Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение высшего образования

Курганский государственный университет

Кафедра «Безопасность информационных и автоматизированных систем»

РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ В СТАНДАРТЕ OPENMP И ИССЛЕДОВАНИЕ УСКОРЕНИЯ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА НА МНОГОЯДЕРНОМ ПРОЦЕССОРЕ

Методические указания
по выполнению лабораторной работы для студентов
специальности (направления) 10.05.03 (10.03.01)

Курган 2019

Кафедра:	«Безопасность информационных и автоматизированных систем»
Дисциплина:	«Аппаратные средства вычислительной техники» по специальности (направлению) 10.05.03 (10.03.01)
Составил:	канд. техн. наук, доцент В.А. Стукало

Утверждены на заседании кафедры « 25 » сентября 2018 г.

Рекомендованы методическим советом университета « 20 » декабря 2017 г.

Лабораторная работа

Реализация параллельных алгоритмов в стандарте OpenMP и исследование ускорения параллельного алгоритма на многоядерном процессоре

Цели работы:
- ознакомление со стандартом программирования OpenMP;
- получение навыков параллельного программирования;
- подтверждение на практике закона Амдала;
- определение рационального числа потоков команд в параллельной программе.

1 Теоретические основы

Когда-то параллельное программирование было применимо только к суперкомпьютерам. В настоящее время в эпоху широкого распространения многоядерных процессоров параллельное программирование быстро стало технологией, которую должен уметь применять любой профессиональный разработчик программного обеспечения.

1.1 Модель использования памяти, процессоры, ядра, процессы и потоки (нити)

Основной характеристикой при классификации многопроцессорных систем является наличие общей (Symmetric Multiprocessor, SMP системы) или распределенной (Massive Parallel Processor, MPP системы) памяти. Это различие является важнейшим фактором, определяющим способы параллельного программирования и, соответственно, структуру программного обеспечения.

К системам с общей памятью относятся компьютеры с SMP архитектурой, различные разновидности NUMA (Non-Uniform Memory Access) систем и мультипроцессорные векторно-конвейерные компьютеры. Для этих компьютеров характерны: единая оперативная память, единая операционная система, единая подсистема ввода-вывода и множество (обычно до 32) равноправных процессоров, связанных общей шиной или коммутатором. На логическом уровне архитектуре симметричного мультипроцессора соответствует архитектура многоядерного процессора, которая реализуется в виде одновременной многопоточности (Simultaneous Multithreading, SMT) и собственно многоядерности (Multicore).

Технология одновременной многопоточности, позволяющая достичь многопроцессорности на логическом уровне, была предложена в 1995 году и позднее развита компанией Intel под названием технологии гиперпоточности (Hyper Threding, HT). Было замечено, что при выполнении большинства операций оказываются полностью задействованными не более 30% составных компонентов процессора. Для повышения загрузки процессора можно организовать последовательности команд (потоки), выполняющиеся параллельно и независимо друг от друга на аппаратном уровне за счет достаточно простого расширения возможностей процессора. В рамках такого подхода процессор дополняется программными счетчиками, контроллерами прерываний, схемами кон-

троля одновременного выполнения нескольких потоков и т.д. За счет этих дополнительных средств на активной стадии выполнения может находиться несколько потоков; при этом одновременно выполняемые потоки конкурируют за исполнительные блоки единственного процессора. Как правило, число аппаратно-поддерживаемых потоков равно 2, в более редких случаях этот показатель достигает 4 и даже 8. Аппаратно-поддерживаемые потоки на логическом уровне операционных систем Linux и Windows воспринимаются как отдельные процессоры. Например, единственный процессор с двумя аппаратно-поддерживаемыми потоками в менеджере Task Manager операционной системы Windows диагностируется как два отдельных процессора. На процессорах компании Intel с поддержкой технологии гиперпоточности достигается повышение скорости вычислений около 30% при надлежащей реализации программ. Выигрыш от использования одновременной многопоточности может достигаться, если на одном ядре выполняются потоки разнотипных приложений (например, просмотр почты и проигрывание музыки), если на одном ядре выполняется несколько потоков с интенсивным вводом-выводом и пр.

