**Лабораторная работа №1**

**“Знакомство с OpenMP”**

Технология OpenMP является одним из наиболее популярных средств программирования для компьютерных систем с общей памятью, базирующихся на традиционных языках программирования. OpenMP состоит из набора директив для компиляторов и библиотек специальных функций.

В OpenMP распараллеливание выполняется при помощи вставки в текст программы специальных директив, вызова вспомогательных функций и использования переменных окружения. При использовании OpenMP предполагается SPMD-модель (Single ProgramMultiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код. При выполнении параллельной программы работа начинается с инициализации и выполнения главного потока (процесса), который по мере необходимости создает и выполняет параллельные потоки, передавая им необходимые данные. Параллельные потоки из одной параллельной области программы могут выполняться как независимо друг от друга, так и с пересылкой и получением сообщений от других параллельных потоков.

Для обмена данными между параллельными процессами (потоками) в OpenMP используются общие переменные. При обращении к общим переменным в различных параллельных потоках возможно возникновение конфликтных ситуаций при доступе к данным. Для предотвращения конфликтов можно воспользоваться процедурой синхронизации.

Выполнение параллельных потоков в параллельной области программы начинается с их инициализации. Она заключается в создании дескрипторов порождаемых потоков и копировании всех данных из области данных главного потока в области данных создаваемых параллельных потоков. После порождения потоки нумеруются последовательными натуральными числами, причем главный поток имеет номер 0.

Наибольший ресурс параллелизма в программах сосредоточен в циклах. Поэтому наиболее распространенным способом распараллеливания является то или иное распределение итераций циклов. Если между итерациями некоторого цикла нет информационных зависимостей, то их можно каким-либо 48 способом раздать разным процессорам для одновременного исполнения. Различные способы распределения итераций позволяют добиваться максимально равномерной загрузки нитей, между которыми распределяются итерации цикла. Статический способ распределения итераций позволяет уже в момент написания программы точно определить, какой нити достанутся какие итерации. Однако он не учитывает текущей загруженности процессоров, соотношения времён выполнения различных итераций и некоторых других факторов. Эти факторы в той или иной степени учитываются динамическими способами распределения итераций. Кроме того, возможно отложить решение по способу распределения итераций на время выполнения программы (например, выбирать его, исходя из текущей загруженности нитей) или возложить выбор распределения на компилятор и/или систему выполнения. Обмен данными в OpenMP происходит через общие переменные. Это приводит к необходимости разграничения одновременного доступа разных нитей к общим данным. Для этого предусмотрены достаточно развитые средства синхронизации.

Программа, созданная с использованием технологии OpenMP, может быть использована и в качестве последовательной программы. Таким образом, нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова функций OpenMP могут быть подставлены специальные «заглушки» (stubs), текст которых приведен в описании стандарта. Они гарантируют корректную работу программы в последовательном случае – нужно только перекомпилировать программу и подключить другую библиотеку.

Значительная часть функциональности OpenMP реализуется при помощи директив компилятору. Они должны быть явно вставлены пользователем, что позволит выполнять программу в параллельном режиме. Директивы OpenMP в программах на языке Си являются указаниями препроцессору, начинающимися с #pragma omp. Формат директивы на Си/Си++: #pragma omp [опция [[, ] опция …]

Объектом действия большинства директив является один оператор или блок, перед которым расположена директива в исходном тексте программы. В OpenMP такие операторы или блоки называются ассоциированными с директивой. Ассоциированный блок должен иметь одну точку входа в начале и одну точку выхода в конце. Порядок опций в описании директивы несущественен, в одной директиве большинство опций может встречаться несколько раз.

Все директивы OpenMP можно разделить на 3 категории: определение параллельной области, распределение работы, синхронизация. Чтобы задействовать функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл omp.h. Если вы используете в приложении только OpenMP-директивы, включать этот файл не требуется. Функции назначения параметров имеют приоритет над соответствующими переменными окружения.

