Параллельные секции

Директива *sections* используется для задания не итеративного параллелизма. Эта директива определяет набор независимых секций кода, каждая из которых выполняется своим потоком.

Необходимость в использовании таких конструкций возникает, если в программе есть объемные (по тексту) вычисления, не охваченные общим заголовком цикла.

Синтаксис:
#pragma omp sections [опция [[,] опция]...]
<структурный блок>

Опции директивы sections
private(список переменных *)*
firstprivate(список переменных *)*
lastprivate(список переменных *)*
reduction(оператор : список переменных *)*
nowait

Директива *section*

#pragma omp section
<structured block>

Перед первым участком кода в блоке *sections* директива *section* не обязательна (однако так делать не рекомендуется).

Неизвестно, как будут распределены потоки между секциями (первая секция не обязательно будет выполняться потоком номер 0 или потоком номер 1, …).

Если количество потоков больше количества секций, то часть потоков (какая именно – заранее не определено) для выполнения данного блока секций не будет задействована.

Если наоборот, потоков меньше, чем секций, то некоторым (или всем) потокам достанется выполнение более, чем по одной секции. Пример низкоуровневого распараллеливания:

```
#pragma omp sections firstprivate( alfa, beta ) shared( n )
{
#pragma omp section
{
  for (int i = 1; i < n; ++i)  alfa[ i ] = alfa[ i-1 ] + 1;
}
#pragma omp section
{
  for (int i = 1; i < n; ++i)  beta[ i ] = beta[ i-1 ] - 1;
}
…
```

Параллельные циклы. Директива for

Синтаксис:

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

<структурный блок>

Эта директива относится к следующему за данной директивой блоку, включающему оператор цикла *for*, который должен удовлетворять ряду ограничений.

Главное ограничение состоит в том, что заголовок цикла должен выглядеть так:

for([целочисленный тип] i = инвариант цикла;

i { <,>,=,<=,>= } инвариант цикла;

i { +,- }= инвариант цикла)

Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива воздействует только на самый внешний цикл.

Дополнительные требования к распараллеливаемому циклу

Кроме того, должны выполняться следующие требования:

1. Корректная программа не должна зависеть от того, какой именно поток какую именно итерацию параллельного цикла выполнит.
2. Нельзя использовать побочный выход из параллельного цикла (операторы *goto*, *break*, *continue*).
3. Размер блока итераций, указанный в опции *schedule* директивы *for* (если эта опция указана), не должен изменяться в рамках цикла.
4. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в том случае, если она специфицирована в качестве общей переменной, то ***неявно делается локальной*** при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции *lastprivate* директивы for.

Все эти ограничения введены для того, чтобы компилятор OpenMP мог обеспечить точное определение количества итераций в момент входа в цикл и последующее их распределение между потоками параллельного региона.

