Модель данных

Модель данных

Модель данных в OpenMP предполагает наличие как *общей* для всех нитей

области памяти, так и *локальной области* памяти для каждой нити. В параллельных областях переменные программы разделяются на два основных класса:

- **shared - все нити видят одну и ту же переменную;**
- **private - каждая нить видит свой экземпляр данной переменной.**

***Общая переменная* всегда существует лишь в одном экземпляре для всей области действия и доступна всем нитям под одним и тем же именем. Объявление *локальной переменной* вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждой нити. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях.**

Если *несколько переменных* одновременно *записывают* значение общей переменной *без выполнения синхронизации* или если как минимум *одна нить читает* значение общей переменной и как минимум *одна нить записывает* значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой *«гонки данных»* (data race), при которой результат выполнения программы *непредсказуем*.

По умолчанию*, все переменные, *порождённые вне параллельной

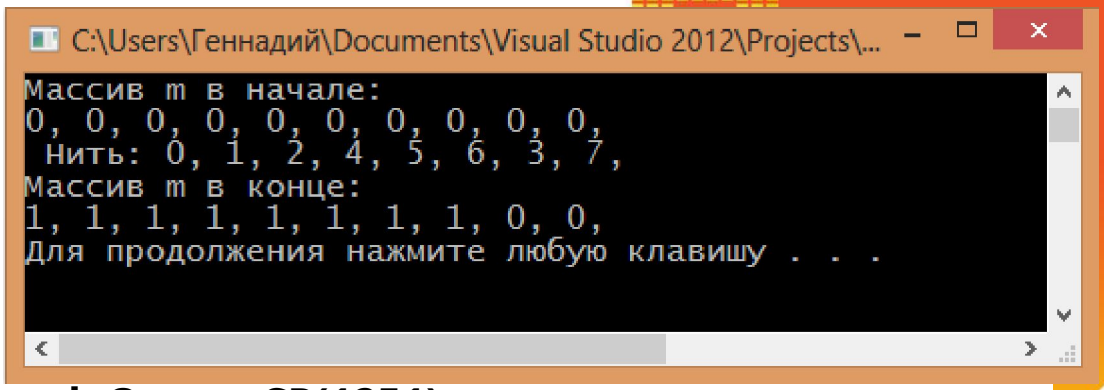
Пример 12

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251); SetConsoleOutputCP(1251);
        int n=1;
   printf("n в последовательной области (начало): %d\n", n);
#pragma omp parallel private(n)
   {    int n=2;
        printf("Значение n на нити (на входе): %d\n", n);
        n=omp_get_thread_num();
        printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("n в последовательной области (конец): %d\n", n);
system("Pause");
}
```

Результаты выполнения

n в последовательной области (начало): 1
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 0
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 2
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 5
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 3
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 1
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 7
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 4
Значение n на нити (на входе): 2
Значение n на нити (на выходе): 6
n в последовательной области (конец): 1

Пример 13




```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);          SetConsoleOutputCP(1251);
    int i, m[10];
    cout<<"Массив m в начале:\n";
for (i=0; i<10; i++){      m[i]=0;       cout<<m[i]<<", ";   }
    cout<<"\n Нить: ";
#pragma omp parallel shared(m)
  {   m[omp_get_thread_num()]=1;
   cout<< omp_get_thread_num()<< ", ";    }    cout<<"\n";
   cout<<"Массив m в конце:\n";
  for (i=0; i<10; i++)  cout<<  m[i]  <<", ";        cout<<"\n";
system("Pause");
}
```

Статические и динамические переменные

В языке Си статические (static) переменные, определённые в параллельной области программы, являются общими (shared). Динамически выделенная память также является общей, однако указатель на неё может быть как общим, так и локальным. Отдельные правила определяют назначение классов переменных при входе и выходе из параллельной области или параллельного цикла при использовании опций reduction, firstprivate, lastprivate, copyin

Пример 14

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);          SetConsoleOutputCP(1251);
  int n=1;
   printf("Значение n в начале: %d\n", n);
#pragma omp parallel firstprivate(n)
   {
     printf("Значение n на нити (на входе): %d\n", n);
          n=omp_get_thread_num();
     printf("Значение n на нити (на выходе): %d\n", n);
   }
   printf("Значение n в конце: %d\n", n);
system("Pause");   }
```

