ДИРЕКТИВЫ OPENMP - **PARALLEL**

Основная директива для создания параллельной области

```
int main ()
{
  //последовательная область, выполняется корневой тред
  . . .
  //Начало параллельной области
  #pragma omp parallel [опции]
  {
  //операторы выполняются всеми тредами
  . . .
  //все треды завершают работу, остается только корневой тред
  }
  //последовательная область, выполняется корневой тред
  . . .
}
```

СИНТАКСИС ДИРЕКТИВЫ **PARALLEL**

```
#pragma omp parallel [опции ...] newline
{

}
if (scalar_expression)
num_threads (integer_expression)
private (list)
firstprivate (list)
shared (list)
default (shared | none)
reduction (operator: list)
copyin (list)
```

ОПЦИЯ IF

`if (scalar_expression)`

– распараллеливание по **условию**.

Если значение выражения ≠ 0,

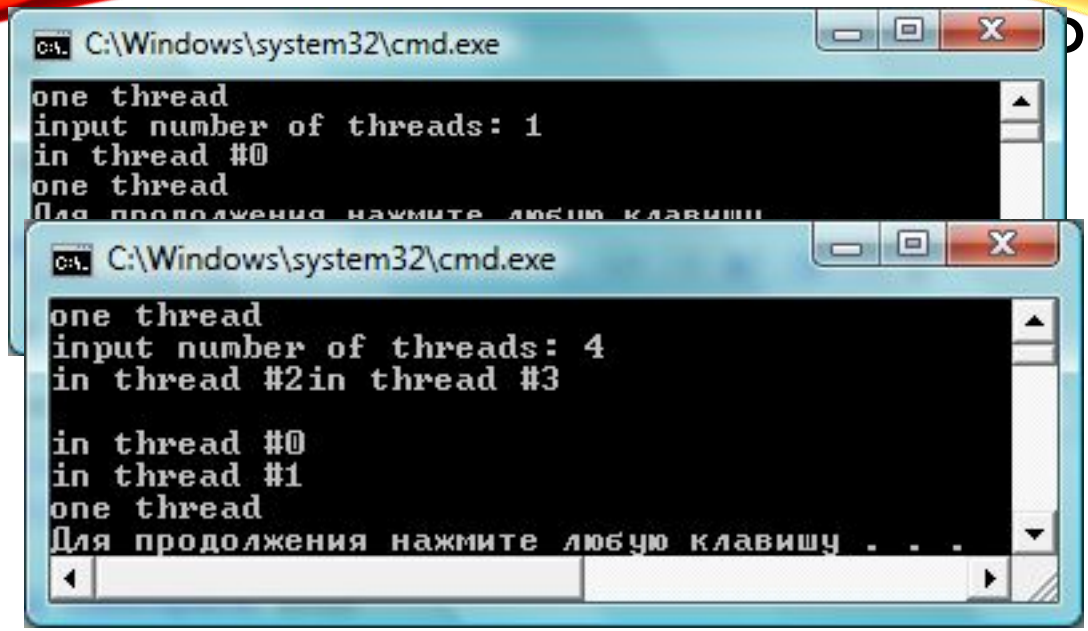
то осуществляется **распараллеливание**.

Иначе операторы

параллельной области выполняются

единственным корневым тредом.

```
#include <iostream>
#include <omp.h>

using namespace std;

int main()
{
   int n;
  cout << "one thread" << endl;
  cout << "input number of threads: ";
  cin >> n;
  omp_set_num_threads(n);
  #pragma omp parallel if( n>1 )
    {
     int k = omp_get_thread_num();
     cout << "in thread #" << k << endl;
    }
  cout << "one thread" << endl;
   return 0;}
```



ОПЦИЯ **NUM_THREADS**

`num_threads (integer_expression)`

– **явное задание количества тредов**, которые будут выполнять операторы параллельной области.

