Лабораторная работа №6
Синхронизация
Целый набор директив в OpenMP предназначен для синхронизации работы нитей.
Барьер
Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы **barrier**.
Си:
#pragma omp barrier
Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (**task**).
Пример 24 демонстрирует применение директивы **barrier**. Директива **barrier** используется для упорядочивания вывода от работающих нитей. Выдачи с разных нитей "Сообщение **1**" и "Сообщение **2**" могут перемежаться в произвольном порядке, а выдача "Сообщение **3**" со всех нитей придёт строго после двух предыдущих выдач.

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

Пример *24.* Директива *barrier* на языке Си.

Директива ***ordered***
Директивы **ordered** (**ordered ... end ordered**) определяют блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.
Си:
#pragma omp ordered
Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция **ordered**. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.
Пример 25 иллюстрирует применение директивы **ordered** и опции **ordered**. Цикл **for** помечен как **ordered**. Внутри тела цикла идут две выдачи – одна вне блока **ordered**, а вторая – внутри него. В результате первая выдача получается неупорядоченной, а вторая идёт в строгом порядке по возрастанию номера итерации.

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

```
n=omp_get_thread_num();
#pragma omp for ordered
for (i=0; i<5; i++)
{
printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
{
printf("ordered: Нить %d, итерация %d\n", n, i);
}
}
}
}
```
Пример *25.* Директива *ordered* и опция *ordered* на языке Си.

Критические секции

С помощью директивы **critical** оформляется критическая секция программы.

Си:

#pragma omp critical [(<имя_критической_секции>**)]**

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

Пример 26 иллюстрирует применение директивы **critical**. Переменная **n** объявлена вне параллельной области, поэтому по умолчанию является общей. Критическая секция позволяет разграничить доступ к переменной **n**.

Каждая нить по очереди присвоит **n** свой номер и затем напечатает полученное значение.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
#pragma omp parallel
{
#pragma omp critical
{
n=omp_get_thread_num();
printf("Нить %d\n", n);
}
}
}
```
Пример *26.* Директива *critical* на языке Си.

Если бы в примере 26 не была указана директива **critical**, результат выполнения программы был бы непредсказуем. С директивой **critical** порядок вывода результатов может быть произвольным, но это всегда будет набор одних и тех же чисел от **0** до

OMP_NUM_THREADS-1. Конечно, подобного же результата можно было бы добиться другими способами, например, объявив переменную **n** локальной, тогда каждая нить работала бы со своей копией этой переменной. Однако в исполнении этих фрагментов разница существенная.
Если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь какой-либо одной нитью. Остальные нити, даже если они уже подошли к данной точке программы и готовы к работе, будут ожидать своей очереди. Если критической секции нет, то все нити могут одновременно выполнить данный участок кода. С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность.

Директива ***atomic***
Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная **sum** является общей и оператор вида **sum=sum+expr** находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой **atomic**.
Си:
#pragma omp atomic
Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части оператора присваивания, при этом вычисления в правой части не обязаны быть атомарными.
Пример 27 иллюстрирует применение директивы **atomic**. В данном примере производится подсчет общего количества порожденных нитей. Для этого каждая нить увеличивает на единицу значение переменной **count**. Для того, чтобы предотвратить одновременное изменение несколькими нитями значения переменной, стоящей в левой части оператора присваивания, используется директива **atomic**.

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

Пример *27.* Директива *atomic* на языке Си.

Замки
Один из вариантов синхронизации в OpenMP реализуется через механизм замков (*locks*). В качестве замков используются общие целочисленные переменные (размер должен быть

достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.
Замок может находиться в одном из трёх состояний: неинициализированный, разблокированный или заблокированный. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.
Есть два типа замков: простые замки и множественные замки. Множественный замок может многократно захватываться одной нитью перед его освобождением, в то время как простой замок может быть захвачен только однажды. Для множественного замка вводится понятие коэффициента захваченности *(nesting count)*. Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.
Для инициализации простого или множественного замка используются соответственно функции **omp_init_lock()** и **omp_init_nest_lock()**.
Си:
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
После выполнения функции замок переводится в разблокированное состояние. Для множественного замка коэффициент захваченности устанавливается в ноль.
Функции **omp_destroy_lock()** и **omp_destroy_nest_lock()** используются для переведения простого или множественного замка в неинициализированное состояние.
Си:
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
Для захватывания замка используются функции **omp_set_lock()** и **omp_set_nest_lock()**.
Си:
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние.
Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.
Для освобождения замка используются функции **omp_unset_lock()** и **omp_unset_nest_lock()**.
Си:
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_lock_t *lock);
Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения замка есть нити, заблокированные на операции, захватывающей данный замок, замок будет сразу же захвачен одной из ожидающих нитей.
Пример 28 иллюстрирует применение технологии замков. Переменная **lock** используется для блокировки. В последовательной области производится инициализация данной переменной с помощью функции **omp_init_lock()**.
В начале параллельной области каждая нить присваивает переменной **n** свой порядковый номер. После этого с помощью функции **omp_set_lock()** одна из нитей выставляет блокировку, а остальные нити ждут, пока нить, вызвавшая эту функцию, не снимет блокировку с помощью функции **omp_unset_lock()**. Все нити по очереди выведут сообщения "Начало

закрытой секции**...**" и "Конец закрытой секции**...**", при этом между двумя сообщениями от одной нити не могут встретиться сообщения от другой нити.
В конце с помощью функции **omp_destroy_lock()** происходит освобождение переменной **lock**.