Результат выполнения примера 14

Значение n в начале: 1
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 0
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 2
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 5
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 1
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 3
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 4
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 6
Значение n на нити (на входе): 1
Значение n на нити (на выходе): 7
Значение n в конце: 1

Директива threadprivate

Директива threadprivate указывает, что переменные из списка должны быть размножены с тем, чтобы каждая нить имела свою локальную копию:

#pragma omp threadprivate(список)

Директива threadprivate может позволить сделать локальные копии для статических переменных, которые по умолчанию являются общими. Для корректного использования локальных копий глобальных объектов нужно гарантировать, что они используются в разных частях программы одними и теми же нитями. Если на локальные копии ссылаются в разных параллельных областях, то для сохранения их значений необходимо, чтобы не было объемлющих параллельных областей, количество нитей в обеих областях совпадало, а переменная OMP_DYNAMIC была установлена в false с начала первой области до начала второй. Переменные, объявленные как threadprivate, не могут использоваться в опциях директив OpenMP, кроме copyin, copyprivate, schedule, num_threads, if.

Пример 15

```
#include "stdafx.h
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[]){  SetConsoleCP(1251);     SetConsoleOutputCP (1251);
  int num;  n=1;
#pragma omp parallel private (num)
  {   num=omp_get_thread_num();
     printf("Значение n на нити %d (на входе): %d\n", num, n);
n=omp_get_thread_num();
     printf("Значение n на нити %d (на выходе): %d\n", num, n);     }
   printf("Значение n (середина): %d\n", n);
#pragma omp parallel private (num)
  {          num=omp_get_thread_num();
     printf("Значение n на нити %d (ещё раз): %d\n", num, n);     }
system("Pause");  }
```

Результаты выполнения примера 15

Значение n на нити 0 (на входе): 1
Значение n на нити 0 (на выходе): 0
Значение n на нити 6 (на входе): 0
Значение n на нити 6 (на выходе): 6
Значение n на нити 4 (на входе): 0
Значение n на нити 4 (на выходе): 4
Значение n на нити 1 (на входе): 0
Значение n на нити 1 (на выходе): 1
Значение n на нити 5 (на входе): 0
Значение n на нити 5 (на выходе): 5
Значение n на нити 2 (на входе): 0
Значение n на нити 2 (на выходе): 2
Значение n на нити 3 (на входе): 0
Значение n на нити 3 (на выходе): 3
Значение n на нити 7 (на входе): 0
Значение n на нити 7 (на выходе): 7
Значение n (середина): 0
Значение n на нити 7 (ещё раз): 7
Значение n на нити 0 (ещё раз): 0

Если необходимо переменную, объявленную как threadprivate, инициализировать значением размножаемой переменной из нити-мастера, то на входе в параллельную область можно использовать опцию copyin. Если значение локальной переменной или переменной, объявленной как threadprivate, необходимо переслать от одной нити всем, работающим в данной параллельной области, для этого можно использовать опцию copyprivate директивы single.

Пример 16

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int n;
#pragma omp threadprivate(n)
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
    n=1;
#pragma omp parallel copyin(n)
  {
 printf("Значение n: %d (нить %d)  \n", n, omp_get_thread_num());
}

system("Pause");
}
```

Результаты выполненимя примера 16

Значение n: 1 (нить 1)
Значение n: 1 (нить 4)
Значение n: 1 (нить 2)
Значение n: 1 (нить 3)
Значение n: 1 (нить 6)
Значение n: 1 (нить 5)
Значение n: 1 (нить 7)
Значение n: 1 (нить 0)
Для продолжения нажмите любую клавишу . . .

Распределение работы

- Низкоуровневое распараллеливание
- Параллельные циклы
- Параллельные секции
- Директива workshare
- Задачи (tasks)

Распределение работы

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

Низкоуровневое распараллеливание Все нити в параллельной области нумеруются последовательными целыми числами от 0 до N-1, где N — количество нитей, выполняющих данную область.

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:

