**Лабораторная работа 2. Технология OpenMP**

Основным препятствием на пути решения задачи автомати­ческого распараллеливания является сложность выявления частей последовательной программы, которые могли бы выполняться одновременно. Выявление таких частей представляет определен­ные трудности даже для самого разработчика параллельной про­граммы. Эти трудности многократно возрастают, когда такая за­дача решается транслятором в автоматическом режиме. Вследст­вие этого получил распространение подход, основанный на расширении традиционных языков программирования конструк­циями, облегчающими автоматическое выявление параллельных операторов и функций в программе. Таких расширений было предложено достаточно много.

Необходимость унификации средств разработки параллельных программ привела к созданию в 1997 году единого стандарта, получившего название ОрепМР.

Основные компоненты ОрепМР

С точки зрения прикладного программиста в состав среды ОрепМР входят три основных компонента: набор директив транслятору, набор функций библиотеки и переменные окру­жения.

Директивы служат основным средством выражения паралле­лизма в ОрепМР.

Директивы представляют собой дирек­тивы компилятору — «прагмы» (в случае C/C++).

Такой подход удобен тем, что эти директивы игнорируются обычным трансля­тором, не поддерживающим ОрепМР. В результате программу можно компилировать и выполнять в последовательном вариан­те, что существенно облегчает процесс поиска ошибок.

При этом следует принимать во внимание, что в параллельном варианте могут появиться ошибки, которые стали результатом некоррект­но проведенного распараллеливания.

Функции библиотеки позволяют управлять различными ха­рактеристиками в процессе выполнения OpenMP-программы, на­пример числом потоков и т.п. С помощью переменных среды пользователь имеет возможность управлять настройками среды выполнения, например, устанавливать характерные параметры распараллеливания циклов. Использование переменных среды позволяет влиять на поведение параллельной программы без пе­рекомпиляции.

Модель выполнения ОрепМР-программы

Модель выполнения задает эталонное поведение программы, определяя, каким образом она должна выполняться и какой на­блюдаемый эффект будет при этом произведен. Тем самым мо­дель задает рамки, в пределах которых конкретная реализация может выбрать наиболее эффективный для данной платформы способ выполнения программы.

В ОреnМР принята достаточно традиционная модель многопоточности (рис. 1). Вначале программа всегда состоит из одного потока, называемого глав­ным (master-thread), и выполняется в обычном последовательном режиме. Если в некоторый момент в программе выполняется ди­ректива распараллеливания, то к главному потоку добавляется еще несколько, образуя группу параллельно выполняющихся по­токов. Потоки выполняют некоторый фрагмент программы, после чего параллельный участок заканчивается, и выполнение вновь продолжает один поток.

Таким образом, процесс выполнения ОреnМР-программы состоит в чередовании последовательных и параллельных участков.

Параллельные участки могут быть вложенными. Если ОреnМР функционирует в режиме с разрешенным вложенным параллелизмом, то каждый из выполняющихся потоков может создавать новые потоки при выполнении нового параллельного участка. Параллельный участок всегда завершается барьерной синхронизацией всех выполняющих его потоков (она может быть и неявной).

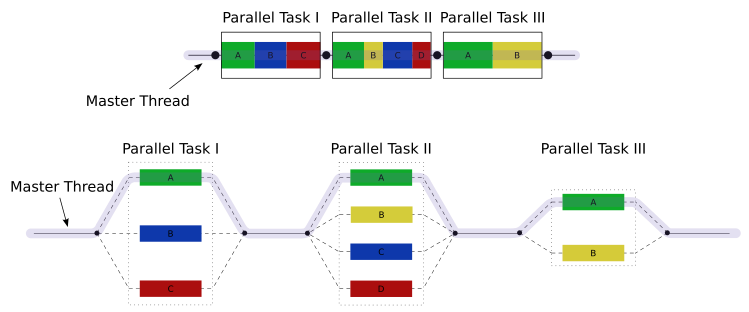


Рисунок 1.

Модель памяти ОрепМР-программы

Переменные в OpenMP-программе делятся на общие (shared) и индивидуальные (private).

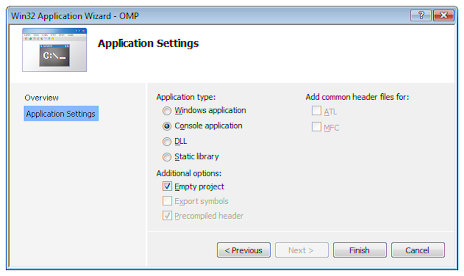
Индивидуальные переменные соот­ветствуют некоторому потоку и могут считываться или записы­ваться только им. Общие переменные доступны для чтения и за­писи нескольким потокам одной группы. Следует отметить, что если чтение и запись или повторная запись общей переменной производятся без синхронизации, то результирующее значение переменной считается неопределенным.

В ОрепМР отсутствует строгая корреляция между классом памяти переменной (статическая, ав­томатическая и т.п.) и тем, как она рассматривается различными потоками: автоматическая переменная может рассматриваться как общая на некотором параллельном участке, а статическая пе­ременная может быть «приватизирована» каждым потоком.

Простейший пример программы на ОреnМР

Пре­жде все­го, нуж­но за­пу­стить Visual Studio, и вы­брать File →​ New → ​Project… По­явит­ся ок­но со­зда­ния про­ек­та. Вы­бе­ри­те тип про­ек­та «Win32», шаб­лон — «Win32 Console Application». Вве­ди­те имя про­ек­та, вы­бе­ри­те пап­ку для хра­не­ния про­ек­та.

