Лабораторная работа №4

Распределение работы

OpenMP предлагает несколько вариантов распределения работы между запущенными нитями. Конструкции распределения работ в OpenMP не порождают новых нитей.

Низкоуровневое распараллеливание

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.

Си:

int omp_get_thread_num(void);

Вызов функции **omp_get_num_threads()** позволяет нити получить количество нитей в текущей параллельной области.

Си:

int omp_get_num_threads(void);

Пример 17 демонстрирует работу функций **omp_get_num_threads()** и **omp_get_thread_num()**. Нить, порядковый номер которой равен **0**, напечатает общее количество порождённых нитей, а остальные нити напечатают свой порядковый номер.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int count, num;
#pragma omp parallel
{
count=omp_get_num_threads();
num=omp_get_thread_num();
if (num == 0) printf("Всего нитей: %d\n", count);
else printf("Нить номер %d\n", num);
}
}
```

Пример *17.* Функции *omp_get_num_threads()* и *omp_get_thread_num()* на языке Си.

Использование функций **omp_get_thread_num()** и **omp_get_num_threads()** позволяет назначать каждой нити свой кусок кода для выполнения, и таким образом распределять работу между нитями в стиле технологии MPI. Однако использование этого стиля программирования в OpenMP далеко не всегда

оправдано – программист в этом случае должен явно организовывать синхронизацию доступа к общим данным. Другие способы распределения работ в OpenMP обеспечивают значительную часть этой работы автоматически.

Параллельные циклы

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, 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**.

Си:

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

Эта директива относится к идущему следом за данной директивой блоку, включающему операторы **for**

Возможные опции:

_ **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 с последнего витка цикла;

_ **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 завершения всех итераций цикла выполняется заданный оператор; оператор для языка Си – **+**, *****, **-**, **&**, |, ^, **&&**, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

_ **schedule(type[, chunk])** – опция задаёт, каким образом итерации цикла распределяются между нитями;

_ **collapse(n)** — опция указывает, что **n** последовательных тесновложенных 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 про-

исходит 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. Если директива **end do** в явном виде не указана, то в конце параллельного цикла синхронизация все равно будет выполнена.
На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла.
Размер блока итераций, указанный в опции **schedule**, не должен изменяться в рамках цикла.
Формат параллельных циклов на языке Си упрощённо можно представить следующим образом:
for([целочисленный тип**] i =** инвариант цикла**;**
i {<,>,=,<=,>=} инвариант цикла**;**
i {+,-}= инвариант цикла**)**
Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций.
Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл.
Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции **lastprivate**.
Пример 18 демонстрирует использование директивы **for**. В последовательной области инициализируются три исходных массива **A**, **B**, **C**. В параллельной области данные массивы объявлены общими. Вспомогательные переменные **i** и **n** объявлены локальными. Каждая нить присвоит переменной **n** свой порядковый номер. Далее с помощью директивы **for** определяется цикл, итерации которого будут распределены между существующими нитями. На каждой **i**-ой итерации данный цикл сложит **i**-ые элементы массивов **A** и **B** и результат запишет в **i**-ый элемент массива **C**. Также на каждой итерации будет напечатан номер нити, выполнившей данную итерацию.

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

```
{
/* Получим номер текущей нити */
n=omp_get_thread_num();
#pragma omp for
for (i=0; i<10; i++)
{
C[i]=A[i]+B[i];
printf("Нить %d сложила элементы с номером %d\n",
n, i);
}
}
}
```

Пример *18.* Директива *for* на языке Си.

В опции **schedule** параметр **type** задаёт следующий тип распределения итераций:

- **static** – блочно-циклическое 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 задается.
- **runtime** – способ распределения итераций выбирается во время работы программы по значению переменной среды **OMP_SCHEDULE**. Параметр **chunk** при этом не задаётся.

Пример 19 демонстрирует использование опции **schedule** с параметрами **(static)**, **(static, 1)**, **(static, 2)**, **(dynamic)**, **(dynamic, 2)**, **(guided)**, **(guided, 2)**. В

параллельной области выполняется цикл, итерации которого будут распределены между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. В тело цикла вставлена также задержка, имитирующая некоторые вычисления.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
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++)
{
printf("Нить %d выполнила итерацию %d\n",
omp_get_thread_num(), i);
sleep(1);
}
}
}
```

Пример *19.* Опция *schedule* на языке Си.

Пример 20 демонстрирует использование опции **schedule** с параметрами **(static, 6)**, **(dynamic, 6)**, **(guided, 6)**. В параллельной области выполняется цикл, итерации которого будут распределены между существующими нитями. На каждой итерации будет напечатано, какая нить выполнила данную итерацию. В тело цикла вставлена также задержка, имитирующая некоторые вычисления.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int i;
#pragma omp parallel private(i)
{
#pragma omp for schedule (static, 6)
```

```
//#pragma omp for schedule (dynamic, 6)
//#pragma omp for schedule (guided, 6)
for (i=0; i<200; i++)
{
printf("Нить %d выполнила итерацию %d\n",
omp_get_thread_num(), i);
sleep(1);
}
}
}
```

Пример *20.* Опция *schedule* на языке Си.

Значение по умолчанию переменной **OMP_SCHEDULE** зависит от реализации.
Если переменная задана неправильно, то поведение программы при задании опции **runtime** также зависит от реализации.
Задать значение переменной **OMP_SCHEDULE** в Linux в командной оболочке bash можно при помощи команды следующего вида:

```
export OMP_SCHEDULE="dynamic,1"
```

Изменить значение переменной **OMP_SCHEDULE** из программы можно с помощью вызова функции **omp_set_schedule()**.
Си:

```
void omp_set_schedule(omp_sched_t type, int chunk);
```

Допустимые значения констант описаны в файле **omp.h**. Как минимум, они должны включать для языка Си следующие варианты:

```
typedef enum omp_sched_t {
omp_sched_static = 1,
omp_sched_dynamic = 2,
omp_sched_guided = 3,
omp_sched_auto = 4
} omp_sched_t;
```

При помощи вызова функции **omp_get_schedule()** пользователь может узнать текущее значение переменной **OMP_SCHEDULE**.
Си:

```
void omp_get_schedule(omp_sched_t* type, int* chunk);
```

При распараллеливании цикла программист должен убедиться в том, что итерации данного цикла не имеют информационных зависимостей. Если цикл не содержит зависимостей, его итерации можно выполнять в любом порядке, в том числе параллельно. Соблюдение этого важного требования компилятор не проверяет, вся ответственность лежит на программисте. Если дать указание компилятору распараллелить цикл, содержащий зависимости, компилятор это сделает, но результат работы программы может оказаться некорректным.

Параллельные секции

Директива **sections** используется для задания конечного (неитеративного) параллелизма.

Си:

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

Эта директива определяет набор независимых секций кода, каждая из которых выполняется своей нитью.

Возможные опции:

_ **private(**список**)**;

_ **firstprivate(**список**)**;

_ **lastprivate(**список**)**;

_ **reduction(**оператор**:**список);

_ **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.

Директива **section** задаёт участок кода внутри секции **sections** для выполнения одной нитью.

Си:

#pragma omp section

Перед первым участком кода в блоке **sections** директива **section** не обязательна. Какие именно нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции.

Пример 21 иллюстрирует применение директивы **sections**. Сначала три нити, на которые распределились три секции **section**, выведут сообщение со своим номером, а потом все нити напечатают одинаковое сообщение со своим номером.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
#pragma omp sections
{
#pragma omp section
```

```
{
printf("Первая секция, процесс %d\n", n);
}
#pragma omp section
{
printf("Вторая секция, процесс %d\n", n);
}
#pragma omp section
{
printf("Третья секция, процесс %d\n", n);
}
}
printf("Параллельная область, процесс %d\n", n);
}
}
```

Пример *21.* Директива *sections* на языке Си.

Пример 22 демонстрирует использование опции **lastprivate**. В данном примере 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.

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

```
}
}
printf("Значение n на нити %d: %d\n",
omp_get_thread_num(), n);
}
printf("Значение n в последовательной области: %d\n", n);
}
```

Пример *22.* Опция *lastprivate* на языке Си.

Задачи *(tasks)*

Директива **task** применяется для выделения отдельной независимой задачи.

Си:

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

Текущая нить выделяет в качестве задачи ассоциированный с директивой блок операторов. Задача может выполняться немедленно после создания или быть отложенной на неопределённое время и выполняться по частям. Размер таких частей, а также порядок выполнения частей разных отложенных задач определяется реализацией.

Возможные опции:

_ **if(**условие**)** — порождение новой задачи только при выполнении некоторого условия; если условие не выполняется, то задача будет выполнена текущей нитью и немедленно;

_ **untied** — опция означает, что в случае откладывания задача может быть продолжена любой нитью из числа выполняющих данную параллельную область; если данная опция не указана, то задача может быть продолжена только породившей её нитью;

_ **default(private|firstprivate|shared|none)** – всем переменным в задаче, которым явно не назначен класс, будет назначен класс **private**, **firstprivate** или **shared** соответственно; **none** означает, что всем переменным в задаче класс должен быть назначен явно; в языке Си задаются только варианты **shared** или **none**;

_ **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;

_ **shared(**список**)** – задаёт список переменных, общих для всех нитей.

Для гарантированного завершения в точке вызова всех запущенных задач используется директива **taskwait**.

Си:

#pragma omp taskwait

Нить, выполнившая данную директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные данной нитью независимые задачи.

Задания

1) Могут ли функции **omp_get_thread_num()** и **omp_get_num_threads()** вернуть одинаковые значения на нескольких нитях одной параллельной области?
2) Можно ли распределить между нитями итерации цикла без использования директивы **for**?
3) Можно ли одной директивой распределить между нитями итераци сразу нескольких циклов?
4) Возможно ли, что при статическом распределении итераций цикла нитям достанется разное количество итераций?
5) Могут ли при повторном запуске программы итерации распределяемого цикла достаться другим нитям? Если да, то при каких способах распределения итераций?
6) Для чего может быть полезно указывать параметр **chunk** при способе распределения итераций **guided**?
7) Можно ли реализовать параллельные секции без использования директив **sections** и **section**?
8) Как при выходе из параллельных секций разослать значение некоторой локальной переменной всем нитям, выполняющим данную параллельную область?
9) В каких случаях может пригодиться механизм задач?
10) Напишите параллельную программу, реализующую скалярное произведение двух векторов.
11) Напишите параллельную программу, реализующую поиск максимального значения вектора.