Большая вычислительная производительность может быть обеспечена за счёт реализации в единственном кремниевом кристалле нескольких вычислительных ядер в составе одного многоядерного процессора, при этом по своим вычислительным возможностям эти ядра не уступают обычным процессорам. Различия многоядерных процессоров могут состоять в количестве имеющихся ядер и в способах использования кэш-памяти ядрами процессора – кэш-память может быть как общей, так и распределенной для разных ядер. Например, кэш-память первого уровня может быть локальной для каждого ядра, а кэш-память всех последующих уровней и оперативная память − общей. Многоядерность позволяет повышать производительность процессоров при уменьшении энергопотребления, повышении надежности, снижении сложности логики процессоров и т.п. и стала одним из основных направлений развития компьютерной техники. В настоящее время для массового использования доступны двух-, четырех -, шести - и восьми - ядерные процессоры. В научно-технической литературе наряду с рассмотрением обычных многоядерных (multi-core) процессоров начато обсуждение процессоров с массовой многоядерностью (many-core), когда в составе процессоров будут находиться сотни и тысячи ядер!

Единая операционная система, управляющая работой всего компьютера, функционирует в виде множества процессов. Каждая пользовательская программа (задача) также запускается как отдельный процесс. Операционная система сама каким-то образом распределяет процессы по процессорам. С каждым процессом связывают его адресное пространство, из которого он может читать и в которое он может писать данные, и одиночный поток исполняемых команд. Адресное пространство содержит саму программу, данные к программе, стек программы. С каждым процессом связывается набор регистров: счетчик команд (регистр, в котором содержится адрес следующей, стоящей в очереди на выполнение команды. После выборки команды из памяти счетчик команд корректируется и указывает на следующую команду), указатель стека и др. В многоза-

дачной системе процессор переключается с процесса на процесс и исполняет их в квазипараллельном (псевдопараллельном) режиме. При этом в физический счетчик команд процессора загружается логический счетчик команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в памяти, в логическом счетчике команд процесса.

В принципе для распараллеливания программ можно использовать механизм порождения процессов. Однако этот механизм не очень удобен, поскольку каждый процесс функционирует в своем адресном пространстве, и основное достоинство этих систем − общая память − не может быть использован простым и естественным образом. Более выгодно (и до 100 раз скорее) создать квазипараллельный поток внутри процесса с одним адресным пространством.

Для распараллеливания программ чаще используется механизм порождения потоков (нитей (threads), легковесных процессов), для которых не создается отдельного адресного пространства, но которые в многопроцессорных системах также распределяются по процессорам. С каждым потоком связывают счетчик команд, регистры для текущих переменных, стек, состояние. Потоки делят между собой элементы своего процесса: адресное пространство, глобальные переменные, открытые файлы, таймеры, семафоры, статистическую информацию. В остальном модели потоков и модели процессов идентичны.

К особенностям реализации Windows относят четыре понятия: задание (набор процессов с общими квотами и лимитами), процесс (контейнер ресурсов (память...), содержит как минимум один поток), поток (именно исполняемая часть, планируемая ядром), волокно (облегченный поток, управляемый полностью в пространстве пользователя). Один поток может содержать несколько волокон. Потоки работают в режиме пользователя, но при системных вызовах переключаются в режим ядра. Из-за переключения в режим ядра и обратно сильно замедляется работа системы, поэтому было введено понятие волокна. У каждого потока может быть несколько волокон.

В языке программирования С возможно прямое использование механизма потоков для распараллеливания программ посредством вызова соответствующих системных функций, а в компиляторах с языка FORTRAN этот механизм используется либо для автоматического распараллеливания, либо в режиме задания распараллеливающих директив компилятору (такой подход поддерживают и компиляторы с языка С).