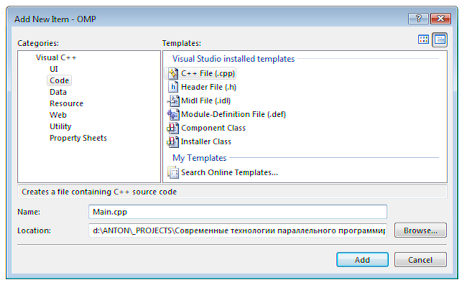
На­жми­те кноп­ку «OK», по­явит­ся ок­но на­строй­ки бу­ду­ще­го про­ек­та. Вы­бе­ри­те вклад­ку «Application Settings», и вклю­чи­те указатель «Empty project» (это не обязательно, по-умолчанию среда разработки создаст проект, состоящий из одного файла main.cpp):



Ри­су­нок 2. Ок­но на­строй­ки бу­ду­ще­го про­ек­та

По на­жа­тию кноп­ки «Finish» про­ект бу­дет со­здан.

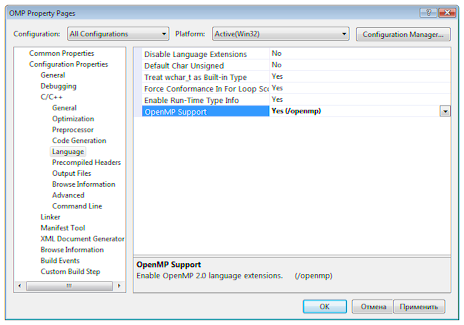
Те­перь на­жми­те Project → Add New Item, по­явит­ся ок­но до­бав­ле­ния эле­мен­тов в про­ект. До­бавь­те .cpp-файл в про­ект:



Ри­су­нок 3. Ок­но до­бав­ле­ния эле­мен­тов в про­ект

По­сле это­го вам бу­дет предо­став­ле­но ок­но для вво­да ис­ход­но­го ко­да про­грам­мы.

Для вклю­че­ния OpenMP на­жми­те Project → OMP Properties (OMP — имя про­ек­та из мо­их при­ме­ров). Сле­ва ввер­ху по­явив­ше­го­ся ок­на вы­бе­ри­те «All Configurations» и в раз­де­ле Configuration Properties → C/C++ → Language вклю­чи­те «OpenMP Support»:



Ри­су­нок 4. Вклю­ча­ем OpenMP в свой­ствах про­ек­та

По­сле это­го сно­ва за­пу­сти­те про­грам­му, на­жав Debug → Start Without Debugging.

Следует отметить, что при проблемах на этапе линковки приложения (ошибки, связанные с COFF) в первую очередь рекомендуется проверить наличие необходимых компонентов среды разработки (поддержки C++), а также отключить инкрементальную компоновку (см. Проект->Свойства <проект>->Свойства конфигурации->Компоновщик->Общие->Включить инкрементальную компоновку).

Рассмотрим простейшую программу на ОреnМР. Программа начинается с подключения необходимых библиотек. Для того чтобы получить возможность использовать функции и макросы ОреnМР, необходимо подключить заголовочный файл omp.h (строка 2). В данной программе содержится всего одна директива ОреnМР, расположенная в строке 5. Это — директива parallel, обозначающая параллельный участок.

#include <stdio.h>

#include <omp.h>

omp\_set\_num\_threads( 2 );

int main(){

#pragma omp parallel

printf(“Hello World!\n”);

}

Рассмотрим подробнее синтаксис директивы ОрепМР. В языках C/C++ для выражения конструкций ОрепМР применяется механизм директив компилятору, начинающихся с ключевого слова #pragma. Чтобы директивы ОрепМР можно было отличить от прочих директив, после слова pragma следует ключевое слово оmp. Далее следует имя директивы и возможные опции, если они предусмотрены.

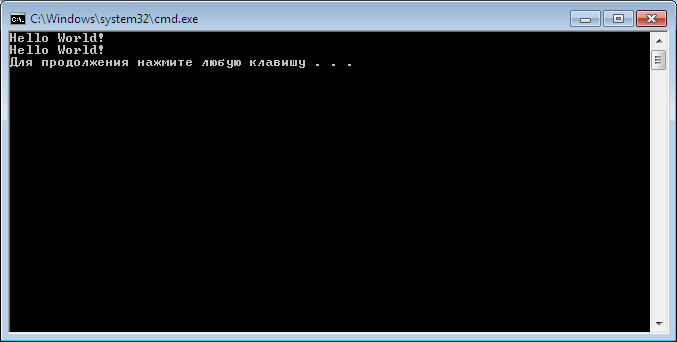
Директива parallel, как и большинство других директив ОрепМР, применяется непосредственно к оператору, следующе­му за ней. В частности, такой оператор может быть составным, т.е. представлять последовательность операторов, заключенную в фигурные скобки. Такой подход позволяет применять директиву к произвольным последовательным участкам кода программы. Оператор, к которому применяется директива, должен иметь од­ну точку входа и одну точку выхода из него. Такой оператор при­нято называть структурным блоком

#pragma omp parallel

В рассматриваемом примере директива применяется к опера­тору, состоящему из единственного выражения — вызова функ­ции printf. Действие директивы parallel состоит в создании параллельного участка, в результате чего оператор, к которому применяется это директива, будет выполнен несколькими соз­данными потоками.