Параллельная область задается при помощи директивы parallel:

**#pragma omp parallel [опция [[, ] опция …]**

Возможные опции для этой директивы:

- **num\_threads(<целое число>)** позволяет установить количество нитей (процессов)

- **shared (<список>)** задает список переменных, размещающихся в одной и той же области памяти для всех потоков (то есть они будут совместно использованы)

- **private (<список>)** задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;

- **if(<условие>)** – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме;

- **reduction(<оператор>: <список>)** – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: +, \*, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

Перед запуском проекта необходимо установить в свойствах проекта «С/С++» -> «Язык» -> «Поддержка OpenMP».

**Пример 1**

int main()

{

cout << “Последовательная часть” << endl;

#pragma omp parallel num\_threads(4)

{

cout << “Параллельная часть” <<endl;

}

}

Рассмотрим некоторые функции библиотеки OpenMP периода выполнения:

Функция **omp\_set\_num\_threads(int num)**; - позволяет установить значение переменной среды OMP\_NUM\_TREADS, значение которой используется для определения количества нитей, если не задано иное.

В некоторых случаях система может динамически изменять количество нитей, используемых для выполнения параллельной области, например, для оптимизации использования ресурсов системы. Это возможно, если переменная среды OMP\_DYNAMIC установлена в true. Переменную OMP\_DYNAMIC можно установить с помощью функции **omp\_set\_dynamic().**

Функция **omp\_get\_max\_threads()** возвращает максимально допустимое число нитей для использования в следующей параллельной области.

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды OMP\_NESTED. Функция **omp\_set\_nested()** разрешает или запрещает вложенный параллелизм. В качестве значения параметра задаётся 0 или 1.

Функция **omp\_get\_thread\_num()** позволяет получить номер текущей нити.

**Пример 2**

int main()

{

omp\_set\_num\_threads(2);

#pragma omp parallel num\_threads(3)

{

cout<<"Параллельная область 1" << endl;

}

#pragma omp parallel

{

cout<<"Параллельная область 2\n" << endl;

}

}

**Пример 3**

int main()

{

#pragma omp parallel num\_threads(4)

{

cout << omp\_get\_thread\_num() <<endl;

}

}

В OpenMP предусмотрены функции для работы с системным таймером. Функция **omp\_get\_wtime()** возвращает в вызвавшей нити астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Если некоторый участок программы окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменён за время существования процесса. Таймеры разных нитей могут быть не синхронизированы и выдавать различные значения.

**Пример 4**

int main()

{

double t1 = omp\_get\_wtime();

#pragma omp parallel num\_threads(4)

{

cout << omp\_get\_thread\_num() << " ";

}

double t2 = omp\_get\_wtime();

cout << t2 - t1 << endl;

}

В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

- **shared** (общие; все нити видят одну и ту же переменную);

- **private** (локальные, приватные; каждая нить видит свой экземпляр данной переменной).

Общая переменная всегда существует лишь в одном экземпляре для всей области действия и доступна всем нитям под одним и тем же именем. Объявление локальной переменной вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждой нити. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях. Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум одна нить читает значение общей переменной и как минимум одна нить записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой «гонки данных» (data race), при которой результат выполнения программы непредсказуем.

По умолчанию, все переменные, порождённые вне параллельной области, при входе в эту область остаются общими (shared). Исключение составляют переменные, являющиеся счетчиками итераций в цикле, по очевидным причинам. Переменные, порождённые внутри параллельной области, по умолчанию являются локальными (private).

**Пример 5**

int main()

{

int n=1;

cout << "n в последовательной области (начало): " << n;

#pragma omp parallel private(n)

{

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

cout << "Значение n на нити:" << n;

}

cout << "n в последовательной области (конец): " << n;

return 0;

}

В данном примере демонстрируется использование опции private. В последовательной области переменно n присвоено значение 1. Далее порождается параллельная область, где переменной n каждой нити присваивается номер потока и выводится на экран. После выхода из параллельной области значение n снова оказывается равным 1.

