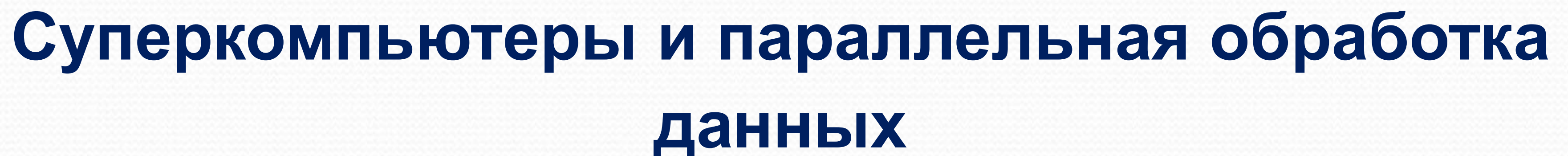


Суперкомпьютеры и параллельная обработка данных

Бахтин Владимир Александрович
к.ф.-м.н., ведущий научный сотрудник
Института прикладной математики им М.В.Келдыша РАН
кафедра системного программирования
факультет вычислительной математики и кибернетики
Московского университета им. М.В. Ломоносова

Клауза copyin

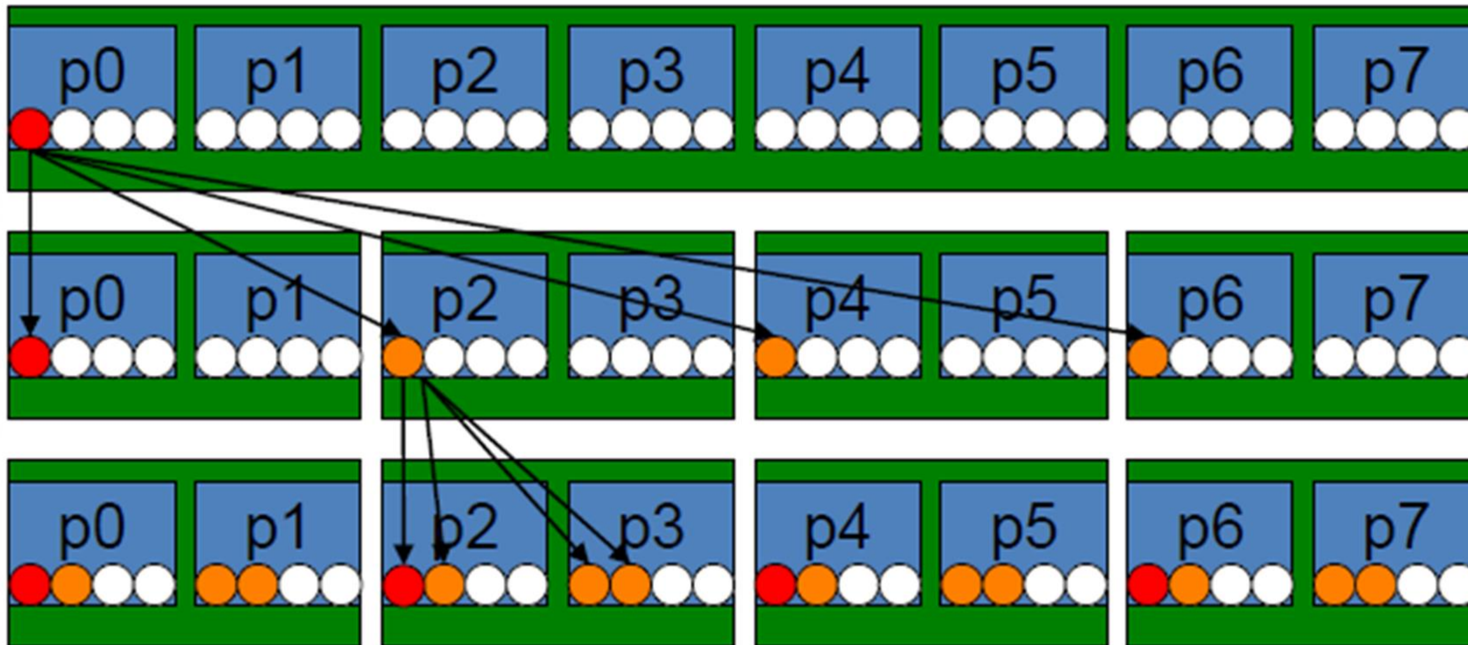
copyin(list)

Значение каждой threadprivate-переменной из списка list, устанавливается равным значению этой переменной в master-нити

```
float* work;
int size;
float val;
#pragma omp threadprivate(work,size,val)
void compute()
{
   work = (float*)malloc( sizeof(float)*size);
   for(int i = 0; i < size; ++i ) work[i] = val;
   … // computation with work array elements
}
int main()
{
    printf("Enter the size of array and value\n");
    scanf("%d",&size);
    scanf("%f",&val);
    #pragma omp parallel copyin(val,size)
        compute();
}
```

Клауза proc_bind (OpenMP 4.0)

```
#pragma omp parallel proc_bind(spread) num_threads(4)
 {
   #pragma omp parallel proc_bind(close) num_threads(4)
        work();
 }
```



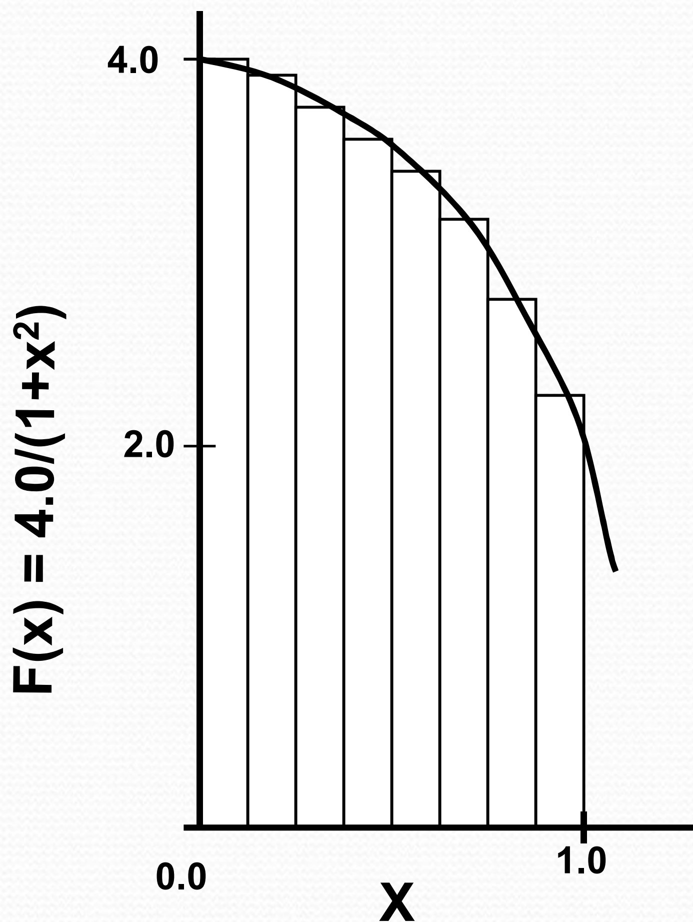
Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- Новые возможности OpenMP

Конструкции распределения работы

- Распределение витков циклов (директива for)
- Циклы с зависимостью по данным. Организация конвейерного выполнения для циклов с зависимостью по данным.
- Распределение нескольких структурных блоков между нитями (директива SECTION).
- Выполнение структурного блока одной нитью (директива single)
- Распределение операторов одного структурного блока между нитями (директива WORKSHARE)
- Понятие задачи