```
#include <stdio.h>
#include <omp.h>
omp_lock_t lock;
int main(int argc, char *argv[])
{
int n;
omp_init_lock(&lock);
#pragma omp parallel private (n)
{
n=omp_get_thread_num();
omp_set_lock(&lock);
printf("Начало закрытой секции, нить %d\n", n);
sleep(5);
printf("Конец закрытой секции, нить %d\n", n);
omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
}
```

Пример *28.* Использование замков на языке Си.

Для неблокирующей попытки захвата замка используются функции **omp_test_lock()** и **omp_test_nest_lock()**.
Си:

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает **1**, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается **0.**
Пример 29 иллюстрирует применение технологии замков и использование функции **omp_test_lock()**. В данном примере переменная **lock** используется для блокировки. В начале производится инициализация данной переменной с помощью функции **omp_init_lock()**. В параллельной области каждая нить присваивает переменной **n** свой порядковый номер. После этого с помощью функции **omp_test_lock()** нити попытаются выставить блокировку.
Одна из нитей успешно выставит блокировку, другие же нити напечатают сообщение "Секция закрыта**...**", приостановят работу на две секунды с помощью функции **sleep()**, а после снова будут пытаться установить блокировку. Нить, которая установила блокировку, должна снять её с помощью функции **omp_unset_lock()**. Таким образом, код, находящийся между функциями установки и снятия блокировки, будет выполнен каждой нитью по очереди. В данном случае, все нити по очереди выведут сообщения "Начало закрытой секции**...**" и "Конец закрытой секции**...**", но при этом между двумя сообщениями от одной нити могут встретиться сообщения от других нитей о неудачной попытке войти в закрытую секцию. В конце с помощью функции **omp_destroy_lock()** происходит освобождение переменной **lock**.

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

```
omp_lock_t lock;
int n;
omp_init_lock(&lock);
#pragma omp parallel private (n)
{
n=omp_get_thread_num();
while (!omp_test_lock (&lock))
{
printf("Секция закрыта, нить %d\n", n);
sleep(2);
}
printf("Начало закрытой секции, нить %d\n", n);
sleep(5);
printf("Конец закрытой секции, нить %d\n", n);
omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
}
```

Пример *29.* Функция *omp_test_lock()* на языке Си.

Использование замков является наиболее гибким механизмом синхронизации, поскольку с помощью замков можно реализовать все остальные варианты синхронизации.

Директива ***flush***

Поскольку в современных параллельных вычислительных системах может использоваться сложная структура и иерархия памяти, пользователь должен иметь гарантии того, что в необходимые ему моменты времени все нити будут видеть единый согласованный образ памяти. Именно для этих целей и предназначена директива **flush**.

Си:

#pragma omp flush [(список**)]**

Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельных переменных, то эти переменные можно явно перечислить в директиве списком. До полного завершения операции никакие действия с перечисленными в ней переменными не могут начаться.

Неявно **flush** без параметров присутствует в директиве **barrier**, на входе и выходе областей действия директив **parallel**, **critical**, **ordered**, на выходе областей распределения работ, если не используется опция **nowait**, в вызовах функций **omp_set_lock()**, **omp_unset_lock()**, **omp_test_lock()**, **omp_set_nest_lock()**, **omp_unset_nest_lock()**, **omp_test_nest_lock()**, если при этом замок устанавливается или снимается, а также перед порождением и после завершения любой задачи (**task**). Кроме того, **flush** вызывается для переменной, участвующей в операции, ассоциированной с директивой **atomic**. Заметим, что **flush** не применяется на входе области распределения работ, а также на входе и выходе области действия директивы **master**.

Задания

1) Что произойдёт, если барьер встретится не во всех нитях, исполняющих текущую параллельную область?

2) Могут ли две нити одновременно находиться в различных критических секциях?

3) В чём заключается разница в использовании критических секций и директивы **atomic**?

4) Смоделируйте при помощи механизма замков:

- барьерную синхронизацию;
- критическую секцию.

5) Когда возникает необходимость в использовании директивы **flush**?

6) Реализуйте параллельный алгоритм метода Гаусса решения систем линейных алгебраических уравнений. Выберите оптимальные варианты распараллеливания и проведите анализ эффективности реализации.