Процедура omp\_set\_num\_threads(2); говорит об использовании 2 потоков

В окне выполнения увидим:



Следующий пример демонстрирует использова­ние информационных функций. Директива задает параллельный участок, который будет выполняться груп­пой потоков. Функция печати распечатывает текст при­ветствия, номер каждого потока и число потоков в группе.

int main() {

omp\_set\_num\_threads(4);

int n=0;

n = omp\_get\_num\_threads();

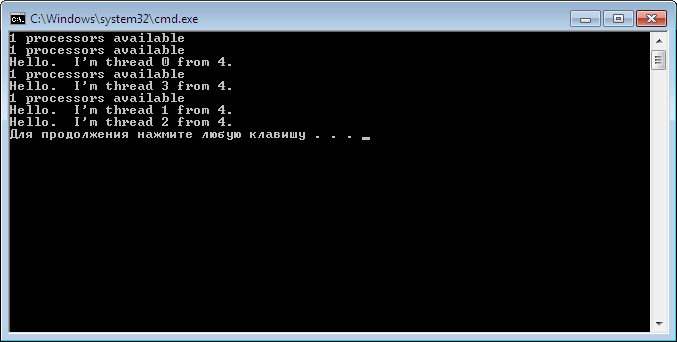
#pragma omp parallel num\_threads(4)

printf(“Hello. I’m thread %d from %d.\n”, omp\_get\_thread\_num(),

omp\_get\_num\_threads() );

}

Результат выполнения:



Опции для переменных в OpenMP-программе

В предыдущем разделе рассмотрены базовые возможности ОрепМР, с помощью которых можно инициировать параллель­ный участок в программе. Теперь рассмотрим средства зада­ния дисциплины работы с данными в ОрепМР-программе.

Переменные в ОрепМР могут быть либо общими для группы потоков, либо индивидуальными для каждого потока. Принад­лежность переменной к тому или иному типу определяется либо с помощью явного указания, либо правилами по умолчанию.

Правила по умолчанию применяются к переменным, которые не являются аргументами опций каких-либо директив. Согласно этим правилам:

* любая переменная, объявленная вне блока параллельного выполнения, будет общей для потоков, выполняющих этот блок
* общими также являются статические переменные;
* любая автоматическая переменная, объявленная внутри блока параллельного выполнения, будет индивидуальной для потоков, выполняющих этот блок;
* локальные переменные и формальные параметры функ­ций, вызываемых внутри блока параллельного выполне­ния, также будут индивидуальными.

Проиллюстрируем эти правила на следующем примере:

void f (int c) {

static double z;

int x;

}

int main() {

double y;

#pragma omp parallel

{

int a;

f(5);

}

}

По отношению к параллельному участку кода z и y – общие переменные, a,c,x – индивидуальные переменные.

Правила по умолчанию можно изменить с помощью специ­альных опций директивы parallel. Аргументом опции явля­ется список идентификаторов переменных, разделенных запя­той. В список аргументов могут входить только переменные, принадлежащие области видимости, включающей параллельный участок. Другим словами, они должны быть определены до начала параллельного участка. Рассмотрим наиболее употреби­тельные опции.

Опция **private** определяет список переменных, которые будут индивидуальными для потоков, выполняющих параллель­ный участок. При этом не задается, каким именно образом произ­водится инициализация значения переменных на потоках. Также неопределенным будет значение переменной на главном потоке после завершения параллельного участка.

Опция **firstprivate** задает способ инициализации инди­видуальных переменных: переменные, перечисленные в списке аргументов этой области, получают значение, равное значению переменной на главном потоке в момент входа в параллельный участок. Таким образом, опция firstprivate предоставляет всю функциональность опции private, добавляя к ней способ инициализации переменных.

Опция **reduction** определяет значение переменных, вхо­дящих в список ее аргументов, на главном потоке после заверше­ния параллельного участка как результат выполнения редуктивной операции. На каждом из потоков, выполняющих параллель­ный участок, переменная получает значение, соответствующее редуктивной операции:

Опция редукции значений переменных на потоках: reduction(operator:list)

list — список идентификаторов

operator – одна из следующих редуктивных операций:

|  |  |
| --- | --- |
| Операция | Значение  для инициализации |
| + | 0 |
| \* | 1 |
| - | 0 |
| & | ~0 |
| | | 0 |
| ^ | 0 |
| && | 1 |
| || | 0 |

После завершения параллельного участка значение перемен­ной на главном потоке меняется в результате применения к нему и значениям переменных на всех потоках указанной операции. Порядок применения операции определяется реализацией.

Более формально это можно выразить следующим образом. Пусть некоторая переменная а входит в список аргументов опции reduction с операцией ор. Пусть параллельный участок вы­полнялся п потоками и до него переменная имела значение v. Если в конце выполнения параллельного участка локальные ко­пии переменной а имели значения vb v„, то после параллель­ного участка переменная а на главном потоке получит значение, равное (v op V1| op V2 op … op v„).

Следующий пример иллюстрирует применение опции reduction. Программа, представленная на листинге, вы­числяет произведение целых чисел, переданных через аргументы командной строки. В строках 6, 7 производится разбор аргумен­тов командной строки: сомножители сохраняются в переменных а и Ь.