Вычисление числа π



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

Мы можем аппроксимировать интеграл как сумму прямоугольников:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Где каждый прямоугольник имеет ширину Δx и высоту $F(x_i)$ в середине интервала

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
   {
        int id = omp_get_thread_num();
        int numt = omp_get_num_threads();
        for (i = id + 1; i <= n;  i=i+numt)
        {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
        }
     }
     pi = h * sum;
     printf("pi is approximately %.16f", pi);
     return 0;
}
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
   {
     #pragma omp for schedule (static,1)
     for (i = 1; i <= n;  i++)
       {
             x = h * ((double)i - 0.5);
             sum += (4.0 / (1.0 + x*x));
       }
     }
     pi = h * sum;
     printf("pi is approximately %.16f", pi);
     return 0;
}
```

Вычисление числа π на OpenMP

```
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        int iam = omp_get_thread_num();
        int numt = omp_get_num_threads();
        int start = iam * n / numt + 1;
        int end = (iam + 1) * n / numt;
        if (iam == numt-1) end = n;
        for (i = start; i <= end;  i++)
        {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
         }
     }
     pi = h * sum;
     printf(“pi is approximately %.16f”, pi);
     return 0;
}
```

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
int main ()
{
    int n =100, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h) reduction(+:sum)
    {
        #pragma omp for schedule (static)
        for (i = 1; i <= n;  i++)
         {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
         }
     }
     pi = h * sum;
     printf(“pi is approximately %.16f”, pi);
     return 0;
}
```

Распределение витков цикла

#pragma omp for *[клауза[[,]клауза] ...]*
for (*init-expr***;** *test-expr***;** *incr-expr***)** *структурный блок*

где клауза одна из :

- **private(***list***)**
- **firstprivate(***list***)**
- **lastprivate(***list***)**
- **reduction(***operator***:** *list***)**
- **schedule(***kind[***,** *chunk_size]***)**
- **collapse(***n***)**
- **ordered(***n***)**
- **nowait**

Распределение витков цикла

init-expr : var = loop-invariant-expr1
| integer-type var = loop-invariant-expr1
| random-access-iterator-type var = loop-invariant-expr1
| pointer-type var = loop-invariant-expr1

test-expr:
var relational-op loop-invariant-expr2
| loop-invariant-expr2 relational-op var

incr-expr: ++var
| var++
| --var
| var --
| var += loop-invariant-integer- expr
| var -= loop-invariant-integer- expr
| var = var + loop-invariant-integer- expr
| var = loop-invariant-integer- expr + var
| var = var - loop-invariant-integer- expr

relational-op: <
| <=
| >
| >=

var: signed or unsigned integer type
| random access iterator type
| pointer type

Parallel Random Access Iterator Loop (OpenMP 3.0)

```
#include <vector>
void iterator_example()
{
   std::vector<int> vec(23);
   std::vector<int>::iterator it;
#pragma omp parallel for default(none) shared(vec)
   for (it = vec.begin(); it < vec.end(); it++)
   {
     // do work with *it //
   }
}
```

Использование указателей в цикле (OpenMP 3.0)

```
void pointer_example ()
{
   char a[N];
   #pragma omp for default (none) shared (a,N)
   for (char *p = a; p < (a+N); p++ )
   {
       use_char (p);
   }
}
```

for (char *p = a; p != (a+N); p++) - **ошибка**

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void good_collapsing(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for collapse (2)
        for (i=0; i<n; i++) {
                for (j=0; j < n; j++)
                    work(i, j);
        }
     }
}
```

Клауза collapse:

collapse (*положительная целая константа*)

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void error_collapsing(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for collapse (2)
        for (i=0; i<n; i++) {
            work_with_i (i);            //  Ошибка
            for (j=0; j < n; j++)
                work(i, j);
        }
    }
}
```

Клауза collapse может быть использована только для распределения витков тесно-вложенных циклов.

Распределение витков многомерных циклов. Клауза collapse (OpenMP 3.0)

```
void work(int i, int j) {}
void error_collapsing(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for collapse (2)
        for (i=0; i<n; i++) {
            for (j=0; j < i; j++)            //  Ошибка
                work(i, j);
        }
    }
}
```

Клауза collapse может быть использована только для распределения витков циклов с прямоугольным индексным пространством.

Использование редукционных операций

```
void reduction (float *x, int *y, int n)
{
  int i, b, c;
  float a, d;
  a = 0.0;
  b = 0;
  c = y[0];
  d = x[0];
  #pragma omp parallel for private(i) shared(x, y, n) \
                        reduction(+:a) reduction(^:b) \
                        reduction(min:c) reduction(max:d)
    for (i=0; i<n; i++) {
      a += x[i];
      b ^= y[i];
      if (c > y[i]) c = y[i];
      d = fmaxf(d,x[i]);
    }
}
```

Реализация редукционных операций

```
#include <limits.h>
void reduction_by_hand (float *x, int *y, int n)
{
  int i, b, b_p, c, c_p;
  float a, a_p, d, d_p;
  a = 0.0f;
  b = 0;
  c = y[0];
  d = x[0];
  #pragma omp parallel shared(a, b, c, d, x, y, n)  private(a_p, b_p, c_p, d_p)
  {
    a_p = 0.0f;   b_p = 0;   c_p = INT_MAX;    d_p = -HUGE_VALF;
    #pragma omp for private(i) nowait
    for (i=0; i<n; i++) {
      a_p += x[i];   b_p ^= y[i];   if (c_p > y[i]) c_p = y[i];   d_p = fmaxf(d_p,x[i]);
    }
    #pragma omp critical
    {
      a += a_p;   b ^= b_p;   if( c > c_p ) c = c_p;   d = fmaxf(d,d_p);
    }
  }
}
```

Редукционные операции, определяемые пользователем (OpenMP 4.0)

```
#pragma omp declare reduction (reduction-identifier : typename-list :
combiner) [initializer(initializer-expr)]
```

- reduction-identifier - название редукционной операции
- typename-list – тип (типы)
- combiner – выражение для объединения частичных результатов
- initializer – начальное значение создаваемых приватных переменных
- omp_out refers to private copy that holds combined value
- omp_in refers to the other private copy
- omp_priv represents the private element
- omp_orig represents the original variable

Использование редукционных операций, определяемых пользователем (OpenMP 4.0)

```
struct point
{
   int x;
   int y;
} points[N], minp = { MAX_INT, MAX_INT };

#pragma omp declare reduction (min : struct point : \
    omp_out.x = omp_in.x > omp_out.x ? omp_out.x : omp_in.x, \
    omp_out.y = omp_in.y > omp_out.y ? omp_out.y : omp_in.y ) \
    initializer ( omp_priv = { MAX_INT, MAX_INT })

#pragma omp parallel for reduction (min: minp)
for (int i = 0; i < N; i++)
{
  if (points[i].x < minp.x) minp.x = points[i].x;
  if (points[i].y < minp.y) minp.y = points[i].y;
}
```

Редукционные операции, определяемые пользователем (OpenMP 4.0)

```
#pragma omp declare reduction (merge : std::vector<int> :
omp_out.insert (omp_out.end(), omp_in.begin(), omp_in.end()))

void schedule (std::vector<int> &v, std::vector<int> &filtered)
{
    #pragma omp parallel for reduction (merge: filtered)
    for (std:vector<int>::iterator it = v.begin(); it < v.end(); it++)
        if ( filter(*it) ) filtered.push_back(*it);
}
```