1.2 Стандарты параллельного программирования для вычислительных систем с общей памятью

В последние годы все более популярной становится система программирования OpenMP (Open specifications for Multi-Processing). Интерфейс OpenMP задуман как стандарт для программирования в модели общей памяти. В OpenMP входят спецификации набора директив компилятору, процедур и переменных среды. По сути дела, он реализует идею «инкрементального распараллеливания», позаимствованную из языка HPF (High Performance Fortran −

Fortran для высокопроизводительных вычислений). Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы. При этом система программирования OpenMP предоставляет разработчику большие возможности по контролю над поведением параллельного приложения. Вся программа разбивается на последовательные и параллельные области. Все последовательные области выполняет главная нить, порождаемая при запуске программы, а при входе в параллельную область главная нить порождает дополнительные нити. Предполагается, что OpenMP-программа без какой-либо модификации должна работать как на многопроцессорных системах, так и на однопроцессорных. В последнем случае директивы OpenMP просто игнорируются. Следует отметить, что наличие общей памяти не препятствует использованию технологий программирования, разработанных для систем с распределенной памятью. Многие производители SMP систем предоставляют также такие технологии программирования, как MPI и PVM. В этом случае в качестве коммуникационной среды выступает разделяемая память.

1.3 OpenMP: обзор

OpenMP – это API-интерфейс, который является отраслевым стандартом для создания параллельных приложений для компьютеров с совместным использованием памяти. Главная задача OpenMP — облегчить написание программ, ориентированных на циклы. Такие программы часто создаются для высокопроизводительных вычислений. OpenMP стал успешным языком параллельного программирования. Кроме того, в Intel был создан вариант OpenMP для поддержки кластеров. OpenMP поддерживает такой стиль программирования, при котором параллелизм добавляется постепенно, пока имеющаяся последовательная программа не превратится в параллельную.

OpenMP — постоянно развивается. Отраслевая группа под названием «Комиссия по проверке архитектуры OpenMP» проводит регулярные собрания, чтобы добавить к этому языку новые расширения. В выпуск Open MP версии 3.0 входит возможность организации очереди задач. Это позволяет OpenMP обрабатывать более широкий спектр структур управления и задействовать более общие рекурсивные алгоритмы.

OpenMP основывается на модели программирования «разветвление-объединение» (fork-join). Работа программы OpenMP начинается с одного потока. Когда программисту требуется добавить в программу параллелизм, выполняется разветвление на несколько потоков, чтобы создать группу потоков. Эти потоки выполняются параллельно в рамках фрагмента кода, который называется параллельным участком. В конце параллельного участка все потоки заканчивают свою работу и снова объединяются вместе. После этого исходный (или «главный») поток продолжает выполняться до тех пор, пока не начнется следующий параллельный участок (или не наступит конец программы).

Языковые конструкции в OpenMP определены как директивы компилятора, которые сообщают компилятору, что он должен делать, чтобы реализовать

требуемый параллелизм. В языках C и C++ такие директивы называются «прагмы».

Прагма OpenMP всегда имеет один и тот же вид:

#pragma omp construct_name one_or_more_clauses.

Имя_конструкции — это параллельное действие, которое требуется программисту, а операторы позволяют изменить это действие или управлять средой данных, которую наблюдают потоки.

OpenMP — это язык явного параллельного программирования. При создании потока или назначении этому потоку некоторой задачи программист должен указать нужное действие. Соответственно, даже в таком простом API-интерфейсе, как OpenMP, имеется широкий спектр конструкций и условий, которые программисту необходимо знать. Однако большую часть работы с OpenMP можно выполнить с помощью небольшого подмножества всего языка.

Для создания потока в OpenMP используется конструкция «parallel»:

#pragma omp parallel
{
…. A block of statements
}.

Если такая конструкция используется без уточняющих операторов, то число потоков, которые создает программа, определяется средой выполнения (обычно это число равно числу процессоров или ядер). Каждый поток будет выполнять блок инструкций, который следует за прагмой parallel. Это может быть почти любой набор разрешенных инструкций в C; единственным ограничением является запрет на переходы внутрь этого блока инструкций или из него. В OpenMP это общее ограничение. Такой блок инструкций без переходов называется «структурированный блоком».

Значительная часть параллельного программирования состоит именно в том, чтобы поручить всем потокам выполнение одних и тех же инструкций. Но чтобы использовать OpenMP в полной мере, потребуется разделить между потоками работу по выполнению набора инструкций. Такой тип поведения называется «совместное выполнение работы». Самая типичная конструкция для совместной работы — это конструкция цикла (в C это цикл for):

#pragma omp for.