Директива parallel в строке 9 имеет три опции: firstprivate, reduction и num\_threads. Опция firstprivate (а) означает, что переменная а будет индиви­дуальной и инициализируется на всех потоках значением, взятым с главного. Опция reduction (+ : t) означает, что после вы­хода из параллельного участка переменная на главном потоке t будет увеличена на величину суммы значений этой переменной, подсчитанных на каждом из потоков. В данном случае это озна­чает, что переменная t получит значение, равное сумме значения ее локальных копий, так как до входа в участок параллельного выполнения переменная имела значение 0. Последняя опция num\_threads (b) задает число потоков, равное второму со­множителю b. В теле параллельного участка присутствует един­ственный оператор присваивания t = а, в результате которого индивидуальные переменные t получают одинаковые значения, равные первому сомножителю. Общее число потоков, таким об­разом, равняется величине второго сомножителя, а на каждом из них переменная t имеет значение, равное первому сомножителю. В результате редукции, после выхода из параллельного участка переменная t получит значение, равное произведению а и Ь. В строке 14 это значение выводится на печать.

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main(int argc, char\* argv[]) {

int a, b, t;

a = atoi(argv[1]);

b = atoi(argv[2]);

printf(“a b = %d %d\n”, a,b) ;

t = 0;

#pragma omp parallel reduction (+ : t) num\_threads (b)

{

printf(“a b = %d %d\n”, a,b) ;

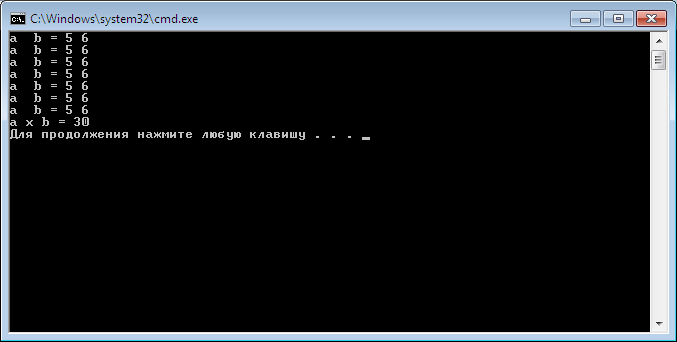
t= a;

}

printf(“a x b = %d\n”, t) ;

}

Результат



Синхронизация в OpenMP

Как и в любой среде многопоточного программирования, в OpenMP важную роль играет синхронизация. Синхронизация не­обходима, если различные потоки работают с общими данными. Если чтение и запись или повторная запись общей переменной производятся без синхронизации, то результирующее значение переменной считается неопределенным.

Самым простым способом защитить данные от возможных проблем, связанных с одновременным доступом, является дирек­тива atomic. Эта директива применяется к оператору-выражению одного из следующих типов:

х ор= ехрг; х ++; ++ х; х —; — х;

где х — переменная, ехрг — выражение, не ссылающееся на х, ор — бинарная операция. Данная директива обеспечивает ато­марность соответствующей операции: в момент выполнения этой операции одним из потоков другие потоки не имеют доступа к переменной х.

Другим базовым механизмом синхронизации является меха­низм критических секций. Критическая секция в коде программы выделяется с помощью директивы critical, которая имеет следующий синтаксис:

#pragma omp critical [name]

Опция name является необязательной; если она отсутствует, то считается, что директива имеет зарезервированное имя. Таким образом, каждой критической секции ставится в соответствие некоторое имя. Синхронизация обеспе­чивается следующим образом: критические секции с одинаковы­ми именами не могут выполняться одновременно. В случае возникновения ситуации конкуренции, они выполняются последовательно.

Это свойство может быть использовано для объединения результатов выполнения различных потоков:

long unsigned int sum = 0;

long unsigned psum = 0;

#pragma omp parallel firstprivate(psum)

{

psum = 0;

#pragma omp for

for (int i = 0; i < N; i++) {

psum += a[i] \* a[i];

}

#pragma omp critical

{

sum += psum;

}

}

Здесь параллельный цикл for вычисляет частичные суммы psum. При завершении каждой из нитей (на которые разбит цикл for), выполняется критическая секция. В критической секции частичная сумма добавляется к общей сумме. Ввиду того что код критической секции выполняется последовательно, гарантируется корректное потокобезопасное выполнение данной процедуры: в каждый момент времени выполняется лишь один экземпляр этого блока.

В некоторых случаях требуется ограничить набор потоков, выполняющих некоторый фрагмент кода. Это достигается при помощи директивы master, имеющей синтаксис:

#pragma omp master

Оператор, к которому применяется данная директива, вы­полняется только главным потоком.

Распределение работы между параллельными потоками

Директива parallel позволяет инициировать параллельное выполнение участка программы группой потоков. Используя ин­формацию о номере потока, которая доступна через функцию omp\_get\_thread\_num, и механизмы синхронизации, можно разрабатывать достаточно сложные параллельные программы. При этом процесс разработки остается достаточно сложным и низко­уровневым. Однако, ОрепМР предоставляет механизмы автоматизации распре­деления работы по потокам, которые рассматри­ваются далее.

Последовательный участок внутри параллельного — директива single

Самой простой директивой распределения работы в ОрепМР является директива single. Эта директива выделяет оператор, выполняемый только одним потоком из группы (не обязательно главным). Таким образом, внутри параллельного участка появля­ется фрагмент кода, выполняемый в последовательном режиме.

Информационные зависимости в программе

Одной из основных причин, которая не позволяет провести эффективное распараллеливание, является наличие информации­онных зависимостей между операторами программы. Информа­ционные зависимости имеют место в случае, когда один из опе­раторов использует результаты работы другого оператора. На­пример, считывает значения переменной, измененной другим оператором.