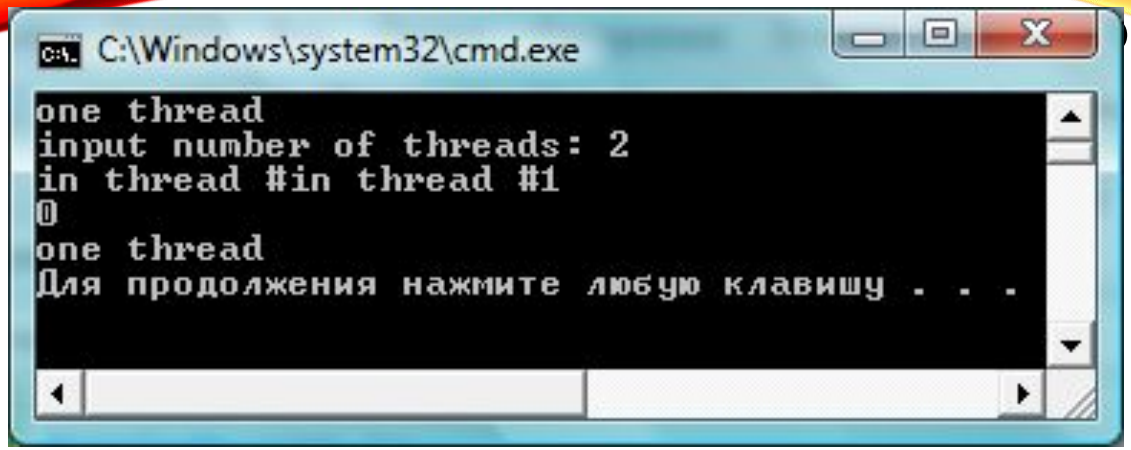
По умолчанию выбирается последнее

значение, установленное функцией

omp_set_num_threads(),

или (если не вызывалась функция) значение переменной

OMP_NUM_THREADS




```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
    int n;
  cout << "one thread" << endl;
  cout << "input number of threads: ";
  cin >> n;
  #pragma omp parallel if( n>1 ) num_threads(n)
     {
    int k = omp_get_thread_num();
    cout << "in thread #" << k << endl;
     }
  cout << "one thread" << endl;
   return 0;
}
```

ЧЕМ ОПРЕДЕЛЯЕТСЯ КОЛИЧЕСТВО ТРЕДОВ?

Количество тредов в параллельной области определяется следующими параметрами **в порядке старшинства**:

- Значением опции **if**
- Значением опции **num_threads**
- Функцией **omp_set_num_threads()**
- Значением переменной окружения **OMP_NUM_THREADS**
- По умолчанию – обычно это число CPU в узле.

ОПЦИИ ДОСТУПНОСТИ ДАННЫХ

- **Данные** –
  - **Разделяемые**, или общие (для всех тредов)
  - **Локальные** (копии в каждом треде).
- **Преимущество** OpenMP – **динамическое** определение количества копий –
  - В одной параллельной области переменная x – **локальная**
  - В другой – **разделяемая**.

ОПЦИЯ **PRIVATE**

`private (list)`

- задаёт список переменных, для которых создается **локальная копия** в каждом треде.
- Переменные должны быть **объявлены до** вхождения в параллельную область.
- Начальное значение локальных копий переменных из списка **не определено** □ задается в параллельной области.

ПРИМЕР

```
float s = 0;
#pragma omp parallel private(s)
{
   s = s + 1;//некорректно
}
```

Значение копий переменной в параллельной области не определено

ОПЦИЯ FIRST**PRIVATE**

`firstprivate (list)`

- задаёт список переменных, для которых создается **локальная копия** в каждом треде.
- Переменные должны быть **объявлены до** вхождения в параллельную область.
- Начальное значение локальных копий переменных из списка определяется их значением в **корневом треде**.

ПРИМЕР

```
float s = 0;
#pragma omp parallel firstprivate(s)
{
   s = s + 1;//корректно
}
```

Значение копий переменной в параллельной области определяется последним значением в последовательной области

ОПЦИЯ SHARED

`shared (list)`

- задаёт список переменных, которые являются **общими** для всех тредов.
- Переменные должны быть **объявлены до** вхождения в параллельную область.
- Все треды могут не только считывать, но и **изменять** их значения корректность использования обеспечивает программист.

ОПЦИЯ DEFAULT

`default (shared|none)`

- **`default (shared)`**
  всем переменным в параллельной области, которым явно не назначена локализация, будет назначена **shared** *(эта опция используется по умолчанию)*
- **`default (none)`**
  всем переменным в параллельной области локализация должна быть назначена явно.

ОПЦИЯ REDUCTION

reduction (operator: list)

operator: +, *, -, &, |, ^, &&, ||

- задаёт **оператор** и **список переменных** (ранее объявленных);
- для **каждой** переменной создаются локальные **копии** в **каждом** треде;
- локальные копии **инициализируются** :
  - для **+ - | ^ ||** – **0** или аналоги,
  - для *** & &&** – **1** или аналоги;
- над локальными копиями переменных **после** выполнения всех операторов параллельной области **выполняется** заданный **оператор**

ПРИМЕР

```
...
int n = 0;
#pragma omp parallel reduction (+: n)
{
   n++;
   cout << "Текущее значение n:”;
   cout << n << endl;
}
cout << "Число тредов: “ << n << endl;
...
```