```
int omp_get_thread_num(void);
```

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

```
int omp_get_num_threads(void);
```

Пример 17

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
  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);
  }
system("Pause");
}
```

```
Всего нитей: 8
Нить номер 1
Нить номер 4
Нить номер 5
Нить номер 3
Нить номер 6
Нить номер 2
Нить номер 7
Для продолжения нажмите любую клавишу . . .
```

Использование omp_get_thread_num() и omp_get_num_threads()

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 :

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

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

Опции директивы for :

- **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 это: +, *, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;**

Опции директивы for (продолжение):

- **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 происходит 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.

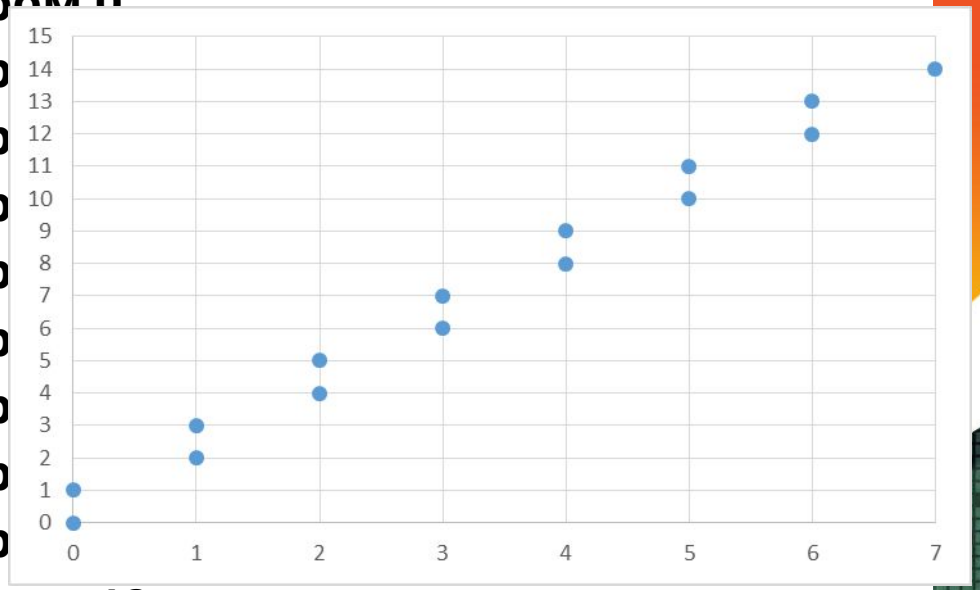
Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции

Пример 18

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
#define N 15
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
 int A[N], B[N], C[N], i, n;
for (i=0; i<N; i++){ A[i]=i; B[i]=2*i; C[i]=0; }
#pragma omp parallel shared(A, B, C) private(i, n)
  { n=omp_get_thread_num();
#pragma omp for
     for (i=0; i<N; i++)      {         C[i]=A[i]+B[i];
       printf("Нить %d сложила элементы с номером %d\n", n, i);
}   }
  cout<<"массив A: "; for ( i = 0; i < N; i++)  { cout << A[i] <<", " ; }
cout<<"\n";
  cout<<"массив B: "; for ( i = 0; i < N; i++)  { cout << B[i] <<", " ; }
cout<<"\n";
```

Результаты выполнения примера 18

Нить 0 сложила элементы с номером 0
Нить 0 сложила элементы с номер
Нить 2 сложила элементы с номер
Нить 2 сложила элементы с номер
Нить 1 сложила элементы с номер
Нить 1 сложила элементы с номер
Нить 5 сложила элементы с номер
Нить 5 сложила элементы с номер
Нить 7 сложила элементы с номер
Нить 6 сложила элементы с номером 12
Нить 6 сложила элементы с номером 13
Нить 4 сложила элементы с номером 8
Нить 4 сложила элементы с номером 9
Нить 3 сложила элементы с номером 6
Нить 3 сложила элементы с номером 7
массив A: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
массив B: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
массив C: 0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42,



Опция schedule

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

Опция schedule (продолжение)

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

Пример 19

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);      SetConsoleOutputCP(1251);
   int i;
#pragma omp parallel private(i)
  {
#pragma omp for schedule (static)
for (i=0; i<15; i++)      {
        printf("Нить %d выполнила итерацию %d\n",
                                omp_get_thread_num(), i);
        Sleep(1);      }   }

system("Pause"); }
```