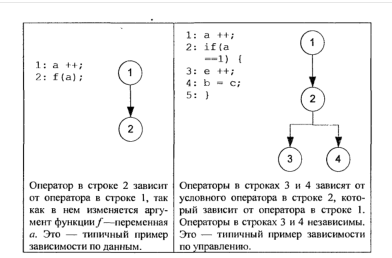
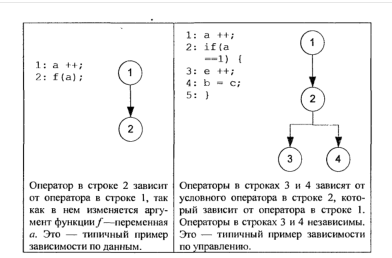
Будем говорить, что два оператора имеют зависимость по данным, если

1) оба обращаются к общему участку памяти, и

2) хотя бы одно из этих обращений является записью.

Помимо зависимости по данным, существует также зависимость по управлению — когда один из операторов является оператором, влияющим на поток управления, а выполнение второго зависит от выполнения первого.

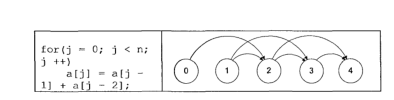
Программе можно сопоставить ориентированный граф, если поставить в соответствие операторам его вершины, а дугами обо­значить информационные зависимости. Примеры фрагментов кода программ и соответствующие им зависимости приводятся на рис.



Информационные зависимости необходимо учитывать при распараллеливании. Очевидно следующее утверждение: если ме­жду операторами нет информационной зависимости, то их можно выполнять одновременно. В противном случае может быть нару­шена семантика программы.

Важным частным случаем информационной зависимости яв­ляется зависимость между итерациями цикла. Две итерации цик­ла зависят друг от друга, если 1) они обращаются к общим дан­ным и 2) хотя бы одно из этих обращений является записью. Пример цикла с зависимостями между итерациями приведен на рис. Для пяти итераций построен граф информационных зави­симостей: итерация j зависит от итераций

j-1 и j-2. Итерации цикла, между которыми существует информацион­ная зависимость, необходимо выполнять в порядке, определяе­мом этой зависимостью. В частности, зависимые итерации не мо­гут выполняться одновременно.



Цикл с информационными зависимостями между итерациями

Стандарт ОрепМР не требует от компилятора проверки зави­симости по данным. Поэтому в большинстве существующих реа­лизаций такая проверка отсутствует. Это означает, что ответст­венность за то, что при распределении работы между параллель­ными потоками зависимость по данным не нарушается, лежит на разработчике.

Разбиение на независимые блоки — директива sections

В ряде случаев в программе удается выделить достаточно крупные фрагменты, которые можно выполнять параллельно. Директива sections позволяет организовать такое выполнение.

Директива применяется к составному оператору, т.е. к последо­вательности операторов, заключенных в фигурные скобки. Тело со­ставного оператора должно быть представлено в виде последова­тельности операторов (секций), каждому из которых, за исключени­ем, быть может, первого, предшествует директива section, имеющая следующий синтаксис:

*#pragma omp sections {*

*#pragma omp section*

*Statement*

*#pragma omp section*

*statement2*

*}*

Структурный блок, к которому применяется директива sec­tions, должен выполняться как часть параллельного участка.

При этом каждая из секций этого блока выполняется только один раз одним из потоков, входящих в группу.

При применении данной директивы необходимо убедиться, что разные секции не содержат операторов, между которыми существу­ет информационная зависимость. Если этого не сделать, в результа­те распараллеливания программа может стать некорректной.

Выполнение директивы sections должно происходить внут­ри параллельного участка. Предусмотрена сокращенная директива parallel sections, позволяющая комбинировать инициализа­цию параллельного участка и распределение работы по секциям.

Распараллеливание циклов

Большая часть времени работы приложений приходится на циклические участки. Поэтому ускорению работы таких участков с помощью распараллеливания уделяется большое внимание.

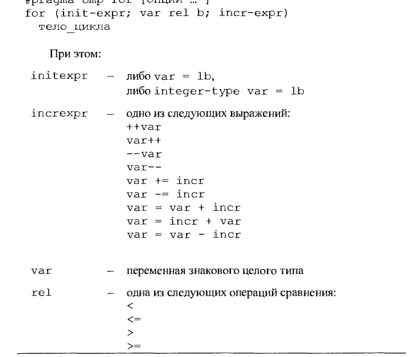
Значительная часть работ по автоматическому распараллели­ванию программ посвящена именно циклам.

Несмотря на значительный прогресс, достигнутый в области автоматического распараллеливания циклов, эффективность параллельного кода, генерируемого современными промышленными компиляторами, невысока. Это связано с тем, что задача автоматического выявле­ния и анализа зависимостей по данным между итерациями цикла в общем случае является чрезвычайно сложной. Кроме того, рас­параллеливанию целесообразно подвергать только циклы, вы­полнение которых вносит существенный вклад в работу про­граммы. Автоматическое выявление таких циклов также является трудновыполнимой задачей.

Для преодоления этой проблемы в ОреnМР принят следую­щий подход: разработчик параллельного приложения самостоя­тельно выбирает циклические участки, которые надо распаралле­ливать. При необходимости имеющиеся циклы модифицируются с целью устранения информационных зависимостей. Выбранные циклы помечаются специальной директивой, в которой указыва­ются некоторые параметры распределения работы. В процессе трансляции компилятор ОреnМР генерирует код, который рас­пределяет итерации цикла по потокам. При этом предполагается, что разработчик убедился в отсутствии информационных зави­симостей между итерациями.