ДИРЕКТИВЫ OPENMP – **PARALLEL FOR**

Основная директива для распараллеливания вычислений (распределения итераций цикла между тредами)

```
. . .
//Начало параллельной области
#pragma omp parallel for [опции]
{
//должен быть цикл
. . .

}
```

ОГРАНИЧЕНИЯ НА ПАРАЛЛЕЛЬНЫЕ ЦИКЛЫ

- **Результат** программы **не зависит** от того, **какой** именно **тред** выполнит конкретную итерацию цикла.
- **Нельзя** использовать побочный выход (break, goto) из параллельного цикла.
- Размер блока итераций, указанный в опции **schedule**, **не должен изменяться** в рамках цикла.
- **Формат** параллельных циклов:

```
for([int_type] i = инвариант цикла;
i {<,>,=,<=,>=} инвариант цикла;
i {+,-}= инвариант цикла)
```

СИНТАКСИС ДИРЕКТИВЫ **PARALLEL FOR**

```
#pragma omp parallel for[опции ...] newline
{ ...for ...
}
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait
```

СИНТАКСИС ДИРЕКТИВЫ **FOR**

```
#pragma omp for[опции ...] newline
{...for ...
}
```

- Используется **внутри** параллельной области, заданной директивой **parallel**, для указания на распараллеливание конкретного цикла.
- Блок не является обязательным для единственного оператора:

```
#pragma omp for[опции ...] newline
  for ...
```

ПРИМЕР: ВЫЧИСЛЕНИЕ СУММЫ

```
void main ()
{
  int i;
  double ZZ, res=0.0;
  omp_set_num_threads(2)
  #pragma omp parallel for reduction(+:res)
  private(ZZ)
    for (i=0; i< 1000; i++)
    {
       ZZ = func(i);
       res = res + ZZ;
    }
}
```

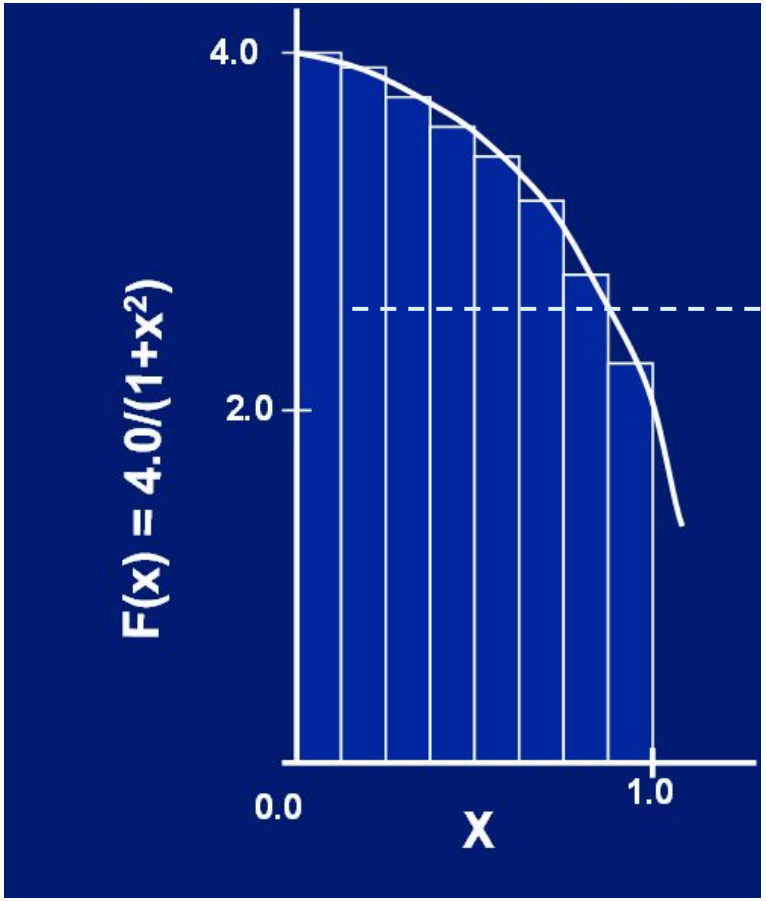
ОЦЕНКА ВРЕМЕНИ ВЫПОЛНЕНИЯ ПОСЛЕДОВАТЕЛЬНОЙ И ПАРАЛЛЕЛЬНОЙ ПРОГРАММ

- Функция `double omp_get_wtime()` возвращает в вызвавшем треде **время** в секундах, прошедшее с некоторого момента в прошлом.
- Если фрагмент кода окружить вызовами функции, то разность возвращаемых значений равна времени выполнения команд данного фрагмента.
- Функция `double omp_get_wtick()` возвращает в вызвавшем треде **разрешение** таймера в секундах.
- Это время можно рассматривать как **меру точности** таймера

