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

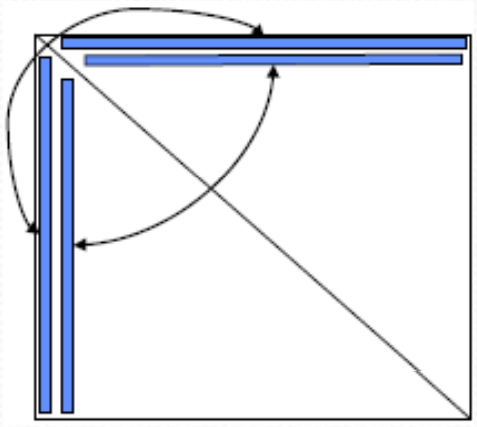
Распределение витков цикла. Клауза schedule

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

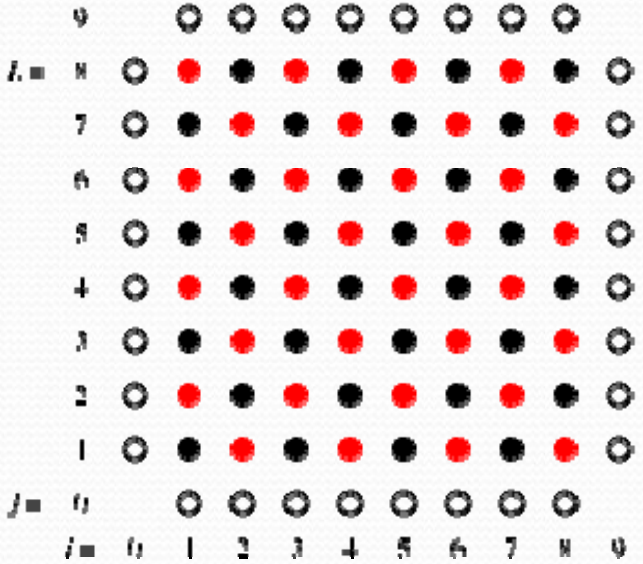
```
#pragma omp parallel for ordered
for(int i = 1; i < 100; i++) {
    #pragma omp ordered
      {
        a[i]= a[i-1] + a[i+1];
      }
}
```

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;
#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

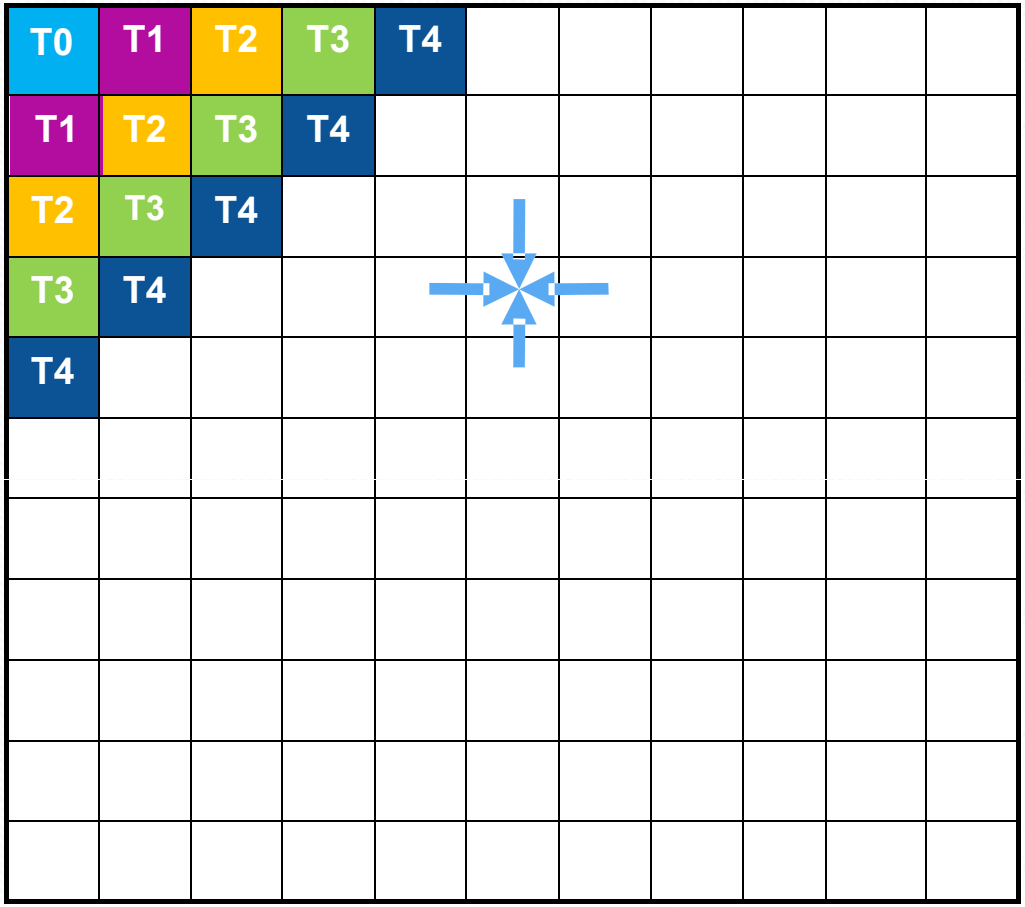
```
for(int i = 1; i < M; i++)
   for(int j = 1; j < N; j++)
       a[i][j] = (a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]) / 4;
```



```
#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;
}
```