Для обозначения распараллеливаемых циклов применяется директива for. В программе эта директива предшествует распа­раллеливаемому циклу типа for. При этом цикл должен удовле­творять дополнительным условиям, выполнение которых требу­ется для того, чтобы компилятор смог эффективно произвести анализ и сформировать качественный параллельный код:

#pragma omp for [опции ... ] for (init-expr; var rel b; incr-expr) тело\_цикла



Важно отметить, что переменная-итератор цикла автомати­чески становится индивидуальной для всех потоков. Стандарт OpenMP не допускает изменение ее значения в теле цикла.

Директива for должна выполняться внутри параллельного участка. Предусмотрена также директива parallel for, пре­доставляющая возможность совмещения функциональности директив parallel и for в одной директиве. В результате выполнения этой директивы создается параллельный участок вы­полнения и итерации цикла распределяются между выполняю­щими его потоками.

В качестве примера применения директивы for рассмотрим программу вычисления поэлементной суммы двух векторов. Сначала рассмотрим последовательную функцию вычисления суммы двух векторов. Функция принимает следую­щие входные параметры:

п – размерность вектора;

а, b — суммируемые вектора;

с — результат.

Функция в цикле (строки 4-5) вычисляет поэлементную сум­му двух векторов. Последовательный вариант

void vsum(int n, int\* a, int\* b,int\* c) {

int I;

for(I =0; I < n; I ++)

c[i] = a[i] + b[i];

}

Итерации цикла в строках 4—5 не имеют информационных зависимостей, так как на каждой следующей итерации обрабаты­ваются новые элементы массивов а, Ь, с. Поэтому к нему можно применить директиву for.

Параллельный вариант

void vsump(int n, int\* a, int\* b,int\* c)

{

int I;

#pragma omp for

for(I =0; I < n; I ++) c[i] = a[i] + b[i];

}

Рассмотрим теперь функцию main, которая вызывает обе функции, измеряет и сравнивает времена их работы (листинг 40). Размерность Параллельный вариант функции суммирования векторов вы­зывается в цикле, который работает внутри параллельного участ­ка.

int main(int argc, char\* argv[])

{

int n, iters, t, I;;

int\* a, \*b, \*c;

n = atoi(argv[1]) ;

iters = atoi(argv[2]);

/\*a = (double\*)malloc(n \* sizeof(double));

b = (double\*)malloc(n \* sizeof(double));

c = (double\*)malloc(n \* sizeof(double));\*/

a= new int[n];

b= new int[n];

c= new int[n];

//randomize();

{for (i=0;i<n;i++)

a[i]=(int)rand()\*100;

for (i=0;i<n;i++)

b[i]=(int)rand()\*100;

t = time(NULL);

for(I =0; I < iters; I ++) {

vsum(n, a, b, c);

for (i=0;i<n;i++)

printf (“a[%d] %d “,I,a[i]);

printf(“\n”);

for (i=0;i<n;i++)

printf (“b[%d] %d “,I,b[i]);

printf(“\n”);

for (i=0;i<n;i++)

printf (“c[%d] %d “,I,c[i]);

printf(«\n»);

}

}

t = time(NULL) - t;

printf(“Sequential: %d iterations consumed %d seconds\n”, iters, t) ;

t = time(NULL);

#pragma omp parallel firstprivate(n)

{

int I,j;

for (j =0; j < iters; j ++) {

vsump(n, a, b, c);

for (i=0;i<n;i++)

printf (“c[%d] %d “,I,c[i]);

printf(“\n”);

}

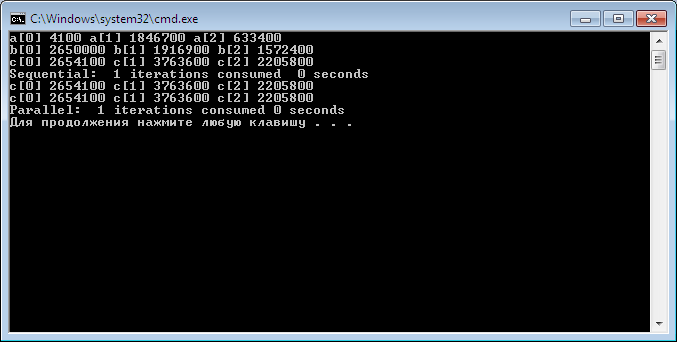
}

t = time(NULL) - t;

printf(“Parallel: %d iterations consumed %d seconds\n”, iters, t) ;

}

Результаты:



int main(int argc, char\* argv[])

{

int n, iters, t, I;;

int\* a, \*b, \*c;

n = atoi(argv[1]) ;

iters = atoi(argv[2]);

/\*a = (double\*)malloc(n \* sizeof(double));

b = (double\*)malloc(n \* sizeof(double));

Устранение зависимости по данным с помощью редукции

Функция суммирования векторов удобна для распараллели­вания, так как итерации цикла не содержат зависимостей по дан­ным. Ситуация обстоит по-другому в случае вычисления скаляр­ного произведения векторов.

Итерации цикла for содержат зависимости по данным: каждая итерация считывает и изменяет значения общей переменной s. В этом ча­стном случае зависимость можно устранить: итоговое значение s не зависит от порядка, в котором выполняются итерации цикла. Этот факт создает предпосылки для распараллеливания.

