Лабораторная работа №2.
Параллельные и последовательные области

В момент запуска программы порождается единственная нить-мастер или «основная» нить, которая начинает выполнение программы с первого оператора. Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити.

Директива *parallel*

Параллельная область задаётся при помощи директивы **parallel**.

Си:

```
#pragma omp parallel [опция [,] опция]...]
```

Возможные опции:

- **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;
- **num_threads (**целочисленное выражение**)** – явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции **omp_set_num_threads()**, или значение переменной **OMP_NUM_THREADS**;
- **default(private|firstprivate|shared|none)** – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс **private**, **firstprivate** или **shared** соответственно;
- **none** означает, что всем переменным в параллельной области 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;
- **copyin(**список**)** – задаёт список переменных, объявленных как **threadprivate**, которые при входе в параллельную область инициализируются значениями соответствующих переменных в нити-мастере;
- **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: для языка Си – **+**, *****, **-**, **&**, **|**, **^**, **&&**, **||**; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

При входе в параллельную область порождаются новые **OMP_NUM_THREADS-1** нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер **0** и становится основной нитью группы («мастером»).

Остальные нити получают в качестве номера целые числа с **1** до **OMP_NUM_THREADS-1**. Количество нитей, выполняющих данную параллельную область, остаётся неизменным до момента выхода из области. При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей.

Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Пример 3 демонстрирует использование директивы **parallel**. В результате выполнения нить-мастер напечатает текст "Последовательная область **1**", затем по директиве **parallel** порождаются новые нити, каждая из которых напечатает текст "Параллельная область", затем порождённые нити завершаются и оставшаяся нить-мастер напечатает текст "Последовательная область **2**".

```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("Последовательная область 1\n");
#pragma omp parallel
{
printf("Параллельная область\n");
}
printf("Последовательная область 2\n");
}
```

Пример *3*. Параллельная область на языке Си.

Пример 4 демонстрирует применение опции **reduction**. В данном примере производится подсчет общего количества порождённых нитей. Каждая нить инициализирует локальную копию переменной **count** значением **0**. Далее, каждая нить увеличивает значение собственной копии переменной **count** на единицу и выводит полученное число. На выходе из параллельной

области происходит суммирование значений переменных **count** по всем нитям, и полученная величина становится новым значением переменной **count** в последовательной области.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int count = 0;
#pragma omp parallel reduction (+: count)
{
count++;
printf("Текущее значение count: %d\n", count);
}
printf("Число нитей: %d\n", count);
}
```

Пример *4*. Опция *reduction* на языке Си.

Сокращённая запись

Если внутри параллельной области содержится только один параллельный цикл, одна конструкция **sections** или одна конструкция **workshare**, то можно использовать укороченную запись: **parallel for**, **parallel sections** или **parallel workshare**. При этом допустимо указание всех опций этих директив, за исключением опции **nowait.**

Переменные среды и вспомогательные функции

Перед запуском программы количество нитей, выполняющих параллельную область, можно задать, определив значение переменной среды **OMP_NUM_THREADS**. Например, в Linux в командной оболочке bash это можно сделать при помощи следующей команды:

```
export OMP_NUM_THREADS=n
```

Значение по умолчанию переменной **OMP_NUM_THREADS** зависит от реализации. Из программы её можно изменить с помощью вызова функции **omp_set_num_threads().**

Си:

```
void omp_set_num_threads(int num);
```

Пример 5 демонстрирует применение функции **omp_set_num_threads()** и опции **num_threads**. Перед первой параллельной областью вызовом функции **omp_set_num_threads(2)** выставляется количество нитей, равное **2.** Но к первой параллельной области применяется опция **num_threads(3),** которая указывает, что данную область следует выполнять тремя нитями. Следовательно, сообщение "Параллельная область **1**" будет выведено тремя нитями. Ко второй параллельной области

опция **num_threads** не применяется, поэтому действует значение, установленное функцией **omp_set_num_threads(2)**, и сообщение "Параллельная область **2**" будет выведено двумя нитями.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
{
printf("Параллельная область 1\n");
}
#pragma omp parallel
{
printf("Параллельная область 2\n");
}
}
```

Пример 5. Функция *omp_set_num_threads()* и опция *num_threads* на языке Си.

В некоторых случаях система может динамически изменять количество нитей, используемых для выполнения параллельной области, например, для оптимизации использования ресурсов системы. Это разрешено делать, если переменная среды **OMP_DYNAMIC** установлена в **true**. Например, в Linux в командной оболочке bash её можно установить при помощи следующей команды:

```
export OMP_DYNAMIC=true
```

В системах с динамическим изменением количества нитей значение по умолчанию не определено, иначе значение по умолчанию: **false**.