**Лабораторная работа №2**

**“Распределение работы между нитями”**

***Низкоуровневое программирование.***

Все нити в параллельной области нумеруются последовательными целыми числами от 0 до N-1, где N — количество нитей, выполняющих данную область. Можно программировать на самом низком уровне, распределяя работу с помощью функций **omp\_get\_thread\_num()** и **omp\_get\_num\_threads()**, возвращающих номер нити и общее количество порождённых нитей в текущей параллельной области, соответственно. Вызов функции **omp\_get\_thread\_num()** позволяет нити получить свой уникальный номер в текущей параллельной области. Вызов функции **omp\_get\_num\_threads()** позволяет нити получить количество нитей в текущей параллельной области.

int main()

{

int count, num;

#pragma omp parallel

{

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

num=omp\_get\_thread\_num();

if (num == 0)

cout << "Всего нитей: " << count << endl;

else

cout << "Нить номер " << num << endl;

}

return 0;

}

***Распараллеливание оператора цикла***

Если в параллельной области встретился оператор цикла без дополнительных указаний, то он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла.

Для распределения итераций цикла между различными нитями можно использовать директиву **for**. Эта директива относится к следующему за ней блоку, содержащему оператор for.

**#pragma omp for [опция [[, ] опция …]**

Рассмотрим опции данной директивы.

* **private(список)** – задает список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено.
* **firstprivate(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити мастере.
* **lastprivate(список)** – переменным, перечисленным в списке, присваивается результат с последнего витка цикла.
* **reduction(оператор:список)** – определяется оператор - операции ( +, -, \*, / и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока; кроме того, определяется список локальных переменных, в котором будут сохраняться соответствующие частичные значения; после завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.
* **schedule(type[, chunk])** – опция задаёт, каким образом итерации цикла распределяются между нитями.
* **collapse(n)** — опция указывает, что n последовательных тесно вложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция collapse не задана, то директива относится только к одному непосредственно следующему за ней циклу.
* **ordered** – опция, говорящая о том, что в цикле могут встречаться директивы ordered; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.
* **nowait** – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными. Предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Если директива параллельного выполнения стоит перед набором вложенных циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл

int main()

{

int s, i, n; s=0;

#pragma omp parallel private (i,n) reduction(+:s)

{

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

#pragma omp for

for (i=1;i<10; i++)

{

s=s+i;

cout << "Нить << n << сложила элементы с номером <<" << i);

}

cout << s << endl;

}

return 0;

}

Параметрами опции **schedule** являются следующие:

* **static** – распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.
* **dynamic** – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.
* **guided** – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения chunk.
* **auto** – способ распределения итераций выбирается компилятором и/или системой выполнения. Параметр chunk при этом не задаётся. runtime – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP\_SCHEDULE. Параметр chunk при этом не задаётся. При распараллеливании цикла следует убедиться в том, что его итерации не имеют зависимостей, и их можно выполнять в любом порядке. Несоблюдение данного требования приведет к получению некорректного результата

int main()

{

omp\_set\_num\_threads(4);

int i;

#pragma omp parallel private (i)

{

#pragma omp for schedule (static)

//#pragma omp for schedule (static, 1)

//#pragma omp for schedule (static, 2)

//#pragma omp for schedule (dynamic)

//#pragma omp for schedule (dynamic, 2)

//#pragma omp for schedule (guided)

//#pragma omp for schedule (guided, 2)

for (i = 0; i < 10; i++)

{

cout << "Нить "<< omp\_get\_thread\_num() << " выполнила итерацию " << i << endl;

}

}

}

***Параллельные секции***

Директива OpenMP **sections** используется для выделения участков программы в области параллельных структурных блоков, выполняющихся в отдельных параллельных потоках. Данная директива содержит набор структурированных блоков, которые распределяются по потокам в группе. Каждый структурированный блок исполняется один раз одним из потоков в группе.

**#pragma omp sections опция[[[,] опция] ...]**

При выполнении этого кода OpenMP сначала создает группу потоков, а затем распределяет между ними обработку итераций цикла, после выполнения которого потоки начинают параллельную обработку оставшихся разделов кода. Если количество разделов программного кода будет больше числа потоков, обработка нескольких разделов будет отложена до тех пор, пока не появятся свободные потоки. В отличие от планирования циклов, распределение нагрузки между потоками при обработке параллельных разделов кода осуществляется и контролируется OpenMP. Программисту остается только выбрать, какие переменные будут общими, а какие – индивидуальными, и предусмотреть выражения уменьшения аналогично сегменту с организацией циклов. Рассмотрим возможные опции данной директивы.