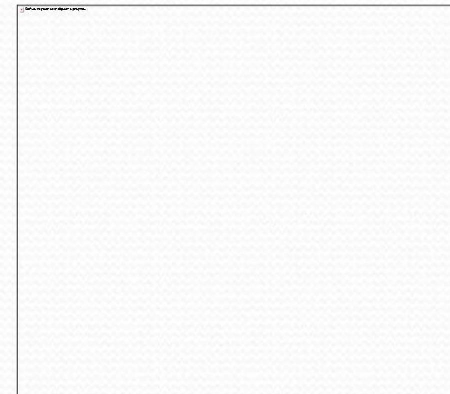
Распределение витков цикла. Клауза schedule

Клауза schedule:
schedule(*алгоритм планирования[, число_итераций]*)

Где алгоритм планирования один из:

- **schedule(static*[, число_итераций]*) - статическое планирование;**
- **schedule(dynamic*[, число_итераций]*) - динамическое планирование;**
- **schedule(guided*[, число_итераций]*) - управляемое планирование;**
- **schedule(runtime) - планирование в период выполнения;**
- **schedule(auto) - автоматическое планирование (OpenMP 3.0).**

```
#pragma omp parallel for private(tmp) shared (a) schedule (runtime)
    for (int i=0; i<N-2; i++)
     for (int j = i+1; j< N-1; j++) {
      tmp = a[i][j];
      a[i][j]=a[j][i];
      a[j][i]=tmp;
     }
```



Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(static)
    for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре будет следующим:

- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-50.
- Поток 2 получает право на выполнение итераций 51-75.
- Поток 3 получает право на выполнение итераций 76-100.

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(static, 10)
    for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре будет следующим:

- Поток 0 получает право на выполнение итераций 1-10, 41-50, 81-90.
- Поток 1 получает право на выполнение итераций 11-20, 51-60, 91-100.
- Поток 2 получает право на выполнение итераций 21-30, 61-70.
- Поток 3 получает право на выполнение итераций 31-40, 71-80

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(dynamic, 15)
  for(int i = 1; i <= 100; i++)
```

Результат выполнения программы на 4-х ядерном процессоре может быть следующим:

- Поток 0 получает право на выполнение итераций 1-15.
- Поток 1 получает право на выполнение итераций 16-30.
- Поток 2 получает право на выполнение итераций 31-45.
- Поток 3 получает право на выполнение итераций 46-60.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 61-75.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 76-90.
- Поток 0 завершает выполнение итераций.
- Поток 0 получает право на выполнение итераций 91-100.

Распределение витков цикла. Клауза schedule

число_выполняемых_потоком_итераций = max(число_нераспределенных_итераций/omp_get_num_threads(), число_итераций)

```
#pragma omp parallel for schedule(guided, 10)
   for(int i = 1; i <= 100; i++)
```

Пусть программа запущена на 4-х ядерном процессоре.

- Поток 0 получает право на выполнение итераций 1-25.
- Поток 1 получает право на выполнение итераций 26-44.
- Поток 2 получает право на выполнение итераций 45-59.
- Поток 3 получает право на выполнение итераций 60-69.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 70-79.
- Поток 2 завершает выполнение итераций.
- Поток 2 получает право на выполнение итераций 80-89.
- Поток 3 завершает выполнение итераций.
- Поток 3 получает право на выполнение итераций 90-99.
- Поток 1 завершает выполнение итераций.
- Поток 1 получает право на выполнение 100 итерации.

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(runtime)
  for(int i = 1; i <= 100; i++)  /* способ распределения витков цикла между
   нитями будет задан во время выполнения программы*/
```

При помощи переменных среды:

csh:

```
setenv OMP_SCHEDULE "dynamic,4“
```

ksh:

```
export OMP_SCHEDULE="static,10“
```

Windows:

```
set OMP_SCHEDULE=auto
```

или при помощи функции системы поддержки:

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

Распределение витков цикла. Клауза schedule

```
#pragma omp parallel for schedule(auto)
  for(int i = 1; i <= 100; i++)
```

Способ распределения витков цикла между нитями определяется реализацией компилятора.

На этапе компиляции программы или во время ее выполнения определяется оптимальный способ распределения.

Распределение витков цикла. Клауза nowait

```
void example(int n, float *a, float *b, float *c, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++) {
            c[i] = (a[i] + b[i]) / 2.0;
            …
        }
        #pragma omp for schedule(static) nowait
          for (i=0; i<n; i++)
             z[i] = sqrt(c[i]);
    }
}
```

Верно в OpenMP 3.0, если количество итераций у циклов совпадает и параметры клаузы schedule совпадают (STATIC + *число_итераций*).

Распределение циклов с зависимостью по данным

```
for(int i = 1; i < 100; i++)
    a[i]= a[i-1] + a[i+1];
```

Между витками цикла с индексами i1 и i2 (i1<i2) существует зависимость по данным (информационная связь) массива *a*, если оба эти витка осуществляют обращение к одному элементу массива по схеме запись-чтение или чтение-запись.

Если виток i1 записывает значение, а виток i2 читает это значение, то между этими витками существует потоковая зависимость или просто зависимость i1 -> i2.

Если виток i1 читает "старое" значение, а виток i2 записывает "новое" значение, то между этими витками существует обратная зависимость i1<- i2.

В обоих случаях виток i2 может выполняться только после витка i1.

Распределение циклов с зависимостью по данным

```
for (int i = 0; i < n; i++) {
   x = (b[i] + c[i]) / 2;
   a[i] = a[i + 1] + x;       // ANTI dependency
}


#pragma omp parallel shared(a, a_copy) private (x)
{
   #pragma omp for
   for (int i = 0; i < n; i++) {
       a_copy[i] = a[i + 1];
   }
   #pragma omp for
   for (int i = 0; i < n; i++) {
       x = (b[i] + c[i]) / 2;
       a[i] = a_copy[i] + x;
   }
}
```

Распределение циклов с зависимостью по данным

```
for (int i = 1; i < n; i++) {
   b[i] = b[i] + a[i - 1];
   a[i] = a[i] + c[i]; // FLOW dependency
}
```

```
b[1] = b[1] + a[0];
a[1] = a[1] + c[1];
b[2] = b[2] + a[1];
a[2] = a[2] + c[2];
b[3] = b[3] + a[2];
a[3] = a[3] + c[3];
```

```
b[1] = b[1] + a[0];
#pragma omp parallel for shared(a,b,c)
for (int i = 1; i < n - 1; i++) {
    a[i] = a[i] + c[i];
    b[i + 1] = b[i + 1] + a[i];
}
a[n - 1] = a[n - 1] + c[n - 1];
```

Клауза и директива ordered

```
void print_iteration(int iter) {
   #pragma omp ordered
      printf("iteration %d\n", iter);
}

int main( ) {
   int i;
   #pragma omp parallel
   {
      #pragma omp for ordered
         for (i = 0 ; i < 5 ; i++) {
            print_iteration(i);
            another_work (i);
         }
   }
}
```

Результат выполнения программы:

```
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
```

Распределение циклов с зависимостью по данным. Клауза и директива ordered

```
#pragma omp parallel for ordered
for(int i = 1; i < 100; i++) {
    #pragma omp ordered
      {
        a[i]= a[i-1] + a[i+1];
      }
}
```

Распределение циклов с зависимостью по данным. Метод переменных направлений (ADI)