ПРИМЕР ЗАМЕРА ВРЕМЕНИ

```
...
double start_time, end_time, tick;
start_time = omp_get_wtime();
...
end_time = omp_get_wtime();
tick = omp_get_wtick();
cout <<"time" << end_time-start_time << endl;
cout << "precision"<< tick<<endl;
```

ПРИМЕР: ВЫЧИСЛЕНИЕ ЧИСЛА π



ПРИМЕР: ВЫЧИСЛЕНИЕ ЧИСЛА π

```
void main ()
{ long num_steps;
  cout << "number of steps = ";
  cin >> num_steps;
  double step = 1./ num_steps;
  double x, pi, sum = 0.0;
  #pragma omp parallel for private(x) reduction(+:sum)
    for (int i = 0; i <= num_steps; i++)
    {   x = i*step;
        sum = sum + 4 /(1 + x*x);
    }
  pi = step * sum;
  int my_precision;
  cout <<"precision = ";
  cin >> my_precision;
  cout.precision(my_precision);//default value = 6
  cout << "pi = " << pi << endl;
}
```

ОПЦИИ ДИРЕКТИВЫ **PARALLEL FOR**

```
#pragma omp parallel for[опции ...] newline
{ ...for ...
}
```

schedule (type [,chunk])

ordered

private (list)

firstprivate (list)

lastprivate (list)

shared (list)

reduction (operator: list)

collapse (n)

nowait

ОПЦИЯ **LASTPRIVATE**

lastprivate (list)

переменным из списка присваивается результат с последней итерации цикла - значение из команд того треда, который бы последним исполнялся последовательно

Пример

```
int i,k;
#pragma omp parallel for private(i) lastprivate(k)
        for(i=0; i<10; i++)
            k = i*i;
// последов. область, i - не определено, k - определено
cout <<"k = " <<  k; // k == 81
```

ОПЦИЯ **SCHEDULE** – УПРАВЛЕНИЕ НАГРУЗКОЙ

`schedule (type [,num_iters])`

- В зависимости от параметров (**`type, num_iters`**) выполнение итераций цикла **распределяется между тредами**.
- По умолчанию **num_iters=1**
- Если опция schedule не указана, то по умолчанию распределение зависит от реализации (CPU, ОС).
- **Возможные значения `type`**
  - **`dynamic`**
  - **`guided`**
  - **`runtime num_iters не задается`**
  - **`static`**

STATIC – СТАТИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ЗАГРУЗКИ ТРЕДОВ:

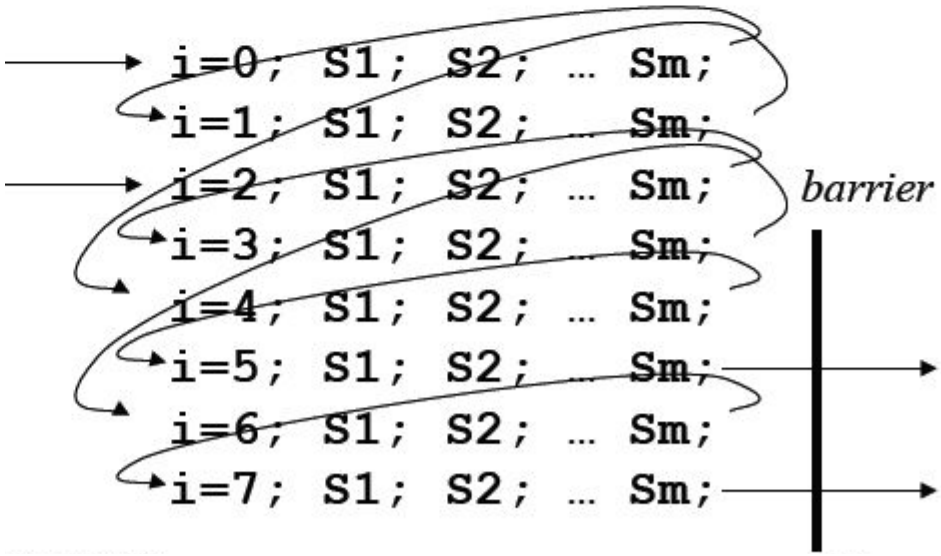
- КАЖДЫЙ ТРЕД (С НУЛЕВОГО) БЕРЕТ ДЛЯ ВЫПОЛНЕНИЯ БЛОК ИЗ <NUM_ITERS> ИТЕРАЦИЙ ЦИКЛА ,

-ОСТАВШИЕСЯ ИТЕРАЦИИ СНОВА ПОСЛЕДОВАТЕЛЬНО РАСПРЕДЕЛЯЮТСЯ ПО ТРЕДАМ, ПОКА НЕ БУДУТ ВЫПОЛНЕНЫ ВСЕ ИТЕРАЦИИ.