Это работает только для простых циклов стандартного вида:

for(i=lower_limit; i<upper_limit; inc_exp).

Конструкция for распределяет итерации цикла между потоками группы, созданными ранее с помощью конструкции parallel. Начальное и конечное значения счетчика цикла, а также выражение для шага счетчика (inc_exp) должны быть полностью определены во время компиляции, а все константы, которые

используются в этих выражениях, должны быть одинаковы для всех потоков группы. Если вдуматься, это не лишено смысла. Система должна вычислить, сколько итераций цикла должно быть выполнено, чтобы разделить их на наборы, которые будут обрабатывать группы потоков. Это можно сделать только согласованно и точно, если все потоки используют одни и те же наборы счетчиков.

Необходимо отметить, что сама по себе конструкция for потоки не создает. Это можно сделать только с помощью конструкции parallel. Для простоты можно поместить конструкции parallel и for в одну и ту же прагму:

#pragma omp parallel for.

При этом будет создана группа потоков для выполнения итераций цикла, который следует непосредственно за прагмой.

Итерации цикла должны быть независимыми, чтобы результат выполнения цикла оставался неизменным независимо от того, в каком порядке и какими потоками выполняются эти итерации. Если один поток записывает переменную, которую затем считывает другой поток, то наблюдается кольцевая зависимость, и результат работы программы будет неверным. Программист должен тщательно проанализировать тело цикла, чтобы убедиться в отсутствии кольцевых зависимостей. В большинстве случаев такая зависимость возникает, если в переменную записываются промежуточные результаты, которые используются в данной итерации цикла. В этом случае приобретенной зависимости можно избежать, объявив, что каждый поток должен иметь собственное значение для этой переменной. Это можно сделать с помощью оператора private. Например, если в цикле используется переменная «tmp», в которой хранится временное значение, к конструкции OpenMP можно добавить следующий оператор, после чего переменную можно будет использовать в теле цикла, не создавая приобретенных зависимостей:

private(tmp).

Кроме того, часто встречается ситуация, когда переменная внутри цикла используется для сложения значений, полученных в каждой итерации. Например, это происходит в цикле, который суммирует результаты вычислений, чтобы получить одно итоговое значение. Такая ситуация часто возникает в параллельном программировании. Она называется «редукция». В OpenMP имеется оператор reduction:

reduction(+:sum).

Как и оператор private, он добавляется в конструкцию OpenMP, чтобы сообщить компилятору, что следует ожидать редукции. После этого создается временная закрытая переменная, которая используется для получения промежуточного результата операции суммирования для каждого потока. В конце выполнения конструкции значения этой переменной для каждого потока суммируются, чтобы получить конечный результат. Операция, которая использует-

ся при редукции, также указывается в операторе. В данном случае это операция «+». OpenMP определяет значение для закрытой переменной, которая используется в редукции, на основе соответствующей математической операции. Например, для «+» это значение равно нулю.

В OpenMP еще много интересных моментов, но и этих двух конструкций и операторов достаточно, чтобы объяснить, как распараллелить программу π.

1.4 Программа вычисления числа π: параллельное интегрирование по формуле прямоугольников

В параллельном программировании вычисление числа π является аналогом программы «Hello world» в обычном. Для расчета π используем формулу:

$$\pi = 4\int_0^1 dx/(1+x) \qquad (1.1)$$

Из курса математики известно, что интеграл можно представить геометрически в виде площади под кривой. В 1968 году математик Г. Александров предложил простой геометрический смысл числа π (рисунок 1).