Переменную **OMP_DYNAMIC** можно установить с помощью функции **omp_set_dynamic()**.

Си:

```
void omp_set_dynamic(int num);
```

На языке Си в качестве значения параметра функции **omp_set_dynamic()** задаётся **0** или **1.** Если система не поддерживает динамическое изменение количества нитей, то при вызове функции **omp_set_dynamic()** значение переменной **OMP_DYNAMIC** не изменится.

Узнать значение переменной **OMP_DYNAMIC** можно при помощи функции **omp_get_dynamic()**.

Си:

```
int omp_get_dynamic(void);
```

Пример 6 демонстрирует применение функций **omp_set_dynamic()** и **omp_get_dynamic()**. Сначала распечатывается значение, полученное

функцией **omp_get_dynamic()** – это позволяет узнать значение переменной **OMP_DYNAMIC** по умолчанию. Затем при помощи функции **omp_set_dynamic()** переменная **OMP_DYNAMIC** устанавливается в **true**, что подтверждает выдача ещё один раз значения функции **omp_get_dynamic()**.

Затем порождается параллельная область, выполняемая заданным количеством нитей (**128**). В параллельной области печатается реальное число выполняющих её нитей. Директива **master** позволяет обеспечить печать только процессом-мастером. В системах с динамическим изменением числа нитей выданное значение может отличаться от заданного (**128**).

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
printf("Значение OMP_DYNAMIC: %d\n", omp_get_dy-
namic());
omp_set_dynamic(1);
printf("Значение OMP_DYNAMIC: %d\n", omp_get_dy-
namic());
#pragma omp parallel num_threads(128)
{
#pragma omp master
{
printf("Параллельная область, %d нитей\n",
omp_get_num_threads());
}
}
}
```

Пример 6. Функции *omp_set_dynamic()* и *omp_get_dynamic()* на языке Си.

Функция **omp_get_max_threads()** возвращает максимально допустимое число нитей для использования в следующей параллельной области.
Си:

```
int omp_get_max_threads(void);
```

Функция **omp_get_num_procs()** возвращает количество процессоров, доступных для использования программе пользователя на момент вызова. Нужно учитывать, что количество доступных процессоров может динамически изменяться.
Си:

```
int omp_get_num_procs(void);
```

Параллельные области могут быть вложенными; по умолчанию вложенная параллельная область выполняется одной нитью. Это управляется установкой переменной среды **OMP_NESTED**. Например, в Linux в командной

оболочке bash разрешить вложенный параллелизм можно при помощи следующей команды:

```
export OMP_NESTED=true
```

Изменить значение переменной **OMP_NESTED** можно с помощью вызова функции **omp_set_nested().**

Си:

```
void omp_set_nested(int nested)
```

Функция **omp_set_nested()** разрешает или запрещает вложенный параллелизм. На языке Си в качестве значения параметра задаётся **0** или **1.** Если вложенный параллелизм разрешён, то каждая нить, в которой встретится описание параллельной области, породит для её выполнения новую группу нитей. Сама породившая нить станет в новой группе нитью-мастером. Если система не поддерживает вложенный параллелизм, данная функция не будет иметь эффекта.

Пример 7 демонстрирует использование вложенных параллельных областей и функции **omp_set_nested().** Вызов функции **omp_set_nested()** перед первой частью разрешает использование вложенных параллельных областей.

Для определения номера нити в текущей параллельной секции используются вызовы функции **omp_get_thread_num().** Каждая нить внешней параллельной области породит новые нити, каждая из которых напечатает свой номер вместе с номером породившей нити. Далее вызов **omp_set_nested()** запрещает использование вложенных параллельных областей. Во второй части вложенная параллельная область будет выполняться без порождения новых нитей, что и видно по получаемой выдаче.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
omp_set_nested(1);
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
#pragma omp parallel
{
printf("Часть 1, нить %d - %d\n", n,
omp_get_thread_num());
}
}
omp_set_nested(0);
#pragma omp parallel private(n)
{
```

```
n=omp_get_thread_num();
#pragma omp parallel
{
printf("Часть 2, нить %d - %d\n", n,
omp_get_thread_num());
}
}
}
```

Пример *7*. Вложенные параллельные области на языке Си.

Узнать значение переменной **OMP_NESTED** можно при помощи функции **omp_get_nested()**.

Си:

```
int omp_get_nested(void);
```

Функция **omp_in_parallel()** возвращает **1,** если она была вызвана из активной параллельной области программы.