ЕСЛИ <NUM_ITERS> НЕ УКАЗАНО, ТО ИТЕРАЦИИ РАВНОМЕРНО РАСПРЕДЕЛЯЮТСЯ МЕЖДУ ТРЕДАМИ.

…

```
#pragma omp for schedule(static,2)
for (i=0; i<k; i++)
{
  S1;
  S2;
  …
  Sm;
}
```



DYNAMIC – ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ЗАГРУЗКИ ТРЕДОВ:

КАЖДЫЙ ТРЕД БЕРЕТ ДЛЯ ВЫПОЛНЕНИЯ БЛОК ИЗ <NUM_ITERS> ИТЕРАЦИЙ ЦИКЛА.

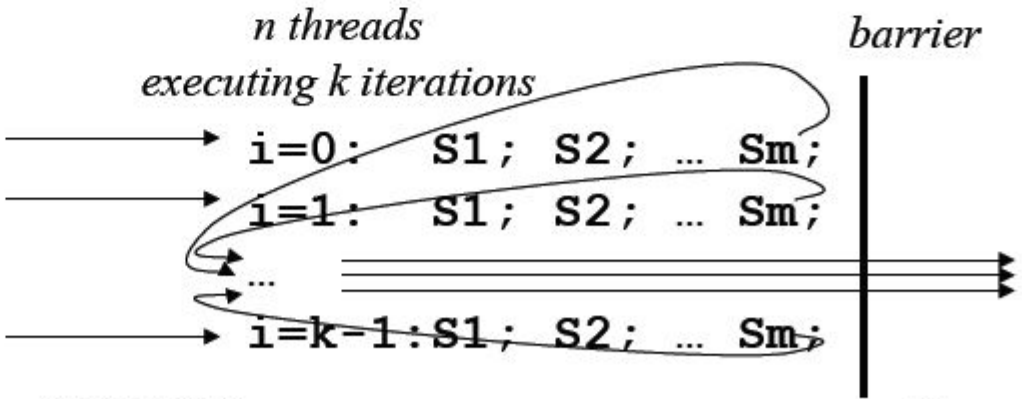
ОСВОБОДИВШИЕСЯ ТРЕДЫ СНОВА БЕРУТ ПО <NUM_ITERS> СЛУЧАЙНЫХ ИТЕРАЦИЙ, ПОКА НЕ БУДУТ ВЫПОЛНЕНЫ ВСЕ ИТЕРАЦИИ

…

```
#pragma omp for schedule(dynamic)
for (i=0; i<k; i++)
{
  S1;
  S2;
  …
  Sm;
}
```

n=3, num_iters=1

When k>n, threads execute randomly chosen loop iterations until all iterations are completed



GUIDED – ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ЗАГРУЗКИ ТРЕДОВ:

- КАЖДЫЙ ТРЕД БЕРЕТ ДЛЯ ВЫПОЛНЕНИЯ N0 (*ЗАВИСИТ ОТ РЕАЛИЗАЦИИ*) ИТЕРАЦИЙ,
КОТОРОЕ (ОТЛИЧИЕ ОТ DYNAMIC)
НА СЛЕДУЮЩИХ ШАГАХ УМЕНЬШАЕТСЯ ДО <NUM_ITERS>

