OpenMP

Директива `sections` определяет набор независимых секций кода, каждая из которых
выполняется своей нитью.

```
#pragma omp sections [опция…]
```

Список опций:

`private(список)`

`firstprivate(список)`

`lastprivate(список)` – переменным присваивается результат из последней секции

`reduction(оператор:список)`

`nowait`

OpenMP

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

`#pragma omp section`
Перед первым участком кода в блоке sections

директива `section` не обязательна. Какие 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.

OpenMP

```
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);
 }
 printf("Параллельная область, процесс %d\n", n);
}
```

OpenMP

```
int n=0;
 #pragma omp parallel
 {
   #pragma omp sections lastprivate(n)
   {
    #pragma omp section
    n=1;
    #pragma omp section
     n=2;
   }
   printf("Значение n на нити %d: %d\n",
            omp_get_thread_num(), n);
}
printf("Значение n в конце: %d\n", n);
```

OpenMP

Директива `task` применяется для выделения отдельной независимой задачи.

```
#pragma omp task [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.

OpenMP

`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 (shared|none)
private (список)
firstprivate (список)
shared (список)
```

OpenMP

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

```
#pragma omp taskwait
```

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

Синхронизации в OpenMP

Самый распространенный способ синхронизации в `OpenMP` – барьер. Он оформляется с помощью директивы `barrier`.

```
#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`).

Синхронизации в OpenMP

```
#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");
   }
}
```

Синхронизации в OpenMP

Директива `ordered` определяет блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле.

```
#pragma omp ordered
```

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.

Синхронизации в OpenMP

```
#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);
          }
      }
    }
}
```

Синхронизации в OpenMP

С помощью директивы `critical` создаётся критическая секция программы.

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

В каждый момент времени в критической секции может находиться не более одной нити. Все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение. Как только работавшая нить выйдет из критической секции, одна из заблокированных нитей войдет в неё. Если на входе стояло несколько нитей, то случайным образом выбирается одна из них, а остальные продолжают ожидание.

Синхронизации в OpenMP

Критические секции могут быть именованными или не именованными. В различных ситуациях улучшает производительность. Согласно стандарту все критические секции без имени, будут ассоциированы одним именем. Присвоение имени позволит вам одновременно выполнять две и более критические секции.

```
 int n;
#pragma omp parallel
  {
    #pragma omp critical
    {
    n=omp_get_thread_num();
    printf("Нить %d\n", n);
    }
  }
}
```

Синхронизации в OpenMP

```
#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.

Синхронизации в OpenMP

```
#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"
     , count);
}
```

Синхронизации в OpenMP

Один из вариантов синхронизации в `OpenMP` реализуется через механизм замков (`locks`). В качестве замков используются общие переменные. 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.

Синхронизации в OpenMP

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.

Синхронизации в OpenMP

Для инициализации простого или множественного замка используются соответственно функции `omp_init_lock()` и `omp_init_nest_lock()`.

```
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.

Синхронизации в OpenMP

Для захватывания замка используются функции `omp_set_lock()` и `omp_set_nest_lock()`.

```
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.

Синхронизации в OpenMP

Для освобождения замка используются функции
`omp_unset_lock()` и
`omp_unset_nest_lock()`.

```
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 есть нити, заблокированные на операции, захватывающей данный замок, он будет сразу захвачен одной из ожидающих нитей.

Синхронизации в OpenMP

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

Синхронизации в OpenMP

Для неблокирующей попытки захвата замка используются функции `omp_test_lock()` и `omp_test_nest_lock()`.

```
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.

Синхронизации в OpenMP

```
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, %d\n", n);
     sleep(2);
  }
  printf("Начало закрытой секции, %d\n", n);
  sleep(5);
  printf("Конец закрытой секции, %d\n", n);
  omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
```