наиболее перспективными с этой точки зрения выглядят гетерогенные вычисления с использованием видеоадаптеров. Совместимость технологий друг с другом приведена в табл. 9.1.

Таблица 9.1

Совместимость рассмотренных технологий

	OpenMP	IntelTBB	WinAPI Threads	POSIX Threads	MPI	CUDA
OpenMP		+	+	+	+	+
Intel TBB	+		+	+	+	+
WinAPI Threads	+	+		–	+	+
POSIX Threads	+	+	–		+	+
MPI	+	+	+	+		+
CUDA	+	+	+	+	+	

Несмотря на большое количество технологий, рассмотренных нами, довольная большая часть осталась вне рассмотрения, среди них можно упомянуть потоки в языке Java, библиотеку шаблонов Parallel Patterns Library от Microsoft и многие другие.

Мы довольно подробно рассмотрели алгоритмы параллельной сортировки и параллельные алгоритмы на графах, но многие алгоритмы, эффективные для других областей применения, остались без внимания.

Вне рассмотрения остались и средства анализа производительности параллельных программ, здесь можно было упомянуть Intel Parallel Studio. Вопросы “тонкой” настройки парал-

лельных программ очень актуальны и могут сильно влиять на эффективность параллельных вычислений.

Мир параллельных вычислений велик и его нельзя рассмотреть целиком в одной книге. Мы надеемся, что данный труд послужит отправной точкой для самостоятельного изучения параллельных вычислений и станет для студента своеобразным компасом, который поможет сориентироваться ему в этом разнообразном и удивительном мире.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Антонов А. С.* Параллельное программирование с использованием технологии OpenMP: учеб. пособие. – М.: Изд-во МГУ, 2009. – 77 с.
2. *Боресков А. В.* Основы работы с технологией CUDA / А. В. Боресков, А. А. Харламов – ДМК Пресс, 2010. – 234 с.
3. *Левин М.П.* Параллельное программирование с использованием OpenMP / М.П. Левин – Бином, 2012. – 121 с.
4. *Роберт Сэджвик.* Алгоритмы на C++ / Сэджвик Р. пер. с англ. – М.: ООО "И.Д. Вильямс", 2011. – 1056 с.
5. *Сандерс Д.* Технология CUDA в примерах / Д. Сандерс, Э. Кэндрот. – ДМК Пресс, 2011. – 232 с.
6. *Эндрюс Г. Р.* Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс – Вильямс, 2003. – 512 с.
7. *Chapman B.* Using OpenMP. A Parallel Programming Model for Shared Memory Architectures / B. Chapman, G. Jost, Ruud van der Pas. – The MIT Press, 2008. – 353 p.
8. *David R.* Programming with POSIX© Threads / David R. – Addison Wesley, 1997.
9. *Farber R.* CUDA Application Design and Development / Farber R. – Morgan Kaufmann, 2011. – 336 p.
10. *Gebali F.* Algorithms and Parallel Computing / Fayez Gebali. – Hoboken: John Wiley & Sons, 2011. – 341 p.
11. *Grama.* Introduction to Parallel Computing / A. Grama, G. Karypis, V. Kumar, A. Gupta – 2 ed., Addison Wesley, 2003. – 856 p.
12. *Hwu Wen-mei W.* GPU Computing Gems Emerald Edition / Hwu Wen-mei W. – Morgan Kaufmann, 2011. – 886 p.
13. *Hwu Wen-mei W.* GPU Computing Gems Jade Edition / Hwu Wen-mei W. – Elsevier Science, 2011. – 896 p.
14. *Pacheco P.* Parallel Programming with MPI / Pacheco P. – Morgan Kaufmann, 1997. – 418 p.
15. *Pharr Matt.* GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computa-

tion / Pharr Matt. – Addison-Wesley Professional, 2005. – 814 p.
16. *Quinn M.* Parallel Programming in C with MPI and OpenMP / M. Quinn, 2003.
17. *Reinders J.* Intel Threading Building Blocks / J. Reinders – O'Reilly Media, 2007. – 336 p.
18. *Snir C.* MPI - The Complete Reference, Volume 1 / C. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra – The MIT Press, 1998. – 426 p.
19. *Snir C.* MPI - The Complete Reference, Volume 2 / C. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra – The MIT Press, 1998. – 362 p.
20. *William G.* Using MPI / G. William, E. Lusk, A. Skjellum – The MIT Press, 1999. – 350 p.
21. *William G.* Using MPI-2 / G. William, E. Lusk, A. Skjellum – The MIT Press, 1999. – 350 p.

ОГЛАВЛЕНИЕ

Учебно-практическое издание

АЛЕКСЕЕВ Александр Георгиевич
ЙОВЕНКО Артем Романович

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Редактор *О. А. Хлебкова*
Компьютерная верстка и правка *А. Г. Алексеев*
Дизайн обложки *А. В. Мальцев*

Согласно Закону №436-ФЗ от 29 декабря 2010 года
данная продукция не подлежит маркировке

Подписано в печать 02.12.2015. Формат 60×84.16.
Бумага газетная. Печать офсетная. Гарнитура Times.
Усл. печ. л. 11,38. Уч.-изд. л. 11,17. Тираж 200 экз. Заказ № 1317.

Издательство Чувашского университета
Типография университета
428015 Чебоксары, Московский просп., 15