N0 ПРОПОРЦИОНАЛЬНО:
<КОЛИЧЕСТВО ИТЕРАЦИЙ ЦИКЛА> /<ЧИСЛО ТРЕДОВ>

Ni ПРОПОРЦИОНАЛЬНО
<КОЛИЧЕСТВО ОСТАВШИХСЯ ИТЕРАЦИЙ ЦИКЛА> /<ЧИСЛО ТРЕДОВ>
…

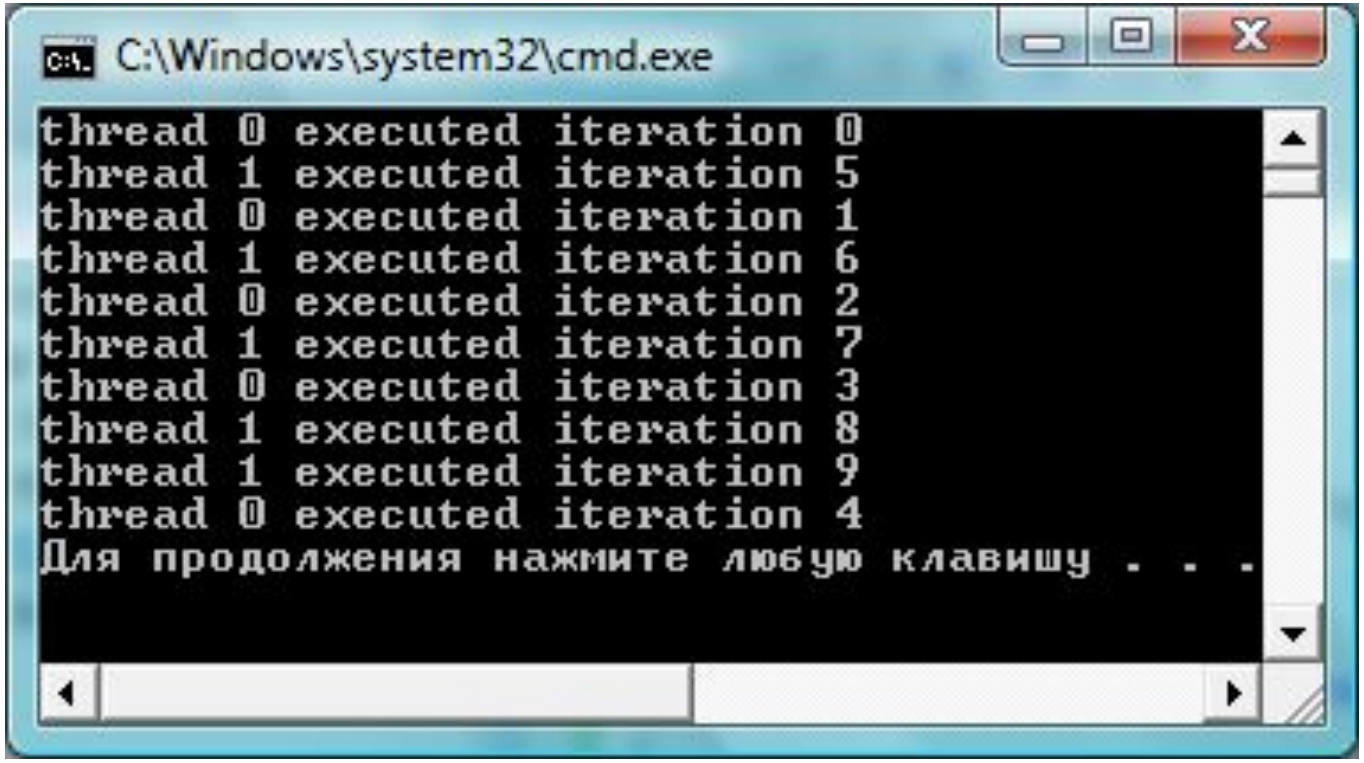
RUNTIME – ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ЗАГРУЗКИ ТРЕДОВ

СПОСОБ РАСПРЕДЕЛЕНИЯ ИТЕРАЦИЙ ВЫБИРАЕТСЯ ВО ВРЕМЯ РАБОТЫ ПРОГРАММЫ ПО ЗНАЧЕНИЮ ПЕРЕМЕННОЙ СРЕДЫ OMP_SCHEDULE.

ПРИМЕР 1


```
#include <stdio.h>
#include <windows.h>
#include <omp.h>
int main(int argc, char *argv[])
{int i;
#pragma omp parallel private(i)
{//#pragma omp for schedule (static)
//#pragma omp for schedule (static, 1)
//#pragma omp for schedule (static, 2)
//#pragma omp for schedule (dynamic)
//#pragma omp for schedule (dynamic, 2)
//#pragma omp for schedule (guided)
  #pragma omp for schedule (guided, 2)
    for (i=0; i<10; i++)
    {  printf("thread %d executed iteration %d\n",
  omp_get_thread_num(), i);
       Sleep(1);
    }
}}
```

РЕЗУЛЬТАТ


C:\Windows\system32\cmd.exe
thread 0 executed iteration 0
thread 1 executed iteration 5
thread 0 executed iteration 1
thread 1 executed iteration 6
thread 0 executed iteration 2
thread 1 executed iteration 7
thread 0 executed iteration 3
thread 1 executed iteration 8
thread 1 executed iteration 9
thread 0 executed iteration 4
Для продолжения нажмите любую клавишу . . .

ЗАДАНИЕ

- Варьируя число итераций, тредов и размер начального блока, проанализировать распределение итераций по тредам.
- Изменяется ли распределение итераций по тредам при нескольких запусках одной и той же программы?

Примечание. `void Sleep(int k)` – задержка в миллисекундах (здесь для имитации вычислений).