Результаты выполнения примера 19

#pragma omp for schedule (static)

Нить 0 выполнила итерацию 0
Нить 2 выполнила итерацию 4
Нить 1 выполнила итерацию 2
Нить 3 выполнила итерацию 6
Нить 6 выполнила итерацию 12
Нить 5 выполнила итерацию 10
Нить 7 выполнила итерацию 14
Нить 4 выполнила итерацию 8
Нить 1 выполнила итерацию 3
Нить 2 выполнила итерацию 5
Нить 3 выполнила итерацию 7
Нить 0 выполнила итерацию 1
Нить 5 выполнила итерацию 11
Нить 6 выполнила итерацию 13
Нить 4 выполнила итерацию 9
Для продолжения нажмите любую клавишу . . .

Результаты выполнения примера 19

#pragma omp for schedule (static, 1)

Нить 0 выполнила итерацию 0
Нить 1 выполнила итерацию 1
Нить 2 выполнила итерацию 2
Нить 6 выполнила итерацию 6
Нить 5 выполнила итерацию 5
Нить 4 выполнила итерацию 4
Нить 3 выполнила итерацию 3
Нить 7 выполнила итерацию 7
Нить 0 выполнила итерацию 8
Нить 3 выполнила итерацию 11
Нить 4 выполнила итерацию 12
Нить 2 выполнила итерацию 10
Нить 6 выполнила итерацию 14
Нить 5 выполнила итерацию 13
Нить 1 выполнила итерацию 9
Для продолжения нажмите любую клавишу . . .

#pragma omp for schedule (static, 2)

Результаты выполнения примера 19

#pragma omp for schedule (dynamic)

Нить 0 выполнила итерацию 0
Нить 2 выполнила итерацию 1
Нить 4 выполнила итерацию 2
Нить 5 выполнила итерацию 3
Нить 3 выполнила итерацию 4
Нить 1 выполнила итерацию 5
Нить 6 выполнила итерацию 6
Нить 7 выполнила итерацию 7
Нить 2 выполнила итерацию 8
Нить 5 выполнила итерацию 9
Нить 4 выполнила итерацию 10
Нить 0 выполнила итерацию 11
Нить 6 выполнила итерацию 12
Нить 1 выполнила итерацию 14
Нить 3 выполнила итерацию 13
Для продолжения нажмите любую клавишу . . .

#pragma omp for schedule (dynamic, 2)