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i+1][j]) / 2;
```

```
for(int i=1; i < M; i++)
  {
      #pragma omp parallel for shared(a)
      for(int j = 1; j < N; j++)
          a[i][j] = (a[i-1][j]+a[i+1][j])/2.;
  }
```

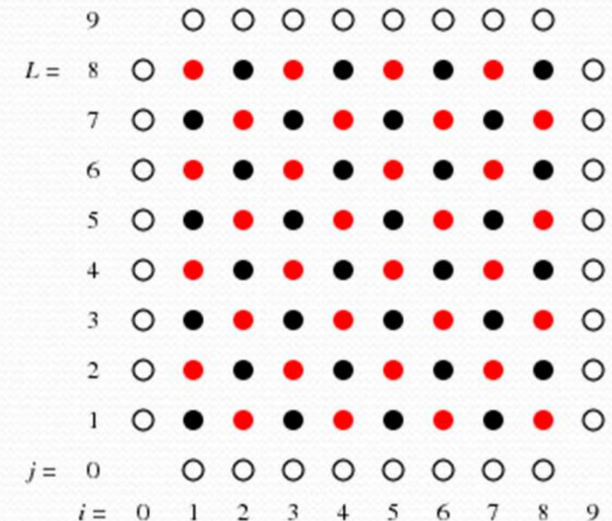
Распределение циклов с зависимостью по данным. Метод переменных направлений (ADI)

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i+1][j]) / 2;
```

```
#pragma omp parallel shared(a)
{
   for(int i=1; i < M; i++)
   {
       #pragma omp for
       for(int j = 1; j < N; j++)
           a[i][j] = (a[i-1][j]+a[i+1][j])/2.;
   }
}
```

Распределение циклов с зависимостью по данным. Метод Red-Black

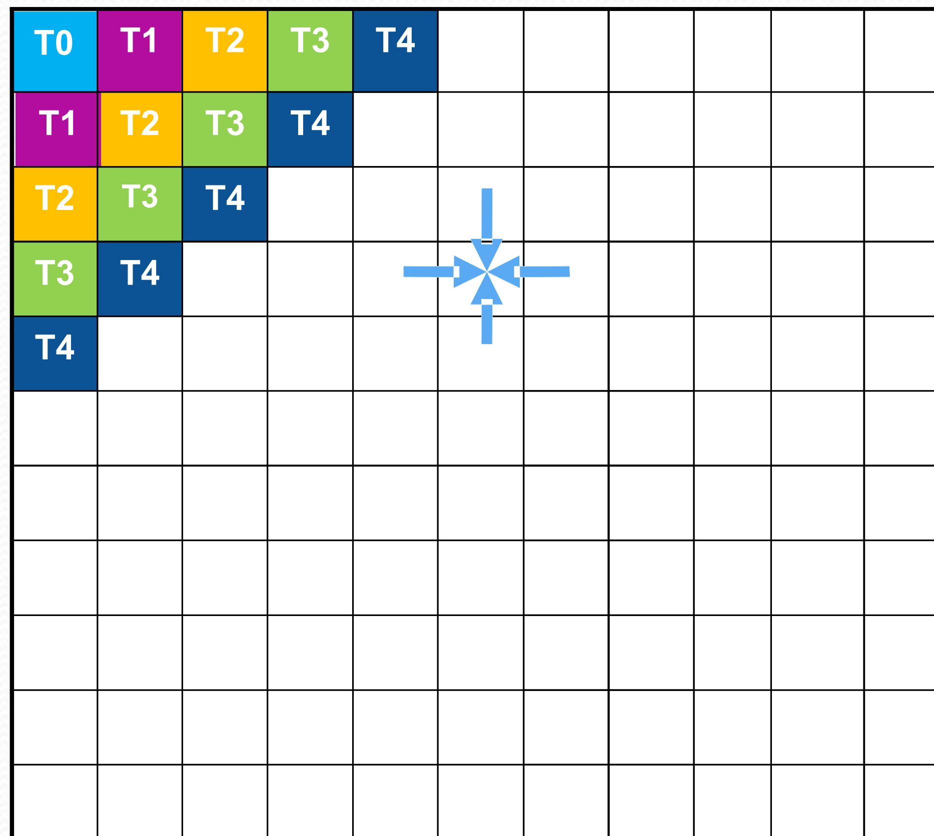
```
for(int i = 1; i < M; i++)
   for(int j = 1; j < N; j++)
        a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```




```
#pragma omp parallel shared(a)
{
     #pragma omp for
    for(int i = 1; i < M; i++)
      for(int j = 1 + i %2 ; j < N; j += 2)
        a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
     #pragma omp for
    for(int i = 1; i < M; i++)
      for(int j = 1 + (i + 1)%2 ; j < N; j += 2)
        a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
}
```

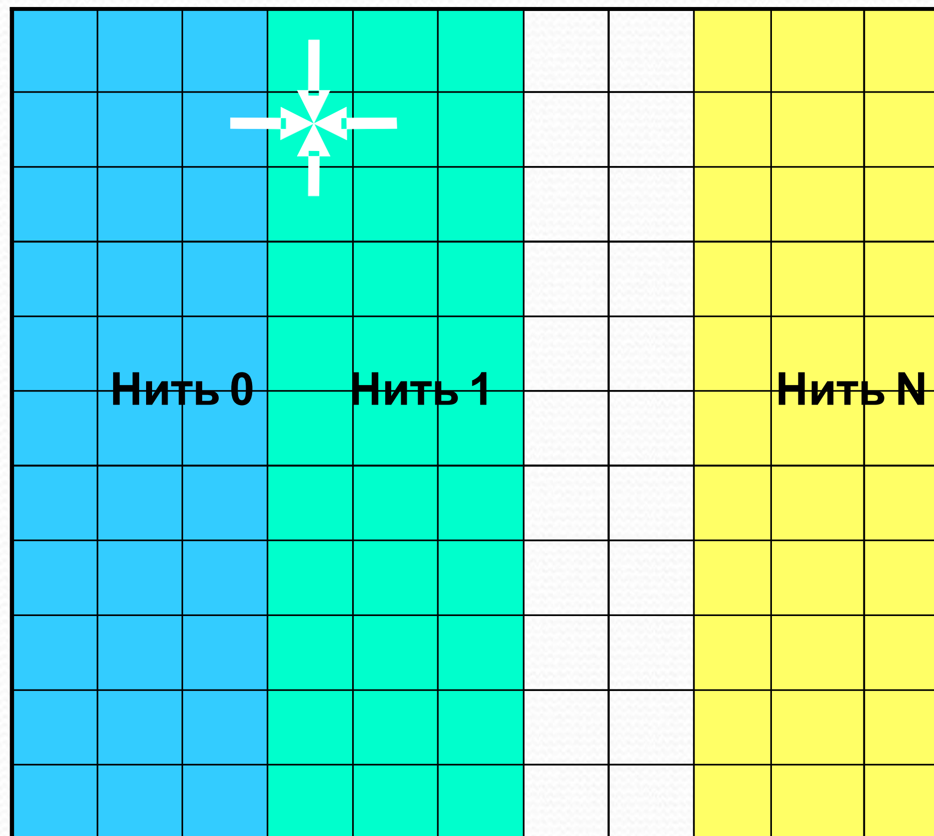
Распределение циклов с зависимостью по данным. Параллелизм по гиперплоскостям

```
for(int i = 1; i < M; i++)
   for(int j = 1; j < N; j++)
      a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```



Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```



Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
for(int i = 1; i < M; i++)
  for(int j = 1; j < N; j++)
    a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```