Пример неверного использования директивы for

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define ARRAYSIZE …
int main( ) {                    // пузырьковая сортировка
      int i, j, tmp, arrayX[ ARRAYSIZE ] ;
      for( i = 0; i < ARRAYSIZE; i += 1) {
            arrayX[ i ] = rand( );
      }
//директивы parallel и for могут быть записаны вместе:
#pragma omp parallel for          //для этого цикла неверно!
      for( i = 0; i < ARRAYSIZE; i += 1 ) {
//этот вариант тоже неверен:#pragma omp parallel for здесь
            for( j = i+1; j < ARRAYSIZE; j++)
                  if( arrayX[ j ] < arrayX[ i ] ) {
                        tmp = arrayX[ i ];
                        arrayX[ i ] = arrayX[ j ];
                        arrayX[ j ] = tmp;
                  }
      }
      …
}
```

Результаты работы неверной программы пузырьковой сортировки

int arrayX[ ] =
{3, 82, 17, 33, 82, 91, 4, 65, 7, 88, 41, 6, 55, 32, 5, 8, 27, 73, 56, 39, 74, 15, 44, 15, 93, 44, 26, 84, 63, 3, 9, 42, 74};

При количестве потоков 1 или 2 (все хорошо):
3, 3, 4, 5, 6, 7, 8, 9, 15, 15, 17, 26, 27, 32, 33, 39, 41, 42, 44, 44, 55, 56, 63, 65, 73, 74, 74, 82, 82, 84, 88, 91, 93

При количестве потоков 4, первый запуск:
3, 3, 3, 15, 4, 8, 26, 4, 5, 6, 7, 8, 8, 9, 15, 15, 17, 88, 65, 55, 42, 41, 32, 27, 82, 63, 63, 73, 74, 74, 84, 93, 93
второй запуск:
3, 9, 4, 3, 33, 3, 3, 74, 15, 5, 5, 6, 7, 7, 15, 15, 17, 88, 73, 65, 55, 42, 74, 41, 32, 27, 26, 39, 42, 44, 63, 84, 93

Опции директивы *for*

- *private*(список переменных)
- *firstprivate*(список переменных)
- *lastprivate*(список переменных)
- *reduction*(оператор : список переменных)
- *nowait*
- *schedule*(*type*[, *chunk*])
- *ordered*
- *collapse*(*n*)

Опция schedule

schedule(type[, chunk])

Задаёт способ распределения итераций цикла по потокам.

type = *static* – блочно-циклическое распределение итераций цикла с размером блока *chunk*. Первый блок из *chunk* итераций выполняет мастер-поток, второй блок — поток с номером 1 и т.д. до последнего потока, затем распределение снова начинается с нулевого потока. 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 потоками. Примеры: *schedule(static, 16)*, *schedule(static)*

Пусть есть цикл *for(int i =0; i < 256; i++) { … }*

#pragma omp for num_threads(4) shedule(static, 16)

Поток 0 выполнит итерации 0-15, 64-79, 128-143, 192-207.

Поток 1 выполнит итерации 16-31, 80-95, 144-159, 208-223.

…

#pragma omp for num_threads(4) shedule(static)

Поток 0 выполнит итерации 0-63

Поток 1 выполнит итерации 64-127 …

Опция schedule

schedule(type[, chunk])

type = *dynamic* – динамическое распределение итераций с фиксированным размером блока: сначала каждый поток получает *chunk* итераций (по умолчанию *chunk* = 1), затем тот поток, который первым заканчивает выполнение своей порции итераций, получает первую оставшуюся порцию из *chunk* итераций и т.д.. Освободившиеся потоки получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные. Примеры: *schedule(dynamic)* или *schedule(dynamic, 4)*

Для того же цикла в первом случае (*schedule(dynamic)*) распределение может оказаться таким:

Поток 0: 0, 6, 8, 13, …
Поток 1: 1, 7, 11, 16, …
Поток 2: 2, 4, 10, 15, …
Поток 3: 3, 5, 9, 12, 14, …

Во втором случае (*schedule(dynamic, 4)*), например, таким:

Поток 0: 0-3, 24-27, 8, 48-51, …
Поток 1: 4-7, 20-23, 32-35, 52-55, …
Поток 2: 8-11, 28-31, 36-39, 44-47, …
Поток 3: 12-15, 16-19, 40-43, 56-59, …

Опция schedule

schedule(type[, chunk])

type = *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 потоков, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку потоков. Примеры: *schedule(guided, 4)* или *schedule(guided).*

Для того же цикла в случае *schedule(guided, 4)* распределение может оказаться таким:

Поток 0: 0-15, 76-87, 121-129, … 244-247
Поток 1: 16-31, 88-99, 130-138, … 252-255
Поток 2: 32-47, 64-75, 139-147, … 248-251
Поток 3: 48-63, 100-111, 112-120, … 240-243

В случае *schedule(guided)*, например, таким:

Поток 0: 0-15, 64-75, 121-129, … 247, 249
Поток 1: 16-31, 88-99, 112-120, …254, 255
Поток 2: 32-47, 76-87, 130-138, … 251, 252
Поток 3: 48-63, 100-111, 139-147, … 248, 253

Опция schedule

schedule(type[, chunk])

type = *auto* – способ распределения итераций (*static*, *dynamic* или *guided*) выбирается компилятором и/или системой выполнения. Параметр *chunk* при этом не задаётся.

type = *runtime* – способ распределения итераций выбирается во время работы программы по значению переменной среды OMP_SCHEDULE (которую устанавливает системный администратор, например, так: *"dynamic, 8"*). Параметр *chunk* при этом значении опции тоже не задаётся.

Опции директивы for

ordered

Эта опция, указывает компилятору, что в теле цикла могут встречаться директивы *omp ordered*. С помощью таких директив внутри тела цикла определяют блоки, которые должны выполняться именно в том порядке, в котором итерации выполняются в последовательном цикле;

В версии стандарта 3.0 появилась еще одна опция: *collapse(n)*, которая указывает компилятору, что *n* последовательных тесно вложенных циклов ассоциируется с данной директивой:

```
#pragma omp parallel for collapse( 2 )
for( int i = 0; i < N; i += 1)
    for( int j = 0; j < M; j += 1)
        …
```

Для этих циклов образуется общее пространство итераций, которое делится между потоками:

```
for( int ij = 0; ij < N * M; ij += 1)
    …
```

Если опция *collapse* не задана, то директива *for* относится только к одному непосредственно следующему за ней циклу.

Параллельная пузырьковая сортировка

Алгоритм пузырьковой сортировки в прямом виде не может быть распараллелен, потому что на каждой последующей итерации цикла используется результат предыдущей. Для параллельной сортировки обычно используется модификация, известная в литературе как метод чет-нечетной перестановки (odd-even transposition).

Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода – в зависимости от четности или нечетности номера итерации сортировки для обработки берутся элементы с четными или нечетными индексами соответственно, их сравнение всегда осуществляется с правыми соседними элементами массива.

Таким образом, на всех нечетных итерациях сравниваются пары

(1, 2), (3, 4), ..., (n-1,n) при нечетном n,

а на четных итерациях обрабатываются элементы

(0, 1), (2, 3), ..., (n-2,n-1).

После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

Параллельная пузырьковая сортировка

```
#define compExch( x, y ) if( x > y ){ tmp=x; x=y; y=tmp;}
#define SIZE …
double arrayX[ SIZE ]  = … ;
int tmp;
…
for ( int i = 0; i < SIZE; i++ ) {
     if ( i % 2 == 0) // четная итерация
     {
#pragma omp parallel for private( tmp )
          for ( int j = 0; j < SIZE - 1; j += 2 )      // (0,1) (2,3) (4,5)…
               compExch( arrayX[ j ], arrayX[ j + 1 ] );
     }
     else          // нечетная итерация
     {
#pragma omp parallel for private( tmp )
          for ( int j = 1; j < SIZE - 1; j += 2 )      // (1,2) (3,4) (5,6)
          compExch(arrayX[  j ], arrayX[ j + 1 ] );
     }
}
```

Задачи (tasks)
только начиная с версии 3.0

Синтаксис:
#pragma omp task [опция [[,] опция]...]
<structured block>

Начиная с 3-й версии OpenMP, любой параллелизм (это директивы *parallel*, *parallel for*, *sections*) реализуется в виде «задач», назначаемых потокам исполняющей системой OpenMP. С помощью директивы *task* любой структурный блок внутри параллельного региона можно объявить как задачу, которая добавляется к списку задач этого региона.

Опции:
if(скалярное выражение *)*
private(список переменных *)*
firstprivate(список переменных *)*
shared(список переменных *)*
default (…)
nowait
untied - означает, что в случае откладывания задача может быть продолжена любым потоком из тех, которые выполняют данную параллельную область. Если эта опция не указана, то задача может быть выполнена только породившим ее потоком.

Пример использования директивы task

```
struct node
{
    struct node *left;
    struct node *right;
    dataType data;
};

extern void process( struct node * );

void traverse( struct node *p )
{
    if (p->left)
#pragma omp task firstprivate(p) untied
            traverse(p->left);

    if (p->right)
#pragma omp task firstprivate(p) untied
            traverse(p->right);
#pragma omp taskwait
    process(p);    //обработка данных data текущего узла
}
```

Директивы taskwait и taskyeld

#pragma omp taskwait

Поток, выполнивший эту директиву, приостанавливается до тех пор, пока не будут завершены все ранее запущенные им независимые задачи.

Например, как написано в примере, эта директива присутствует перед вызовом функции *process*. Это гарантирует, что и левое и правое поддерево будут полностью обработаны до обработки корня (но последовательность «левое-правое» или «правое-левое» останется непредсказуемой)

#pragma omp taskyield

Директива taskyield указывает, что текущая задача может быть приостановлена в пользу выполнения

Группа директив синхронизации

Синхронизация работы потоков в OpenMP может выполняться и явно и неявно.

Неявная синхронизация выполняется в конце любой параллельной области при условии, что в момент создания этой области не была указана опция *nowait*.

В некоторых случаях программисту может потребоваться явно указать необходимость остановки всех или некоторых потоков программы вплоть до момента наступления ожидаемого события.

Барьерная синхронизация

Самый распространенный способ синхронизации в OpenMP – барьерный.

Синтаксис:

```
#pragma omp barrier
```

Критические секции

Синтаксис:
#pragma omp critical [(<имя_критической_секции>)]
<структурный блок>

В каждый момент времени в одной критической секции может находиться не более одного потока.

Если критическая секция уже выполняется каким-либо потоком, то все другие потоки, дошедшие до выполнения этой директивы для секции с данным именем, будут заблокированы, пока вошедший в секцию поток не закончит выполнение структурного блока. Как только этот поток выйдет из критической секции, один из ожидающих потоков войдет в неё. Если входа в критическую секцию ожидало несколько потоков, то выбирается один из них (неважно, как). Остальные будут по-прежнему заблокированы.

Все неименованные секции рассматриваются как одна, даже если они находятся в разных параллельных регионах.

Точно так же, все критические секции, имеющие одно и тоже имя, рассматриваются как одна секция, даже если они находятся в разных параллельных регионах.

Побочные входы и выходы из критической секции (ее структурного блока) запрещены.