Результаты выполнения примера 19

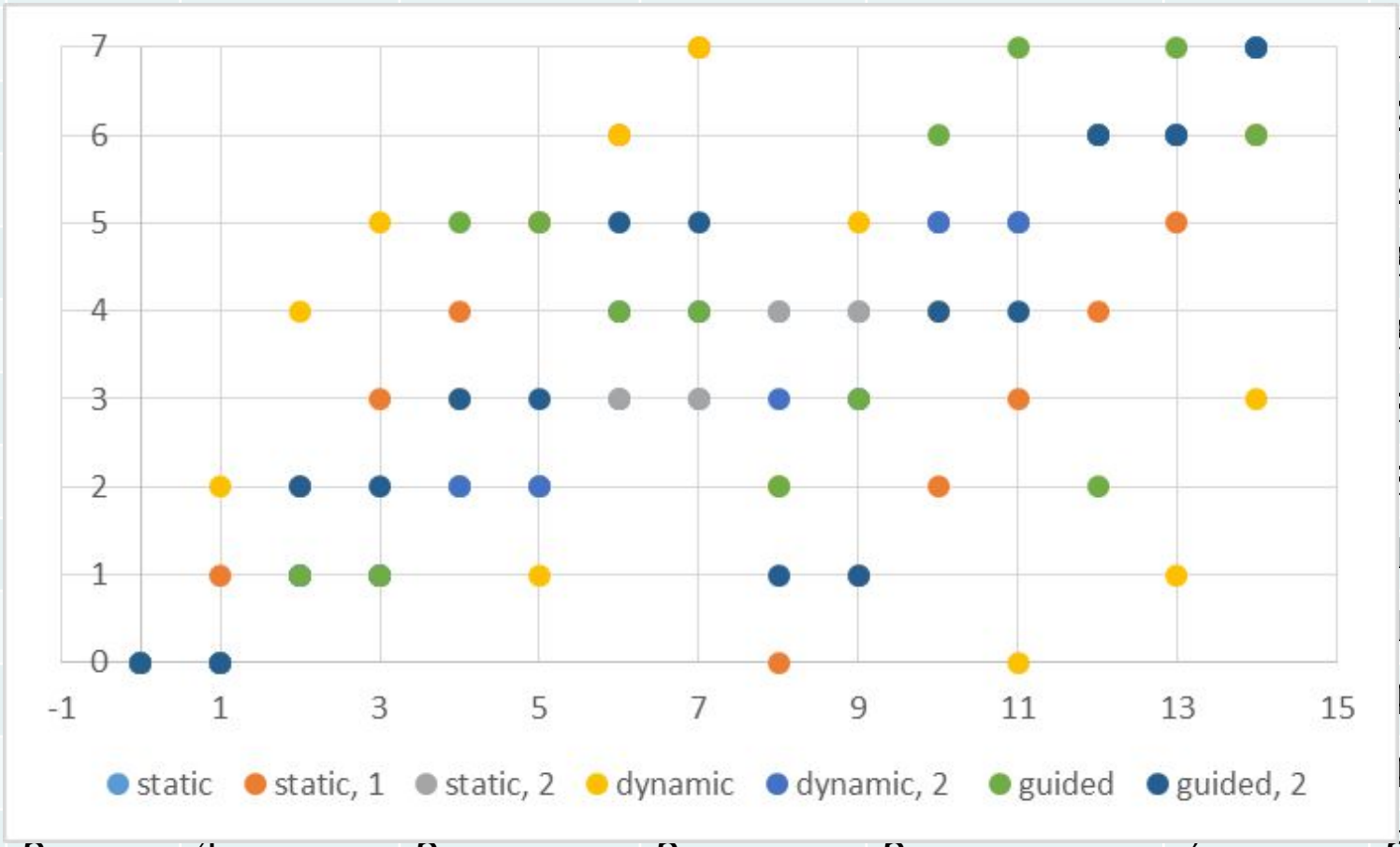
#pragma omp for schedule (guided)

Нить 0 выполнила итерацию 0
Нить 1 выполнила итерацию 2
Нить 5 выполнила итерацию 4
Нить 4 выполнила итерацию 6
Нить 2 выполнила итерацию 8
Нить 3 выполнила итерацию 9
Нить 6 выполнила итерацию 10
Нить 7 выполнила итерацию 11
Нить 2 выполнила итерацию 12
Нить 4 выполнила итерацию 7
Нить 5 выполнила итерацию 5
Нить 1 выполнила итерацию 3
Нить 0 выполнила итерацию 1
Нить 7 выполнила итерацию 13
Нить 6 выполнила итерацию 14
Для продолжения нажмите любую клавишу . . .