T0 T1 T2

T1 T2

T2

Нить 0 Нить 1 Нить 2

Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

isync	1	0	0	0

T0	T1	T2		
T1	T2			
T2				
Нить 0	Нить 1	Нить 2		

Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
int iam, numt, limit;
int *isync = (int *)
malloc(omp_get_max_threads()*sizeof(int));
#pragma omp parallel private(iam,numt,limit)
  {
  iam = omp_get_thread_num ();
  numt = omp_get_num_threads ();
  limit=min(numt-1,N-2);
  isync[iam]=0;
#pragma omp barrier
  for (int i=1; i<M; i++) {
    if ((iam>0) && (iam<=limit)) {
       for (;isync[iam-1]==0;) {
          #pragma omp flush (isync)
       }
       isync[iam-1]=0;
       #pragma omp flush (isync)
    }
#pragma omp for schedule(static) nowait
    for (int j=1; j<N; j++) {
      a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] +
                 a[i][j+1])/4;
    }
    if (iam<limit) {
       for (;isync[iam]==1;) {
          #pragma omp flush (isync)
       }
       isync[iam]=1;
       #pragma omp flush (isync)
     }
   }
}
```

Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
#pragma omp parallel
  {
    int iam = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    for (int newi=1; newi<M + numt - 1; newi++) {
      int i = newi - iam;
      #pragma omp for
        for (int j=1; j<N; j++) {
          if ((i >= 1) && (i< M)) {
            a[i][j]=(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1])/4;
          }
        }
    }
  }
```

Распределение циклов с зависимостью по данным. OpenMP 4.5

```
#pragma omp parallel for ordered(2) shared(a)
 for (int i=1; i<M; i++)
        for (int j=1; j<N; j++) {
              #pragma omp ordered depend (sink: i - 1, j) depend (sink: i, j - 1)
                    a[i][j] = (a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])/4;
              #pragma omp ordered depend (source)
        }
```

Распределение нескольких структурных блоков между нитями (директива sections)

```
#pragma omp sections [клауза[[,] клауза] ...]
{
   [#pragma omp section]
   структурный блок
   [#pragma omp section
   структурный блок ]
   ...
}
```

где клауза одна из :

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
nowait
```

```
void XAXIS();
void YAXIS();
void ZAXIS();
void example()
{
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            XAXIS();
            #pragma omp section
            YAXIS();
            #pragma omp section
            ZAXIS();
        }
    }
}
```

Выполнение структурного блока одной нитью (директива single)

#pragma omp single *[клауза[[,] клауза] ...]*
структурный блок

где клауза одна из :

- **private(***list***)**
- **firstprivate(***list***)**
- **copyprivate(***list***)**
- **nowait**

Структурный блок будет выполнен одной из нитей. Все остальные нити будут дожидаться результатов выполнения блока, если не указана клауза NOWAIT.

```
#include <stdio.h>
static float x, y;
#pragma omp threadprivate(x, y)
void init(float *a, float *b ) {
  #pragma omp single copyprivate(a,b,x,y)
     scanf("%f %f %f %f", a, b, &x, &y);
}
int main () {
   #pragma omp parallel
   {
      float x1,y1;
      init (&x1,&y1);
      parallel_work ();
   }
}
```

Распределение операторов одного структурного блока между нитями (директива WORKSHARE)

```
      SUBROUTINE EXAMPLE (AA, BB, CC, DD, EE, FF, GG, HH, N)
       INTEGER N
       REAL AA(N,N), BB(N,N), CC(N,N)
       REAL DD(N,N), EE(N,N), FF(N,N)
       REAL GG(N,N), HH(N,N)
       REAL SHR
!$OMP PARALLEL SHARED(SHR)
!$OMP WORKSHARE
       AA = BB
       CC = DD
       WHERE (EE .ne. 0) FF = 1 / EE
       SHR = 1.0
       GG (1:50,1) = HH(11:60,1)
       HH(1:10,1) = SHR
!$OMP END WORKSHARE
!$OMP END PARALLEL
       END SUBROUTINE EXAMPLE
```

Понятие задачи

Задачи появились в OpenMP 3.0

Каждая задача:

- Представляет собой последовательность операторов, которые необходимо выполнить.
- Включает в себя данные, которые используются при выполнении этих операторов.
- Выполняется некоторой нитью.

В OpenMP 3.0 каждый оператор программы является частью одной из задач.

- При входе в параллельную область создаются неявные задачи (implicit task), по одной задаче для каждой нити.
- Создается группа нитей.
- Каждая нить из группы выполняет одну из задач.
- По завершении выполнения параллельной области, master-нить ожидает, пока не будут завершены все неявные задачи.

Понятие задачи. Директива task

Явные задачи (explicit tasks) задаются при помощи директивы:

#pragma omp task [клауза[[,] клауза] ...]

структурный блок

где клауза одна из :

- **if (scalar-expression)**
- **final(scalar-expression)** //OpenMP 3.1
- **untied**
- **mergeable** //OpenMP 3.1
- **shared (list)**
- **private (list)**
- **firstprivate (list)**
- **default (shared | none)**
- **depend (dependence-type: list)** //OpenMP 4.0

В результате выполнения директивы task создается новая задача, которая состоит из операторов структурного блока; все используемые в операторах переменные могут быть локализованы внутри задачи при помощи соответствующих клауз. Созданная задача будет выполнена одной нитью из группы.

Понятие задачи. Директива task

```
#pragma omp for schedule(dynamic)
    for (i=0; i<n; i++) {
        func(i);
    }
```

```
#pragma omp single
{
    for (i=0; i<n; i++) {
        #pragma omp task firstprivate(i)
            func(i);
    }
}
```

Использование директивы task

```
typedef struct node node;
struct node {
    int data;
    node * next;
};
void increment_list_items(node * head)
{
   #pragma omp parallel
      {
        #pragma omp single
           {
              node * p = head;
              while (p) {
                 #pragma omp task
                      process(p);
                 p = p->next;
              }
           }
      }
}
```

Использование директивы task. Клауза if

```
double *item;
int main() {
  #pragma omp parallel shared (item)
    {
      #pragma omp single
        {
          int size;
          scanf("%d",&size);
          item = (double*)malloc(sizeof(double)*size);
          for (int i=0; i<size; i++)
             #pragma omp task if (size > 10)
                process(item[i]);
        }
    }
}
```

Если накладные расходы на организацию задач превосходят время, необходимое для выполнения блока операторов этой задачи, то блок операторов будет немедленно выполнен нитью, выполнившей директиву task

Использование директивы task

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
  #pragma omp parallel shared (item)
   {
     #pragma omp single
      {
         for (int i=0; i<LARGE_NUMBER; i++)
           #pragma omp task
              process(item[i]);
      }
   }
}
```

Как правило, в компиляторах существуют ограничения на количество создаваемых задач. Выполнение цикла, в котором создаются задачи, будет приостановлено. Нить, выполнявшая этот цикл, будет использована для выполнения одной из задач

Использование директивы task. Клауза untied

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
    #pragma omp parallel
      {
         #pragma omp single
           {
                 #pragma omp task untied
                {
                   for (int i=0; i<LARGE_NUMBER; i++)
                     #pragma omp task
                        process(item[i]);
                }
           }
      }
}
```

Клауза untied - выполнение задачи после приостановки может быть продолжено любой нитью группы

Использование задач. Директива taskwait

```
#pragma omp taskwait

int fibonacci(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
            i=fibonacci (n-1);
        #pragma omp task shared(j)
            j=fibonacci (n-2);
       #pragma omp taskwait
       return i+j;
    }
}
```

```
int main () {
    int res;
    #pragma omp parallel
    {
        #pragma omp single
        {
            int n;
            scanf("%d",&n);
            #pragma omp task shared(res)
                res = fibonacci(n);
         }
    }
    printf ("Finonacci number =  %d\n", res);
}
```

Использование директивы task. Клауза final