Последовательный вариант функции для скалярного произведения векторов

double dotprod(int n, double\* a, double\* b){

int I;

double s;

s = 0;

for(I =0; I < n; I ++) s += a[i] \* b[i];

return s;

}

Применение к циклу директивы for не является коррект­ным, так как доступ к общей переменной s производится разны­ми потоками без синхронизации. Вариант с синхронизацией представлен на листинге далее. К оператору сложного присваивания применена директива atomic. Это означает, что при выполнении данного оператора различными потоками не проис­ходит конфликтов из-за обращения к общей переменной s. Последовательный вариант функции для скалярного произведения векторов

double dotprod(int n, double\* a, double\* b) {

int I;

double s;

s = 0;

for(I =0; I < n; I ++) s += a[i]\* b[i];

return s;

}

Применение к циклу директивы for не является коррект­ным, так как доступ к общей переменной s производится разны­ми потоками без синхронизации. К оператору сложного присваивания в строке 10 применена директива atomic. Это означает, что при выполнении данного оператора различными потоками не проис­ходит конфликтов из-за обращения к общей переменной s.

Параллельный вариант функции вычисления скалярного произведения векторов — вариант с синхронизацией

*double dotprods(int n, double\* a, double\* b) {*

*int I;*

*double s;*

*s = 0;*

*#pragma omp parallel for*

*for(I =0; I < n; I ++)*

*#pragma omp atomic*

*s += a[i] \* b[i];*

*return s;*

*}*

Применение синхронизации в рассматриваемом примере по­зволяет сохранить корректность, но при этом существенно стра­дает эффективность. Действительно, различные потоки вынуж­дены выполнять итерации цикла по очереди. Вследствие этого реального параллельного выполнения не происходит и вместо ускорения наблюдается замедление работы программы.

Альтернативный вариант основан на следующем простом наблюдении: каждый поток может вычислить часть суммы неза­висимо, а потом следует просто просуммировать значения на разных потоках. Операции суммирования элементов массива и другие редукции достаточно часто встречаются в практике разра­ботки программ. Поэтому в ОрепМР предусмотрена специальная опция reduction для поддержки таких операций. Эта опция предусмотрена для директив parallel, for и sections и имеет два аргумента — редуктивную операцию и редуктивную переменную. Напомним, что редуктивная перемен­ная вычисляется независимо на разных потоках, после чего зна­чения, полученные на разных потоках, комбинируются с помо­щью редуктивной операции.

Если применить опцию reduction, функция вычисления скалярного произведения примет вид, представленный на лис­тинге 43. В этом примере используется комбинированная дирек­тива parallel for с опцией reduction (+: s). В результате каждый поток вычислит часть скалярного произведения. Окончательное значение будет .найдено как сумма значений, вычислен­ных на всех потоках.

Параллельный вариант функции вычисления скалярного произведения векторов — вариант с синхронизацией

*double dotprodp(int n, double\* a, double\* b) {*

*int I;*

*double s;*

*s = 0;*

*#pragma omp parallel for reduction(+:s)*

*for(I =0; I < n; I ++) s += a[i] \* b[i];*

*return s;*

*}*

Применение синхронизации в рассматриваемом примере по­зволяет сохранить корректность, но при этом существенно стра­дает эффективность. Действительно, различные потоки вынуж­дены выполнять итерации цикла по очереди. Вследствие этого реального параллельного выполнения не происходит и вместо ускорения наблюдается замедление работы программы.

Альтернативный вариант основан на следующем простом наблюдении: каждый поток может вычислить часть суммы неза­висимо, а потом следует просто просуммировать значения на разных потоках. Операции суммирования элементов массива и другие редукции достаточно часто встречаются в практике разра­ботки программ. Поэтому в ОрепМР предусмотрена специальная опция reduction для поддержки таких операций. Эта опция предусмотрена для директив parallel, for и sections и имеет два аргумента — редуктивнаую операцию и редуктивную переменную. Напомним, что редуктивная перемен­ная вычисляется независимо на разных потоках, после чего зна­чения, полученные на разных потоках, комбинируются с помо­щью редуктивной операции.

Если применить опцию reduction, функция вычисления скалярного произведения примет вид, представленный в лис­тинге. В этом примере используется комбинированная дирек­тива parallel for с опцией reduction (+: s). В результате каждый поток вычислит часть скалярного произведения. Окончательное значение будет .найдено как сумма значений, вычислен­ных на всех потоках

Параллельный вариант функции вычисления скалярного произведения векторов с применением опции reduction

*double dotprodp(int n, double\* a, double\* b) {*

*int I;*

*double s;*

*s = 0;*

*#pragma omp parallel for reduction(+:s)*

*for(I =0; I < n; I ++) s += a[i] \* b[i];*

*return s;*

Функции, представленные на трех листингах, вычисляют одно и то же значение, но работают с различной производительно­стью. Параллельный вариант с синхронизацией работает неприем­лемо долго: время работы превышает время последовательного варианта на два порядка.

Стратегия распределения итераций цикла по потокам

Итерации цикла, к которому применена директива for, могут по-разному распределяться по потокам.

Повлиять на стратегию рас­пределения, применяемую по умолчанию, позволяет опция sched­ule директивы for.

Данная опция предусматривает два аргумента. Первый определяет способ распределения итераций.

Второй необя­зательный аргумент задает число итераций в порции, которая слу­жит единицей распределения нагрузки.

Предусмотрены четыре раз­личных варианта соответствующие различным комбинациям аргу­ментам schedule (табл.).

Если опция не указана, то применяется стратегия, установленная по умолчанию.