#pragma omp for schedule (guided, 2)

Результаты выполнения примера 19

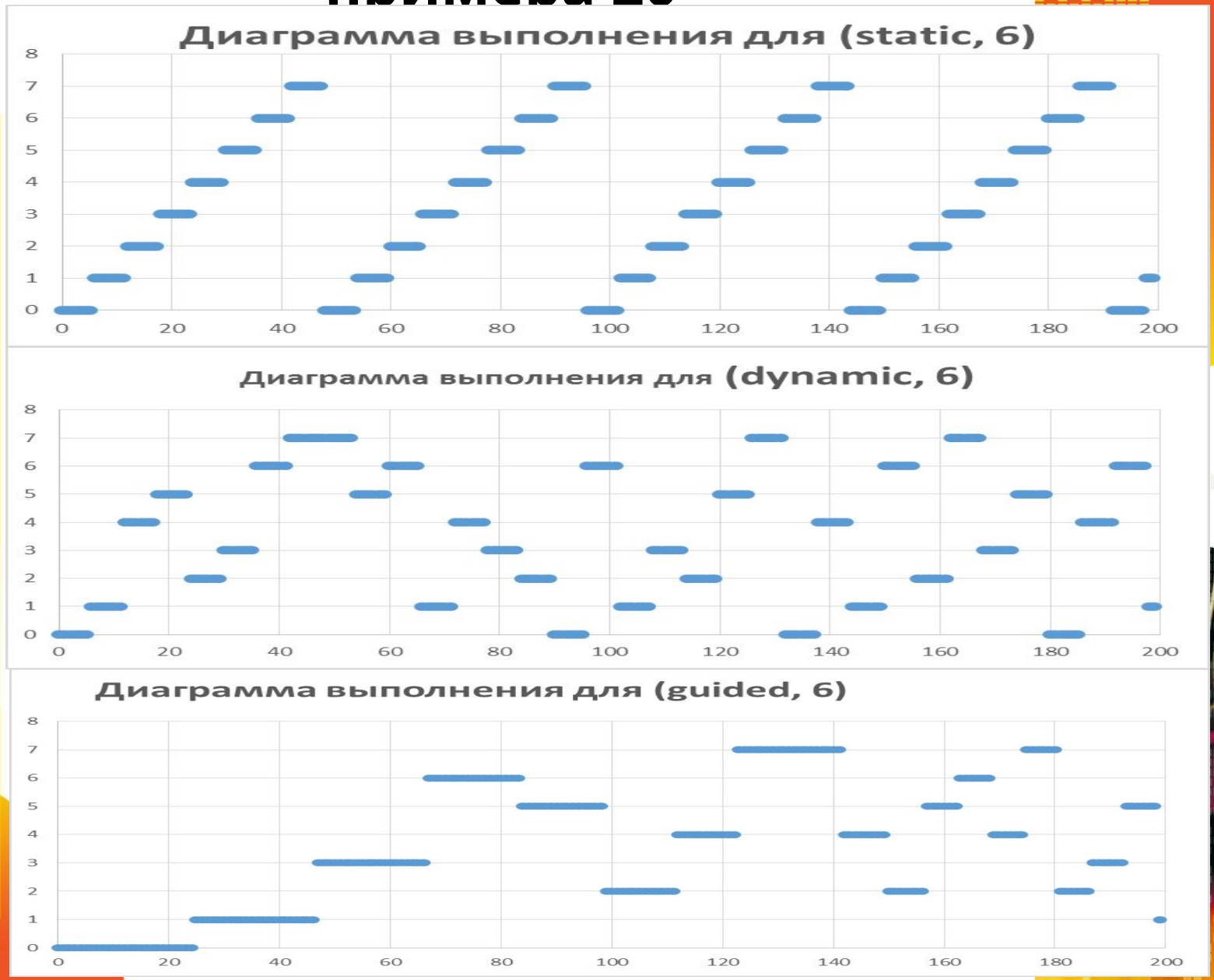
i	static	static, 1	static, 2	dynamic	dynamic, 2	guided	guided, 2
0	0	0	0	0	0	0	0
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12	6	4	6	6	6	2	6
13	6	5	6	1	6	7	6
14	7	6	7	3	7	6	7