```
int fib (int n, int d) {
    int x, y;
    if (n < 2) return 1;
#pragma omp task final (d > LIMIT) mergeable
        x =  fib (n - 1, d + 1);
#pragma omp task final (d > LIMIT) mergeable
        y =  fib (n - 2, d + 1);
#pragma omp taskwait
        return x + y;
}

int omp_in_final (void);
```

Зависимости между задачами (OpenMP 4.0)

Клауза depend(dependence-type : list)
где dependence-type:

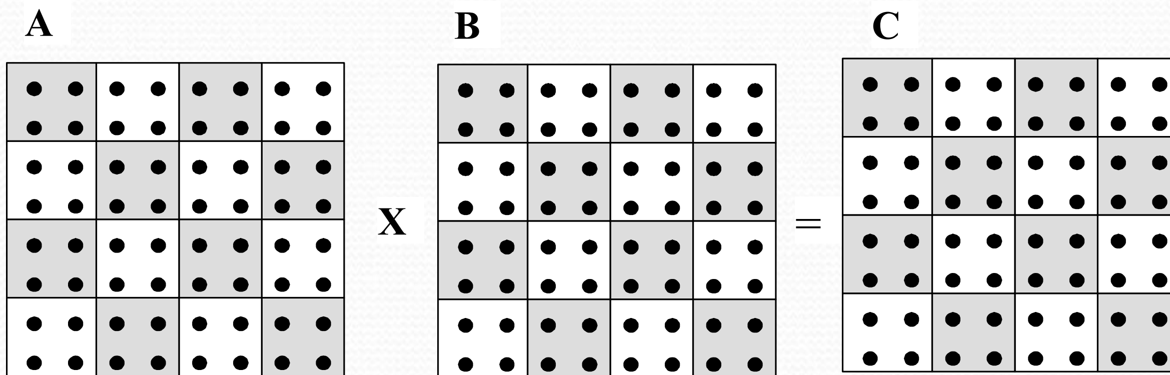
- **in**
- **out**
- **inout**

```
int i, y, a[100];

#pragma omp task depend(out : a)
{
    for (i=0;i<100; i++) a[i] = i + 1;
}

#pragma omp task depend(in : a[0:50]) depend(out : y)
{
    y = 0;
    for (i=0;i<50; i++) y += a[i];
}

#pragma omp task depend(in : y) {
    printf("%d\n", y);
}
```

Умножение матриц

```
void matmul (int N, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      for (k = 0; k < N; k++) {
         C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Умножение матриц



$$\begin{pmatrix} A_{00}A_{01}\ldots A_{0q-1} \\ \cdots \\ A_{q-10}A_{q-11}\ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01}\ldots B_{0q-1} \\ \cdots \\ B_{q-10}B_{q-11}\ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01}\ldots C_{0q-1} \\ \cdots \\ c_{q-10}C_{q-11}\ldots C_{q-1q-1} \end{pmatrix}, \quad C_{ij} = \sum_{s=1}^{q} A_{is}B_{sj}$$

Блочное умножение матриц

```
void matmul_block (int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
          for (ii = i; ii < i+BS; ii++ )
            for (jj = j; jj < j+BS; jj++ )
              for (kk = k; kk < k+BS; kk++ )
                C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

Зависимости между задачами (OpenMP 4.0)

```
void matmul_depend (int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
   int i, j, k, ii, jj, kk;
   for (i = 0; i < N; i+=BS) {
      for (j = 0; j < N; j+=BS) {
         for (k = 0; k < N; k+=BS) {
       #pragma omp task private(ii, jj, kk) firstprivate(i, j, k) \
            depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
            depend ( inout: C[i:BS][j:BS] )
              for (ii = i; ii < i+BS; ii++ )
               for (jj = j; jj < j+BS; jj++ )
                  for (kk = k; kk < k+BS; kk++ )
                     C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
         }
      }
   }
}
```

Приоритет задачи (OpenMP 4.5)

```
void compute_array (float *node, int M);

void compute_matrix (float *array, int N, int M)
{
 int i;
 #pragma omp parallel private(i)
 #pragma omp single
 {
   for (i=0;i<N; i++) {
     #pragma omp task priority(i)
         compute_array(&array[i*M], M);
   }
 }
}
```

Task Affinity (OpenMP 5.0)

```
double * alloc_init_B(double *A, int N);
void compute_on_B(double *B, int N);

void task_affinity(double *A, int N)
{
  double * B;
  #pragma omp task depend(out:B) shared(B) affinity(A[0:N])
  {
    B = alloc_init_B(A,N);
  }
  #pragma omp task depend( in:B) shared(B) affinity(A[0:N])
  {
    compute_on_B(B,N);
  }
  #pragma omp taskwait
}
```

Директива taskloop (OpenMP 4.5)

#pragma omp taskloop [clause[[,]clause]...]
structured-block

*где **clause** – одна из:*

if([taskloop :]scalar-expr)
shared(list)
private(list)
firstprivate(list)
lastprivate(list)
default(shared | none)
grainsize(grain-size)
num_tasks(num-tasks)
collapse(n)
final(scalar-expr)
priority(priority-value)
untied
mergeable
nogroup

Директива taskloop (OpenMP 4.5)

```
for (i = 0; i<SIZE; i++) {
        A[i]=A[i]*B[i]*S;
}

for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}

for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE : i+TS;
    #pragma omp task private(ii) \
           firstprivate(i,UB) shared(S,A,B)
    for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
    }
}
```

Директива taskloop (OpenMP 4.5)

```
for (i = 0; i<SIZE; i++) {
        A[i]=A[i]*B[i]*S;
}

for (i = 0; i<SIZE; i+=TS) {
   UB = SIZE < (i+TS) ? SIZE : i+TS;
   for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
   }
}

for (i = 0; i<SIZE; i+=TS) {
    UB = SIZE < (i+TS) ? SIZE : i+TS;
    #pragma omp task private(ii) \
          firstprivate(i,UB) shared(S,A,B)
    for (ii=i; ii<UB; ii++) {
      A[ii]=A[ii]*B[ii]*S;
    }
}
```

```
#pragma omp taskloop grainsize(TS)
for (i = 0; i<SIZE; i+=1) {
   A[i]=A[i]*B[i]*S;
}
```

Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- Новые возможности OpenMP

Конструкции для синхронизации нитей

- Директива master
- Директива critical
- Директива atomic
- Семафоры
- Директива barrier
- Директива taskyield
- Директива taskwait
- Директива taskgroup   // OpenMP 4.0

Директива master

#pragma omp master

структурный блок

/*Структурный блок будет выполнен MASTER-нитью группы. По завершении выполнения структурного блока барьерная синхронизация нитей не выполняется*/

```
#include <stdio.h>
void init(float *a, float *b ) {
 #pragma omp master
     scanf("%f %f", a, b);
 #pragma omp barrier
}
int main () {
  float x,y;
  #pragma omp parallel
  {
     init (&x,&y);
     parallel_work (x,y);
  }
}
```

Вычисление числа π на OpenMP с использованием критической секции

#pragma omp critical [(name)]
структурный блок

```
#include <omp.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
      double local_sum = 0.0;
 #pragma omp for nowait
       for (i = 1; i <= n;  i++) {
               x = h * ((double)i - 0.5);
               local_sum += (4.0 / (1.0 + x*x));
        }
 #pragma omp critical
               sum += local_sum;
    }
    pi = h * sum;
    printf("pi is approximately %.16f”, pi);
    return 0;
}
```

Использование критической секции

```
int *next_from_queue(int type);
void work(int *val);

void critical_example()
{
  #pragma omp parallel
  {
    int *ix_next, *iy_next;
    #pragma omp critical (xaxis)
         ix_next = next_from_queue(0);
    work(ix_next);

    #pragma omp critical (yaxis)
        iy_next = next_from_queue(1);
    work(iy_next);
   }
}
```

#pragma omp critical [(name)]
структурный блок

Директива atomic