!Если задать значение параметра (0), то работа потока может быть **приостановлена** для того, чтобы позволить другим ожидающим потокам выполняться (**в примере 2**).

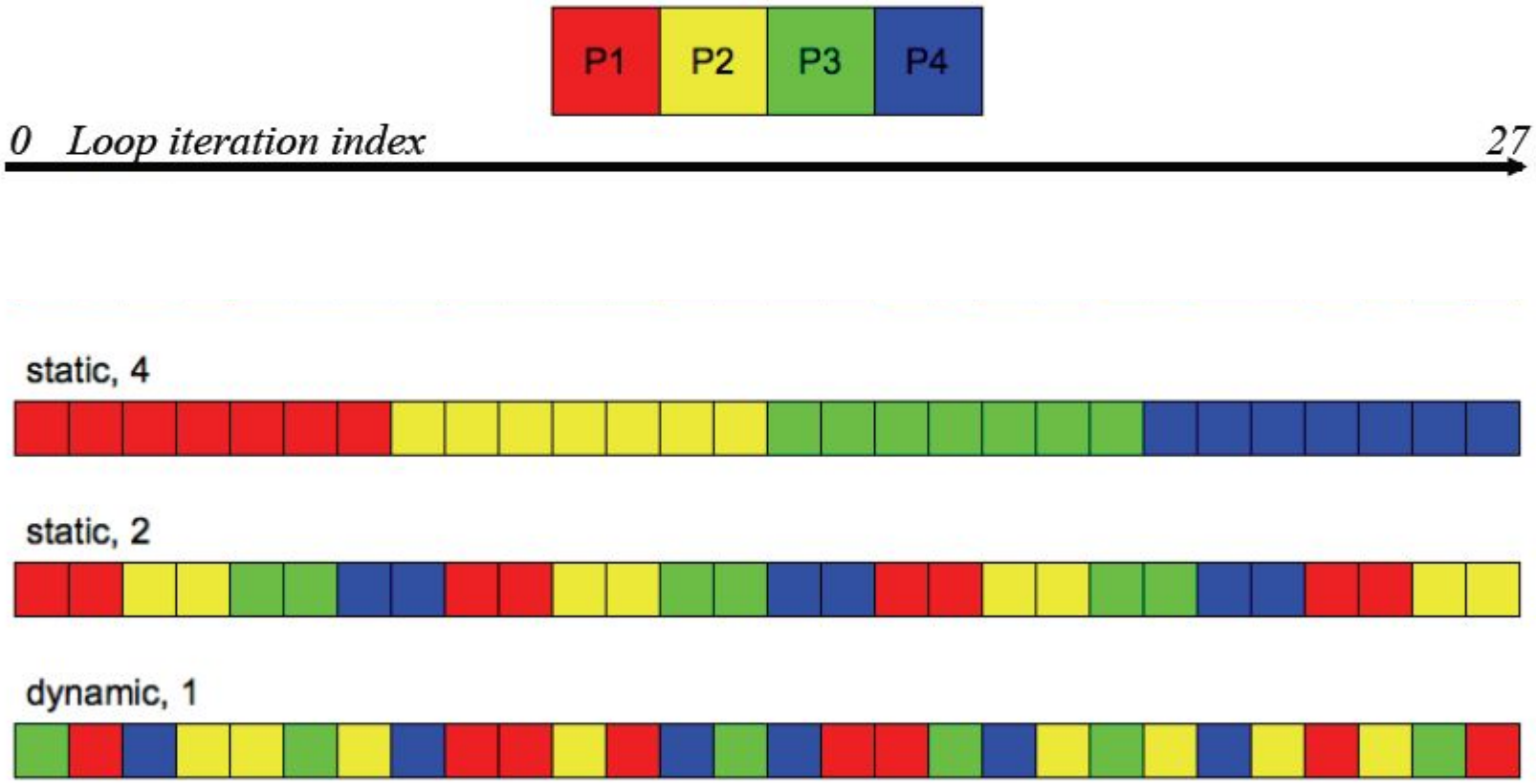
ПРИМЕР 2 msdn.microsoft.com/ru-ru/library/x5aw0hdf(v=vs.90).aspx

```
#define NUM_THREADS 4
#define STATIC_CHUNK 5
#define DYNAMIC_CHUNK 5
#define NUM_LOOPS 20
#define SLEEP_EVERY_N 3

int main( )
{
    int nStatic1[NUM_LOOPS],
        nStaticN[NUM_LOOPS];
    int nDynamic1[NUM_LOOPS],
        nDynamicN[NUM_LOOPS];
    int nGuided[NUM_LOOPS];

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        #pragma omp for schedule(static, 1)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
                Sleep(0);
            nStatic1[i] = omp_get_thread_num( );
        }

        #pragma omp for schedule(static, STATIC_CHUNK)
        for (int i = 0 ; i < NUM_LOOPS ; ++i)
        {
            if ((i % SLEEP_EVERY_N) == SLEEP_EVERY_N)
```

ЗАДАНИЕ

- Протестировать программу, изменяя значения основных параметров.
- Отобразить результат графически.
- Для каких режимов существенно количество тредов?
- Как изменится работа программы, если установить явно время задержки?
- Добавить свои комментарии в текст программы.