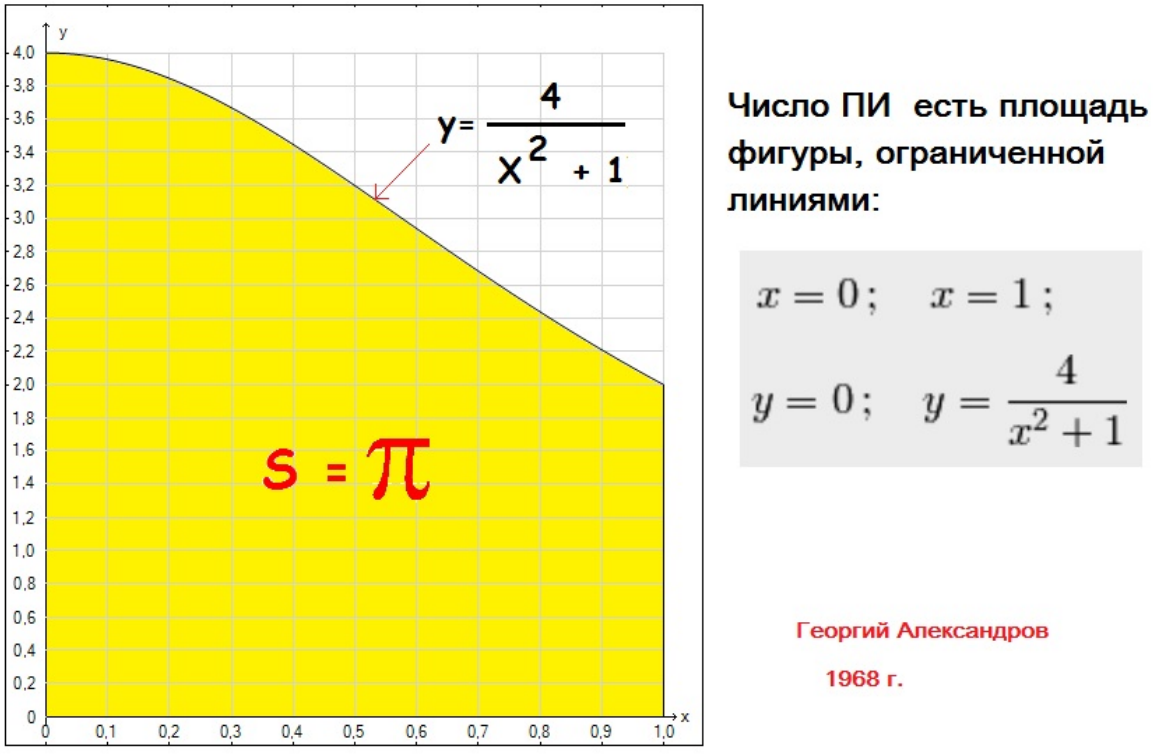

Рисунок 1 − Геометрический смысл числа π

Отсюда следует алгоритм приближенного вычисления значения интеграла. Участок интегрирования разбивается на большое число отрезков. Каждый из отрезков становится основанием прямоугольника, высота которого равна значению подынтегральной функции в центре данного отрезка. Приближенное значение интеграла равно сумме площадей всех прямоугольников.

Можно выбрать подынтегральную функцию и пределы интегрирования так, чтобы значение интеграла было равно π. Для получения точности 10^{-8} необходимо интервал разбить на 10^6 частей. Для начала приведем текст обычной последовательной программы и посмотрим, каким образом ее нужно модифицировать, чтобы получить параллельную версию.

```
static long num_steps = 100000;
double step;
void main ()
{
int i;
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
for (i=0; i<num_steps; i++){
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

1.5 Пример параллельного программирования

Чтобы не усложнять задачу, будем использовать только число потоков по умолчанию. В последовательной программе π имеется единственный цикл, который требуется распараллелить. Итерации цикла полностью независимы, если не считать значения зависимой переменной «x» и накопительной переменной «sum». Необходимо отметить, что «x» используется как временное хранилище для вычислений в итерации цикла. Соответственно, эту переменную необходимо сделать локальной для каждого потока с помощью оператора private:

private(x).

С технической точки зрения, управляющий счетчик цикла создает приобретенную зависимость. Впрочем, в OpenMP управляющий счетчик цикла автоматически становится локальным (закрытой переменной) для каждого потока.

Накопительная переменная «sum» используется для суммирования. Это классический пример редукции, поэтому можно применить оператор reduction:

reduction(+:sum).

После добавления этих операторов к конструкции «parallel for» получим программу π, распараллеленную с помощью OpenMP.