Пример 20

```
#include "stdafx.h"
#include <iostream>
#include <windows.h
#include <omp.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);  SetConsoleOutputCP(1251);
    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);      }   }
system("Pause");   }
```

Результаты выполнения примера 20



Переменная OMP_SCHEDULE

Значение по умолчанию переменной OMP_SCHEDULE зависит от реализации. Если переменная задана неправильно, то поведение программы при задании опции runtime также зависит от реализации.

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

void omp_set_schedule(omp_sched_t type, int chunk);

Допустимые значения констант описаны в файле omp.h (omp_lib.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:

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

Параллельные секции

Директива sections используется для задания конечного (неитеративного) параллелизма:

#pragma omp sections [опция [[,] опция]…]

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

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 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.

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, полученный в последней секции;

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 секций выполняется заданный оператор; оператор это: для языка Си – +, *, -, &, |, ^, &&, ||; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

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

Пример 21

```
int main(int argc, char *argv[]){
SetConsoleCP(1251);    SetConsoleOutputCP(1251);
 int n;
#pragma omp parallel private(n)
  {      n=omp_get_thread_num();
#pragma omp sections
     {
#pragma omp section
       {        cout<< "Первая секция, процесс "<< n  <<"\n"; }
#pragma omp section
       { cout<< "Вторая секция, процесс "<< n  <<"\n"; }
#pragma omp section
       {   cout<< "Третья секция, процесс "<< n  <<"\n"; }
     }
    cout<< "Параллельная область, процесс "<< n  <<"\n";
  }
system("Pause"); }
```

Результаты выполнения примера 21

```
Вторая секция, процесс 3
Первая секция, процесс 0
Третья секция, процесс 1
Параллельная область, процесс 1
Параллельная область, процесс 2
Параллельная область, процесс 4
Параллельная область, процесс 0
Параллельная область, процесс 3
Параллельная область, процесс 7
Параллельная область, процесс 5
Параллельная область, процесс 6
Для продолжения нажмите любую клавишу . . .
```

Пример 22

```
int main(int argc, char *argv[]){
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
      int n=0;
#pragma omp parallel
  {
#pragma omp sections lastprivate(n)
    {
#pragma omp section
{ n=1; printf("Первая секция, нить %d: %d\n", omp_get_thread_num(), n);
}
#pragma omp section
{ n=2; printf("Вторая секция, нить %d: %d\n", omp_get_thread_num(), n);
}
#pragma omp section
{ n=3; printf("Третья секция, нить %d: %d\n", omp_get_thread_num(), n);
} }
    printf("Значение n на нити %d: %d\n",    omp_get_thread_num(), n);
}
  printf("Значение n в последовательной области: %d\n", n);
system("Pause");
```

Результаты выполнения примера 22

Первая секция, нить 0: 1
Вторая секция, нить 1: 2
Третья секция, нить 5: 3
Значение n на нити 5: 3
Значение n на нити 0: 3
Значение n на нити 1: 3
Значение n на нити 3: 3
Значение n на нити 7: 3
Значение n на нити 4: 3
Значение n на нити 2: 3
Значение n на нити 6: 3
Значение n в последовательной области: 3
Для продолжения нажмите любую клавишу . . .

Задачи (tasks)

Директива task применяется для выделения отдельной независимой задачи:

#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:

#pragma omp taskwait

Нить, выполнившая данную директиву, приостанавливается до тех пор, пока

не будут завершены все ранее запущенные данной нитью независимые задач

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(shared|none) – всем переменным в задаче, которым явно не назначен класс, будет назначен класс shared; none означает, что всем переменным в задаче класс должен быть назначен явно;**
- **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.**