ПРИМЕР ИЛЛЮСТРАЦИИ



ОПЦИЯ `COLLAPSE`

`collapse(n)` — n последовательных тесновложенных циклов ассоциируется с данной директивой.

- Для циклов образуется **общее** пространство итераций, которое делится между тредами.
- Если опция **не задана, то директива относится только к одному** - непосредственно следующему за ней циклу.

ОПЦИЯ `ordered`

Опция для указания о том, что в цикле могут встречаться **директивы** `ordered`.

В этом случае определяется **блок внутри тела цикла**, который должен выполняться в порядке, установленном в **последовательном цикле**

ОПЦИЯ **NOWAIT**

- По умолчанию в конце параллельного цикла происходит **неявная барьерная синхронизация** параллельно работающих тредов –

  дальнейшее выполнение происходит только тогда, когда все треды достигнут данной точки (***барьера***).

- Если подобная задержка не нужна, используют опцию **nowait.**

- Это позволяет тредам,
  уже дошедшим до конца цикла, **продолжить выполнение без синхронизации** с остальными тредами.

ПРИМЕР

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main ()
{
  int i, chunk;
  float a[N], b[N], c[N];
// Some initializations
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
         for (i=0; i < N; i++)
              c[i] = a[i] + b[i];
  }
 // end of parallel section
 }
```

ЗАКЛЮЧЕНИЕ ПО РАСПАРАЛЛЕЛИВАНИЮ ЦИКЛОВ

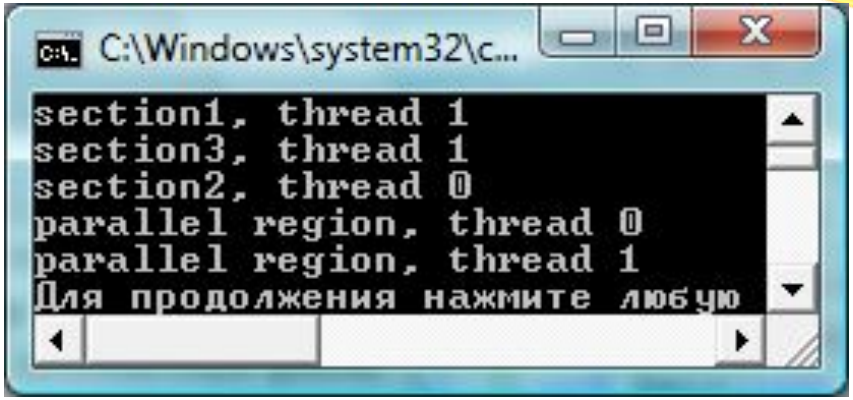
- При распараллеливании цикла надо убедиться в том, что **итерации данного цикла не имеют информационных зависимостей**.
- Если цикл не содержит зависимостей, его итерации можно выполнять в **любом** порядке, в том числе **параллельно**.
- Соблюдение этого требования **компилятор не проверяет**, вся ответственность - на программисте.
- Если дать указание компилятору распараллелить цикл, содержащий зависимости, результат работы программы может оказаться **некорректным**.
- **Задание** – подобрать пример такого цикла, проверить выполнение параллельной программы.

ДИРЕКТИВА SECTIONS

- Используется для реализации **функционального параллелизма**.
- Эта директива определяет набор **независимых** секций кода, каждая из которых выполняется **своим тредом**.

СИНТАКСИС ДИРЕКТИВЫ **SECTIONS**

```
#pragma omp sections [опции ...] newline
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait
{
  #pragma omp section newline
    structured_block //отдельный тред
  #pragma omp section newline
    structured_block //отдельный тред
…
}
```




```
int main()
{  int n;
   #pragma omp parallel private(n)
   {  n=omp_get_thread_num();
      #pragma omp sections
      {
           #pragma omp section
           {    printf("section1, thread %d\n", n);
           }
           #pragma omp section
           {    printf("section2, thread %d\n", n);
           }
           #pragma omp section
           {    printf("section3, thread %d\n", n);
           }
      }
      printf("parallel region, thread %d\n", n);
   }
}
```