```
#include "omp.h"
static long num_steps = 100000; double step;
void main ()
{
int i;
```

```
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
for (i=0; i<num_steps; i++){
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

Необходимо отметить, что с помощью директивы include был также включен стандартный файл для OpenMP:

#include «omp.h».

Это позволяет определить типы и библиотечные подпрограммы времени выполнения, которые иногда требуются программисту OpenMP. Заметим, что в данной программе эти компоненты языка не использовались, но хороший стиль программирования предполагает добавление подключаемого файла OpenMP, хотя бы на тот случай, если он потребуется для будущего изменения программы.

1.6 Ускорение параллельного алгоритма

Ускорение параллельного алгоритма является его наиболее информативной характеристикой, которая показывает, во сколько раз применение параллельного алгоритма уменьшает время решения задачи по сравнению с последовательным алгоритмом. Ускорение параллельного алгоритма определяется величиной:

$$S_N = \frac{T_1}{T_N}, \qquad (1.2)$$

где T_1 − время выполнения алгоритма на одном процессоре,

T_N – время выполнения того же алгоритма на N процессорах.

Идеальным, очевидно, является ускорение S_N=N. В реальности это ускорение недостижимо.

В общем случае можно выделить следующие основные причины потери эффективности параллельных вычислений: время инициализации параллельной программы (startup); несбалансированность загрузки процессоров (load imbalance); затраты на коммуникации (communication costs); наличие в программе последовательных частей (serial part of the code). Основная причина недостижимости идеального ускорения хорошо иллюстрируется законом Амдала.

1.7 Закон Амдала (Amdahl)

Закон Амдала связывает ускорение параллельного алгоритма с долей последовательного кода.

$$S_N = \frac{1}{\alpha + (1-\alpha)/N}, \qquad (1.3)$$

где S_N – ускорение работы программы на N процессорах, а α – доля непараллельного кода в программе.

Эта формула справедлива при программировании и в модели общей памяти, и в модели передачи сообщений, только в понятие «доля непараллельного кода» вкладывается разный смысл. Для SMP систем (модель общей памяти) эту долю образуют те операторы, которые выполняются только главной нитью программы. Для MPP систем (механизм передачи сообщений) непараллельная часть кода образуется за счет операторов, выполнение которых дублируется всеми процессорами. Оценить эту величину из анализа текста программы практически невозможно. Такую оценку могут дать только реальные просчеты на различном числе процессоров. Из формулы (1.3) следует, что N-кратное ускорение может быть достигнуто, только когда доля непараллельного кода равна 0. Очевидно, что добиться этого практически невозможно. Из закона Амдала следует, что если, к примеру, α =0,1, т.е. всего 10% операций программы выполняется последовательно, то при любом, как угодно большом количестве используемых процессоров N, ускорение, превышающее 10, принципиально получить невозможно (таблица 1).

Таблица 1 − Предельное ускорение как функция доли последовательных операций α в процентах и количества процессоров N

Количество процессоров N	α =50%	α =25%	α =10%	α =5%	α =2%
2	1.33	1.60	1.82	1.90	1.96
8	1.78	2.91	4.71	5.93	7.02
32	1.94	3.66	7.80	12.55	19.75
51	1.99	3.97	9.83	19.28	45.63
2048	2.00	3.99	9.96	19.82	48.83

В некотором смысле закон Амдала устанавливает предельное число процессоров, на котором программа будет выполняться с приемлемой эффективностью в зависимости от доли непараллельного кода.

2 Постановка задачи к лабораторной работе

При выполнении лабораторной работы следует реализовать следующее:

1) изучить стандарт программирования OpenMP;

2) в стандарте OpenMP составить параллельную программу вычисления числа π в соответствии с вариантом индивидуального задания (таблица 2), измеряющую время расчета. Точность расчета для разных методов (формул) определяется длительностью расчета, которая не должна быть менее нескольких десятков секунд.