Си:

```
int omp_in_parallel(void);
```

Пример 8 иллюстрирует применение функции **omp_in_parallel()**. Функция **mode** демонстрирует изменение функциональности в зависимости от того, вызвана она из последовательной или из параллельной области. В последовательной области будет напечатано "Последовательная область", а в параллельной – "Параллельная область".

```
#include <stdio.h>
#include <omp.h>
void mode(void){
if(omp_in_parallel()) printf("Параллельная область\n");
else printf("Последовательная область\n");
}
int main(int argc, char *argv[])
{
mode();
#pragma omp parallel
{
#pragma omp master
{
mode();
}
}
}
```

Пример *8*. Функция *omp_in_parallel()* на языке Си.

Директива *single*

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивами **single.**

Си:

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

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;

_ **copyprivate(**список**)** – после выполнения нити, содержащей конструкцию **single**, новые значения переменных списка будут доступны всем одноименным частным переменным (**private** и **firstprivate**), описанным в начале параллельной области и используемым всеми её нитями; опция не может использоваться совместно с опцией **nowait**; переменные списка не должны быть перечислены в опциях **private** и **firstprivate** данной директивы **single**;

_ **nowait** – после выполнения выделенного участка 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 участка, продолжить выполнение без синхронизации с остальными.

Какая именно нить будет выполнять выделенный участок программы, не специфицируется. Одна нить будет выполнять данный фрагмент, а все остальные нити будут ожидать завершения её работы, если только не указана опция **nowait**. Необходимость использования директивы **single** часто возникает при работе с общими переменными.

Пример 9 иллюстрирует применение директивы **single** вместе с опцией **nowait**. Сначала все нити напечатают текст "Сообщение **1**", при этом одна нить (не обязательно нить-мастер) дополнительно напечатает текст "Одна нить". Остальные нити, не дожидаясь завершения выполнения области **single**, напечатают текст "Сообщение **2**". Таким образом, первое появление "Сообщение **2**" в выводе может встретиться как до текста "Одна нить", так и после него. Если убрать опцию **nowait**, то по окончании области **single** произойдёт барьерная синхронизация, и ни одна выдача "Сообщение **2**" не может появиться до выдачи "Одна нить".

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

```
{
#pragma omp parallel
{
printf("Сообщение 1\n");
#pragma omp single nowait
{
printf("Одна нить\n");
}
printf("Сообщение 2\n");
}
}
```

Пример *9*. Директива *single* и опция *nowait* на языке Си.

Пример 10 иллюстрирует применение опции **copyprivate**. В данном примере переменная **n** объявлена в параллельной области как локальная. Каждая нить присвоит переменной **n** значение, равное своему порядковому номеру, и напечатает данное значение. В области **single** одна из нитей присвоит переменной **n** значение **100**, и на выходе из области это значение будет присвоено переменной **n** на всех нитях. В конце параллельной области значение **n** печатается ещё раз и на всех нитях оно равно **100**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
printf("Значение n (начало): %d\n", n);
#pragma omp single copyprivate(n)
{
n=100;
}
printf("Значение n (конец): %d\n", n);
}
}
```

Пример *10*. Опция *copyprivate* на языке Си.

Директива *master*

Директивы **master** выделяют участок кода, который будет выполнен только нитью-мастером. Остальные нити просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

Си:

```
#pragma omp master
```

Пример 11 демонстрирует применение директивы **master**. Переменная **n** является локальной, то есть каждая нить работает со своим экземпляром. Сначала все нити присвоят переменной **n** значение **1**. Потом нить-мастер присвоит переменной **n** значение **2**, и все нити напечатают значение **n**. Затем нить-мастер присвоит переменной **n** значение **3**, и снова все нити напечатают значение **n**. Видно, что директиву **master** всегда выполняет одна и та же нить. В данном примере все нити выведут значение **1**, а нить-мастер сначала выведет значение **2**, а потом - значение **3**.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel private(n)
{
n=1;
#pragma omp master
{
n=2;
}
printf("Первое значение n: %d\n", n);
#pragma omp barrier
#pragma omp master
{
n=3;
}
printf("Второе значение n: %d\n", n);
}
}
```

Пример *11*. Директива *master* на языке Си.

Задания

1. Определите, какое максимальное количество нитей позволяет породить для выполнения параллельных областей программы ваша система.
2. В каких случаях может быть необходимо использование опции **if** директивы **parallel**?
3. Определите, сколько процессоров доступно в вашей системе для выполнения параллельной части программы, и займите каждый из доступных процессоров выполнением одной нити в рамках общей параллельной области.

4. При помощи трёх уровней вложенных параллельных областей породите 8 нитей (на каждом уровне параллельную область должны исполнять 2 нити). Посмотрите, как будет исполняться программа, если запретить вложенные параллельные области
5. Чем отличаются директивы **`single`** и **`master`**?
6. Может ли нить-мастер выполнить область, ассоциированную с директивой **`single`**?
7. Может ли нить с номером **1** выполнить область, ассоциированную с директивой **`master`**?