* **private(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено.
* **firstprivate(список)** – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити мастере.
* **lastprivate(список)** – переменным, перечисленным в списке, присваивается результат, полученный в последней секции.
* **reduction(оператор: список)** – определяется оператор - операции ( +, -, \*, / и т. п.) или функции, для которых будут вычисляться соответствующие частичные значения в параллельных потоках последующего параллельного структурного блока; кроме того, определяется список локальных переменных, в котором будут сохраняться соответствующие частичные значения; после завершения всех параллельных процессов частичные значения складываются (вычитаются, перемножаются и т. п.), и результат сохраняется в одноименной общей переменной.
* **nowait** – в конце блока секций происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция nowait позволяет нитям, уже дошедшим до конца своих секций, продолжить выполнение без синхронизации с остальными. Перед первым участком кода в блоке sections директива section не обязательна. Какие именно будут задействованы нити для выполнения секции, не специфицируется. Если количество нитей больше количества секций, то некоторые нити не будут задействованы. Если же количество секций больше количества нитей, то некоторые нити выполнят больше одной секции

int main()

{

int n = 0;

#pragma omp parallel

{

#pragma omp sections lastprivate(n)

{

#pragma omp section

{

n = 1;

}

#pragma omp section

{

n = 2;

}

#pragma omp section

{

n = 3;

}

}

cout << "Значение n на нити " << omp\_get\_thread\_num()<< n;

}

cout << “Значение в последовательной области” << endl;

return 0;

}

Опция lastprivate используется вместе с директивой sections. Переменная n объявлена как lastprivate переменная. Три нити, выполняющие секции section, присваивают своей локальной копии n разные значения. По выходе из области sections значение n из последней секции присваивается локальным копиям во всех нитях, поэтому все нити напечатают число 3. Это же значение сохранится для переменной n и в последовательной области.

int main()

{

int A[10], B[10], C[10], i, n;

/\* Заполним исходные массивы \*/

for (i=0; i<10; i++)

{

A[i]=i; B[i]=2\*i; C[i]=0;

}

#pragma omp parallel shared(A,B,C) private(i,n)

{

n = opm\_get\_thread\_num();

#pragma omp for

for (int i=0; i<10; i++)

{

C[i]=A[i]+B[i];

cout << “Нить ” << n << “ сложила элементы с номером ” << i << endl;

}

}

}

Для представленного выше кода произвести замеры времени выполнения параллельной части программы на разном числе нитей и оформить в виде таблицы.

Пример работы с матрицами: параллельное перемножение матриц.

void randomiseMatrix(int\*\* matrix, int n, int m) {

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

for (int j = 0; j < m; j++) {

matrix[i][j] = rand() % 11;

}

}

}

int main() {

srand(time(NULL));

int n1 = 1000;

int m1 = 500;

int n2 = 500;

int m2 = 1200;

//Матрица n1 x m1

int\*\* matrix1;

//Матрица n2 x m2

int\*\* matrix2;

matrix1 = new int\* [n1];

for (int i = 0; i < n1; i++) {

matrix1[i] = new int [m1];

}

matrix2 = new int\* [n2];

for (int i = 0; i < n2; i++) {

matrix2[i] = new int[m2];

}

//Генерируем случайные матрицы для умножения

randomiseMatrix(matrix1, n1, m1);

randomiseMatrix(matrix2, n2, m2);

int\*\* result;

result = new int\* [n1];

for (int i = 0; i < n1; i++) {

result[i] = new int[m2];

}

//Устанавливаем число потоков

int threadsNum = 2;

omp\_set\_num\_threads(threadsNum);

int i, j, k;

#pragma omp parallel for shared(matrix1, matrix2, result) private(i, j, k)

for (i = 0; i < n1; i++) {

for (j = 0; j < m2; j++) {

result[i][j] = 0;

for (k = 0; k < m1; k++) {

result[i][j] += (matrix1[i][k] \* matrix2[k][j]);

}

}

}

return 0;

}

Решить задачу, оценить ускорение параллельной программы относительно последовательного выполнения.