```
#pragma omp atomic [ read | write | update | capture ] [seq_cst]
    expression-stmt

#pragma omp atomic capture
    structured-block
```

Если указана клауза read:

```
    v = x;
```

Если указана клауза write:

```
    x = expr;
```

Если указана клауза update или клаузы нет, то expression-stmt:

```
x binop= expr;
x = x binop expr;
x++;
++x;
x--;
--x;
```

x – скалярная переменная, expr – выражение, в котором не присутствует переменная x.

binop - не перегруженный оператор:

+ , * , - , / , & , ^ , | , << , >>

binop=:

++ , --

Директива atomic

Если указана клауза capture, то expression-stmt:

```
v = x++;
v = x--;
v = ++x;
v = -- x;
v = x binop= expr;
```

Если указана клауза capture, то structured-block:

```
{ v = x; x binop= expr;}
{ v = x; x =  x binop expr;}
{ v = x; x++;}
{ v = x; ++x;}
{ v = x; x--;}
{ v = x; --x;}
{ x binop= expr; v = x;}
{ x =  x binop expr;  v = x;}
{ v = x; x binop= expr;}
{ x++; v = x;}
{ ++ x ; v = x;}
{ x--; v = x;}
{ --x; v = x;}
```

Встроенные функции для атомарного доступа к памяти в GCC

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)
type __sync_add_and_fetch (type *ptr, type value, ...)
type __sync_sub_and_fetch (type *ptr, type value, ...)
type __sync_or_and_fetch (type *ptr, type value, ...)
type __sync_and_and_fetch (type *ptr, type value, ...)
type __sync_xor_and_fetch (type *ptr, type value, ...)
type __sync_nand_and_fetch (type *ptr, type value, ...)
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
```

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html

Вычисление числа π на OpenMP с использованием директивы atomic

```
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
    {
      double local_sum = 0.0;
#pragma omp for nowait
       for (i = 1; i <= n;  i++) {
              x = h * ((double)i - 0.5);
              local_sum += (4.0 / (1.0 + x*x));
        }
#pragma omp atomic
              sum += local_sum;
     }
     pi = h * sum;
     printf("pi is approximately %.16f", pi);
     return 0;
}
```

Использование директивы atomic

```
int atomic_read(const int *p)
{
  int value;
  /* Guarantee that the entire value of *p is read atomically. No part of
  * *p can change during the read operation.
  */
  #pragma omp atomic read
  value = *p;
  return value;
}

void atomic_write(int *p, int value)
{
  /* Guarantee that value is stored atomically into *p. No part of *p can change
  * until after the entire write operation is completed.
  */
  #pragma omp atomic write
  *p = value;
}
```

Использование директивы atomic

```
int fetch_and_add(int *p)
{
  /* Atomically read the value of *p and then increment it. The previous value is
  * returned. */
  int old;
  #pragma omp atomic capture
  { old = *p; (*p)++; }
  return old;
}
```

seq_cst - sequentially consistent atomic construct, the operation to have the same meaning as a memory_order_seq_cst atomic operation in C++11/C11

```
#pragma omp atomic capture seq_cst  // OpenMP 4.0
{--x; v = x;} // capture final value of x in v and flush all variables
```

Семафоры

Концепцию семафоров описал Дейкстра (Dijkstra) в 1965

Семафор - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:

P - функция запроса семафора
P(s): [if (s == 0) <заблокировать текущий процесс>; else s = s-1;]

V - функция освобождения семафора
V(s): [if (s == 0) <разблокировать один из заблокированных процессов>; s = s+1;]

Семафоры в OpenMP

Состояния семафора:

- ❑uninitialized
- ❑unlocked
- ❑locked

```
void omp_init_lock(omp_lock_t *lock); /* uninitialized to unlocked*/
void omp_destroy_lock(omp_lock_t *lock); /* unlocked  to uninitialized */
void omp_set_lock(omp_lock_t *lock);  /*P(lock)*/
void omp_unset_lock(omp_lock_t *lock); /*V(lock)*/
int omp_test_lock(omp_lock_t *lock);

void omp_init_nest_lock(omp_nest_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Вычисление числа π с использованием семафоров

```
int main ()
{
   int n =100000, i; double pi, h, sum, x;
   omp_lock_t lck;
   h   = 1.0 / (double) n;
   sum = 0.0;
   omp_init_lock(&lck);
#pragma omp parallel default (none) private (i,x) shared (n,h,sum,lck)
   {
      double local_sum = 0.0;
      #pragma omp for nowait
      for (i = 1; i <= n;  i++) {
            x = h * ((double)i - 0.5);
            local_sum += (4.0 / (1.0 + x*x));
      }
      omp_set_lock(&lck);
            sum += local_sum;
      omp_unset_lock(&lck);
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    omp_destroy_lock(&lck);
    return 0;
}
```

Использование семафоров

```
#include <stdio.h>
#include <omp.h>
int main()
{
    omp_lock_t lck;
    int id;
    omp_init_lock(&lck);
    #pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();
        omp_set_lock(&lck);
        printf("My thread id is %d.\n", id); /* only one thread at a time can execute this printf */
        omp_unset_lock(&lck);
        while (! omp_test_lock(&lck)) {
                skip(id); /* we do not yet have the lock, so we must do something else*/
        }
        work(id); /* we now have the lock and can do the work */
        omp_unset_lock(&lck);
    }
    omp_destroy_lock(&lck);
    return 0;
}
```

```
void skip(int i) {}
void work(int i) {}
```

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_lock_t lck;
} pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_lock(&p->lck);
    p->b += b;
    omp_unset_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
   #pragma omp parallel sections
   {
      #pragma omp section
         incr_pair(p,1,2);
      #pragma omp section
         incr_b(p,3);
   }
}
```

Deadlock!

Использование семафоров

```
#include <omp.h>
typedef struct {
    int a,b;
    omp_nest_lock_t lck;
} pair;
void incr_a(pair *p, int a)
{
    p->a += a;
}
void incr_b(pair *p, int b)
{
    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```

```
void incorrect_example(pair *p)
{
   #pragma omp parallel sections
   {
      #pragma omp section
         incr_pair(p,1,2);
      #pragma omp section
         incr_b(p,3);
   }
}
```

Директива barrier

Точка в программе, достижимая всеми нитями группы, в которой выполнение программы приостанавливается до тех пор пока все нити группы не достигнут данной точки и все задачи, выполняемые группой нитей будут завершены.

#pragma omp barrier

По умолчанию барьерная синхронизация нитей выполняется:

- по завершению конструкции **parallel**;
- при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**) , если не указана клауза **nowait**.