3) провести исследование зависимости времени решения задачи от числа используемых ядер и потоков команд, подключая последовательно 1, 2, 3, 4, … ядра, и для каждой конфигурации ядер назначать последовательно 1, 2, 3, 4, … потока команд, повторяя эксперименты по три раза, вычисляя среднее арифметическое. Результаты измерений занести в таблицы, построить графики ускорений. Для отключения ядер процессора и наблюдения за их графиками загрузки использовать «Диспетчер задач» Windows. Графики загрузки ядер также разместить в отчете. Определить подключение логических и физических ядер по показаниям времени решения задачи при использовании процессора с функцией одновременной многопоточности;

4) по результатам экспериментов определить по формуле Амдала долю последовательных операций кода и теоретически достижимое ускорение алгоритма.

5) дать объяснение полученным результатам, сделать вывод о подтверждении или неподтверждении закона Амдала и рациональном соотношении количества потоков команд и ядер при выполнении программы.

Варианты индивидуальных заданий представлены в таблице 2.

Таблица 2 – Индивидуальные задания

№ поз.	Формула
1	Виет $\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \cdot \ldots$
2	Валлис $\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \ldots = \frac{\pi}{2}$
3	Лейбниц π = (4/1) - (4/3) + (4/5) - (4/7) + (4/9) - (4/11) + (4/13) - (4/15) .
4-6	$\pi = \frac{1}{2}\sum_{k=0}^{\infty}\frac{1}{16^k}\left(\frac{8}{8k+2}+\frac{4}{8k+3}+\frac{4}{8k+4}-\frac{1}{8k+7}\right) =$ $= \frac{1}{4}\sum_{k=0}^{\infty}\frac{1}{16^k}\left(\frac{8}{8k+1}+\frac{8}{8k+2}+\frac{4}{8k+3}-\frac{2}{8k+5}-\frac{2}{8k+6}-\frac{1}{8k+7}\right) =$ $= \sum_{k=0}^{\infty}\frac{(-1)^k}{4^k}\left(\frac{2}{4k+1}+\frac{2}{4k+2}+\frac{1}{4k+3}\right)$
7	Плафф π=∑k=0∞ (4/(8k+1)−2/(8k+4)−1/(8k+5)−1/(8k+6))/ 16**k

Продолжение таблицы 2

8	Монте Карло $\frac{Area\ of\ Cirle}{Area\ of\ Square} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$
9	Мэчин π/4=4arctg(1/5)−arctg(1/239) arctg x=x−x**3/3+x**5/5−x**7/7+···
10	Нилакант (π = 3 + 4/(2*3*4) - 4/(4*5*6) + 4/(6*7*8) - 4/(8*9*10) + +4/(10*11*12) - 4/(12*13*14)...)
11	Рамануджан 1/π=2√2/9801∑k=0∞(4k)!(1103+26390k)/(k!)**4*396**4k
12	Чудновски 426880√10005/π=∑k=0∞(6k)!(13591409+545140134k)/(3k)!(k!)**3(−640320)**3k
13	$\pi = 2\sqrt{3}\sum_{k=0}^{\infty}\frac{(-1)^k}{3^k\,(2k+1)}$

Список литературы

1 OpenMP C/C++ API. Октябрь 1998. (PDF, 201K)

2 http://www.openmp.org/

3 Таненбаум Э., Остин Т. Архитектура компьютера. 6-е изд. − Санкт-Петербург : Издательство: «Питер», 2013, 816 с.

4 http://baumanki.net/lectures/10-informatika-i-programmirovanie/330-lekcii-po-ossio/

5 https://studfiles.net/preview/2874724/

6 http://it.kgsu.ru/ParalAlg/palg036.html

7 http://parallel.ru/tech/tech_dev/openmp.html

Стукало Виктор Александрович

РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ В СТАНДАРТЕ OPENMP И ИССЛЕДОВАНИЕ УСКОРЕНИЯ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА НА МНОГОЯДЕРНОМ ПРОЦЕССОРЕ

Методические указания
по выполнению лабораторной работы для студентов
специальности (направления) 10.05.03 (10.03.01)

Редактор Н.М. Быкова

Подписано к печати12.03	Формат 60×84 1/16	Бумага 65 г/м^2
Печать цифровая	Усл. печ. л. 1,0	Уч.- изд. л. 1,0
Заказ 52	Тираж 25	Не для продажи

Библиотечно-издательский центр КГУ.
640020, г. Курган, ул. Советская, 63/4.
Курганский государственный университет