В качестве примера применения опции schedule можно привести знакомый нам цикл вычисления скалярного произведе­ния векторов, для которого зададим статическое распределение порциями по 8 итераций:

В результате итерации цикла будут распределены цикличе­ски между потоками блоками по 8 итераций.

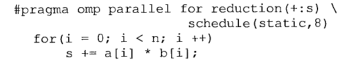
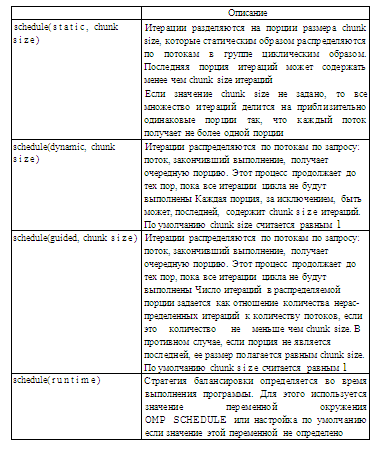


Таблица Опции, определяющие стратегию распределения

итераций цикла по потокам



**Пример выполнения лабораторной работы**

Реализована средствами OpenMP та же задача распараллеливания вычисления суммы квадратов элементов массива, что использовалась в качестве иллюстрации методических указаний по выполнению лабораторной работы 1. Предлагается два варианта реализации – с использованием критической секции и редуктивной переменной.

#include "stdafx.h"

#include <stdio.h>

#include <iostream>

#include <stdlib.h>

#include <time.h>

#include <cmath>

#include <omp.h>

#include <windows.h>

using namespace std;

const int N = 100000000;

int main() {

setlocale(LC\_ALL,"Rus");

int n = 4;

static int a[N];

for (int i = 0; i < (sizeof(a)/sizeof(int)); i++) {

a[i] = i;

}

double time = omp\_get\_wtime();

unsigned long int rez = 0, s = 0;

omp\_set\_dynamic(false);

omp\_set\_num\_threads(n);

long unsigned int sum = 0;

long unsigned psum = 0;

#pragma omp parallel firstprivate(psum) shared(sum) //shedule(static,1000000)

{

psum = 0;

#pragma omp for

for (int i = 0; i < (sizeof(a)/sizeof(int)); i++) {

psum += a[i] \* a[i];

}

#pragma omp critical

{

sum += psum;

}

}

cout << "Сумма квадратов с потоками = " << sum << "\n";

time = omp\_get\_wtime()-time;

cout << "Время вычислений = " << time << "\n";

//-----------------------------

time = omp\_get\_wtime();

sum = 0;

#pragma omp parallel for reduction (+:sum)

for (int i = 0; i < (sizeof(a)/sizeof(int)); i++) {

sum += a[i] \* a[i];

}

time = omp\_get\_wtime()-time;

cout << "Сумма квадратов с потоками через редукцию = " << sum << "\n";

cout << "Время вычислений = " << time << "\n";

//-----------------------------

time = omp\_get\_wtime();

s = 0;

for (int i = 0; i < (sizeof(a)/sizeof(int)); i++) {

s += a[i] \* a[i];

}

time = omp\_get\_wtime()-time;

cout << "Сумма квадратов без потоков = " << s << "\n";

cout << "Время вычислений = " << time << "\n";

system("pause");

return 0;

}

# Индивидуальные задания на лабораторную работу

Краткое содержание работы: Распараллеливание вычислений средствами OpenMP.

Лабораторная работа выполняется на языке C++ в среде Visual Studio. Во всех заданиях следует обеспечить контроль вводимой информации. При некорректном вводе – повторно запрашивать информацию у пользователя.

Выполнить вычисления в однопоточном и многопоточном режиме и сравнить времена выполнения. Количество потоков задаётся пользователем. Количество исходных данных не кратно в общем случае количеству потоков. Исходные данные для задания генерируются с помощью генератора псевдослучайных чисел, где 100000<n<1000000, 100<An<10000000. Результаты сравниваются по времени выполнения при разном числе процессов и объёме данных и оформляются в виде таблицы. В отчёте приводятся снимки экрана, программный код, таблицы тестов и замеров времени выполнения, формулируется вывод. Количество потоков по-умолчанию при необходимости получать средствами OpenMP.

*Примечание: В качестве первого этапа работы рекомендуется переписать лабораторную работу № 1 (задание A) на язык C++, а затем применить к нему средства распараллеливания.*

**Варианты заданий:**

1. Даны последовательности чисел А = {*а0…аn–1*} и С = {*с0…сn–1*}. Создать многопоточное приложение, определяющее, количество совпадающих элементов А и С.

2. Дана последовательность чисел С = {*с0…сn–1*}. Дан набор из *N* пар кодирующих чисел (*ai,bi*), т.е. все *ai* заменяются на  *bi*. Создать многопоточное приложение, кодирующее последовательность Сследующим образом: массив разделяется на подмассивы и каждый поток осуществляет кодирование своего подмассива.

3. Дана последовательность чисел С = {*с0…сn–1*} и число *b*. Создать многопоточное приложение для определения количество вхождений числа *b* в массив C.

4. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска суммы корней квадратных из ai.

5. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска максимального *ai*.

6. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска минимального *ai*.

7. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска всех *ai*, являющихся простыми числами.

8. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска всех *ai*, являющихся квадратами, любого натурального числа.

9. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для вычисления выражения *a0-а1+a2-а3+a4-а5+...*

10. Дана последовательность натуральных чисел {*a0…an–1*}. Создать многопоточное приложение для поиска суммы *∑ai*, где *ai*– четные числа.

­