```
#pragma omp parallel
{
    #pragma omp master
    {
        int i, size;
        scanf("%d",&size);
        for (i=0; i<size; i++) {
            #pragma omp task
                process(i);
        }
    }
    #pragma omp barrier
}
```

Директива taskyield

```
#include <omp.h>
void something_useful ( void );
void something_critical ( void );
void foo ( omp_lock_t * lock, int n )
{
    int i;
    for ( i = 0; i < n; i++ )
        #pragma omp task
        {
            something_useful();
            while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

Директива taskwait

```
#pragma omp taskwait

int fibonacci(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
            i=fibonacci (n-1);
        #pragma omp task shared(j)
            j=fibonacci (n-2);
        #pragma omp taskwait
        return i+j;
    }
}
```

```
int main () {
    int res;
    #pragma omp parallel
    {
        #pragma omp single
        {
            int n;
            scanf("%d",&n);
            #pragma omp task shared(res)
                res = fibonacci(n);
        }
    }
    printf ("Finonacci number =  %d\n", res);
}
```

Директива taskwait

```
#pragma omp task {}  // Task1
#pragma omp task // Task2
{
    #pragma omp task {} // Task3
}
#pragma omp task {} // Task4

#pragma omp taskwait
// Гарантируется что в данной точке завершатся Task1 && Task2  && Task4
```

Директива taskgroup

```
#pragma omp task {}  // Task1
#pragma omp taskgroup
{
   #pragma omp task // Task2
   {
      #pragma omp task {} // Task3
   }
   #pragma omp task {} // Task4
}
// Гарантируется что в данной точке завершатся Task2 && Task3  && Task4
```

Использование директивы taskgroup

```
struct tree_node
{
   struct tree_node *left, *right;
   float *data;
};
typedef struct tree_node* tree_type;
void compute_tree(tree_type tree)
{
   if (tree->left)
   {
     #pragma omp task
        compute_tree(tree->left);
   }
   if (tree->right)
   {
     #pragma omp task
        compute_tree(tree->right);
   }
   #pragma omp task
       compute_something(tree->data);
}
```

```
int main()
{
  tree_type tree;
  init_tree(tree);
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task
       start_background_work();
    #pragma omp taskgroup
    {
      #pragma omp task
        compute_tree(tree);
    }
    print_something ();
  } // only now background work is required
}   // to be complete
```

Содержание

- Тенденции развития современных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ
- Новые возможности OpenMP

Внутренние переменные, управляющие выполнением OpenMP-программы (ICV-Internal Control Variables)

Для параллельных областей:

- nthreads-var
- thread-limit-var
- dyn-var
- nest-var
- max-active-levels-var

Для циклов:

- run-sched-var
- def-sched-var

Для всей программы:

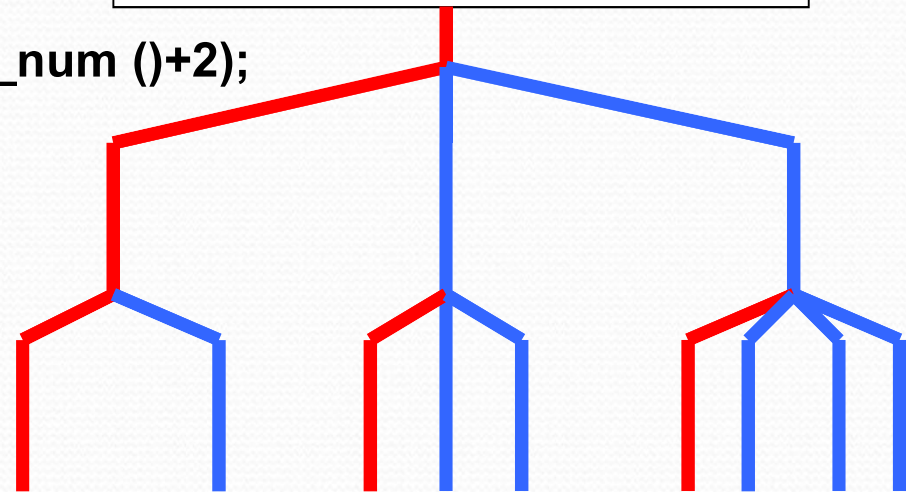
- stacksize-var
- wait-policy-var
- bind-var
- cancel-var
- place-partition-var

Internal Control Variables. nthreads-var

```
void work();

int main () {
    omp_set_num_threads(3);
    #pragma omp parallel
    {
        omp_set_num_threads(omp_get_thread_num ()+2);
        #pragma omp parallel
            work();
    }
}
```

Не корректно в OpenMP 2.5

Корректно в OpenMP 3.0



Существует одна копия этой переменной для каждой задачи

Internal Control Variables. nthreads-var

Определяет максимально возможное количество нитей в создаваемой параллельной области.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_NUM_THREADS 4,3,2

Korn shell:

export OMP_NUM_THREADS=16

Windows:

set OMP_NUM_THREADS=16

void omp_set_num_threads(int num_threads);

Узнать значение переменной можно:

int omp_get_max_threads(void);

Internal Control Variables. thread-limit-var

Определяет максимальное количество нитей, которые могут быть использованы для выполнения всей программы.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

C shell:

setenv OMP_THREAD_LIMIT 16

Korn shell:

export OMP_THREAD_LIMIT=16

Windows:

set OMP_THREAD_LIMIT=16

Узнать значение переменной можно:

int omp_get_thread_limit(void)

Internal Control Variables. dyn-var

Включает/отключает режим, в котором количество создаваемых нитей при входе в параллельную область может меняться динамически.

Начальное значение: Если компилятор не поддерживает данный режим, то false.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_DYNAMIC true

Korn shell:

export OMP_DYNAMIC=true

Windows:

set OMP_DYNAMIC=true

void omp_set_dynamic(int dynamic_threads);

Узнать значение переменной можно:

int omp_get_dynamic(void);

Internal Control Variables. nest-var

Включает/отключает режим поддержки вложенного параллелизма.

Начальное значение: **false**.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_NESTED true

Korn shell:

export OMP_NESTED=false

Windows:

set OMP_NESTED=true

void omp_set_nested(int nested);

Узнать значение переменной можно:

int omp_get_nested(void);

Internal Control Variables. max-active-levels-var

Задает максимально возможное количество активных вложенных параллельных областей.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_MAX_ACTIVE_LEVELS 2

Korn shell:

export OMP_MAX_ACTIVE_LEVELS=3

Windows:

set OMP_MAX_ACTIVE_LEVELS=4

void omp_set_max_active_levels (int max_levels);

Узнать значение переменной можно:

int omp_get_max_active_levels(void);

Internal Control Variables. run-sched-var

Задает способ распределения витков цикла между нитями, если указана клауза **schedule(runtime)**.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для каждой задачи.

Значение переменной можно изменить:

C shell:

setenv OMP_SCHEDULE "guided,4"

Korn shell:

export OMP_SCHEDULE "dynamic,5"

Windows:

set OMP_SCHEDULE=static

```
typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
} omp_sched_t;
```

void omp_set_schedule(omp_sched_t kind, int modifier);

Узнать значение переменной можно:

void omp_get_schedule(omp_sched_t * kind, int * modifier);

Internal Control Variables. def-sched-var

Задает способ распределения витков цикла между нитями по умолчанию.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

```
void work(int i);

int main () {
    #pragma omp parallel
    {
        #pragma omp for
            for (int i=0;i<N;i++)  work (i);
    }
}
```

Internal Control Variables. stack-size-var

Каждая нить представляет собой независимо выполняющийся поток управления со своим счетчиком команд, регистровым контекстом и стеком.

Переменная ***stack-size-var*** задает размер стека.

Начальное значение: зависит от реализации.

Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:

setenv OMP_STACKSIZE 2000500B

setenv OMP_STACKSIZE "3000 k"

setenv OMP_STACKSIZE 10M

setenv OMP_STACKSIZE "10 M"

setenv OMP_STACKSIZE "20 m"

setenv OMP_STACKSIZE "1G"

setenv OMP_STACKSIZE 20000 # Size in Kilobytes

Internal Control Variables. stack-size-var

```
int main () {
    int a[1024][1024];
    #pragma omp parallel private (a)
    {
        for (int i=0;i<1024;i++)
            for (int j=0;j<1024;j++)
                a[i][j]=i+j;
    }
}
```

icl /Qopenmp test.cpp

⇒Program Exception – stack overflow

Linux: ulimit -a

ulimit -s <stacksize in Kbytes>

Windows: /F<stacksize in bytes>

-Wl,--stack, <stacksize in bytes>

setenv KMP_STACKSIZE 10m

setenv GOMP_STACKSIZE 10000

setenv OMP_STACKSIZE 10M