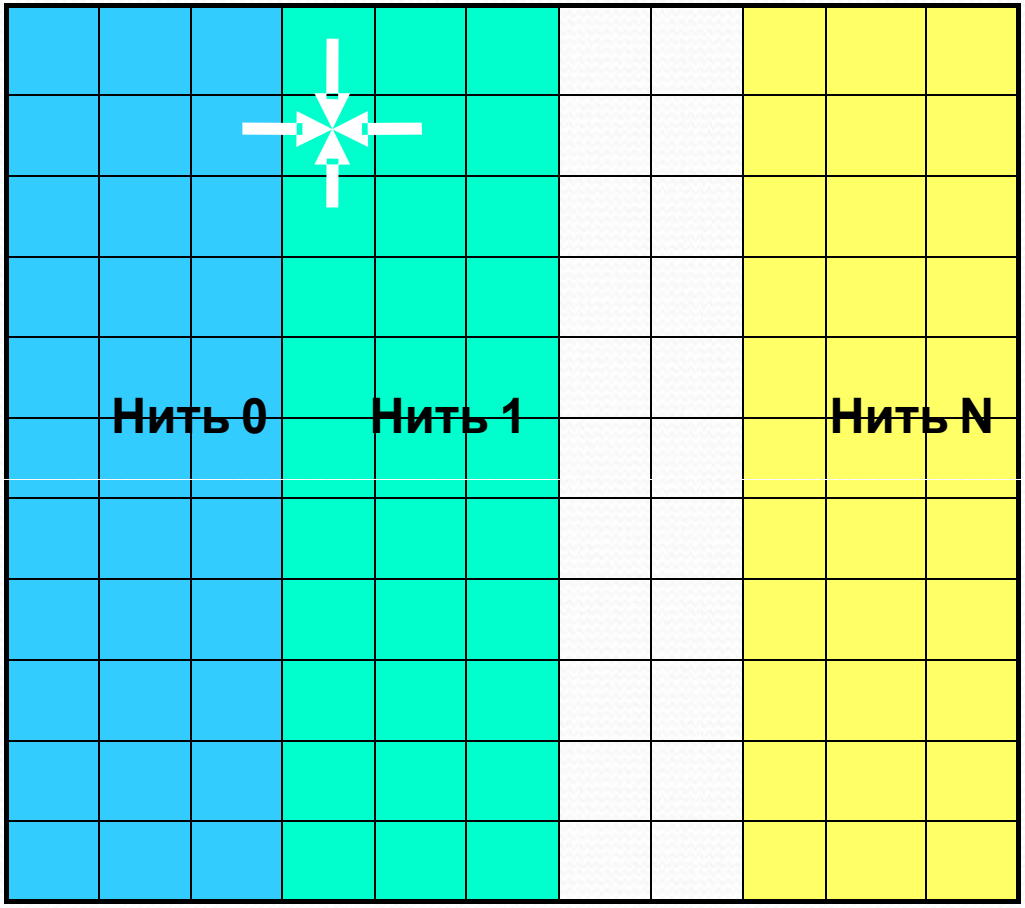
Распределение циклов с зависимостью по данным. Параллелизм по гиперплоскостям

```
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;
```



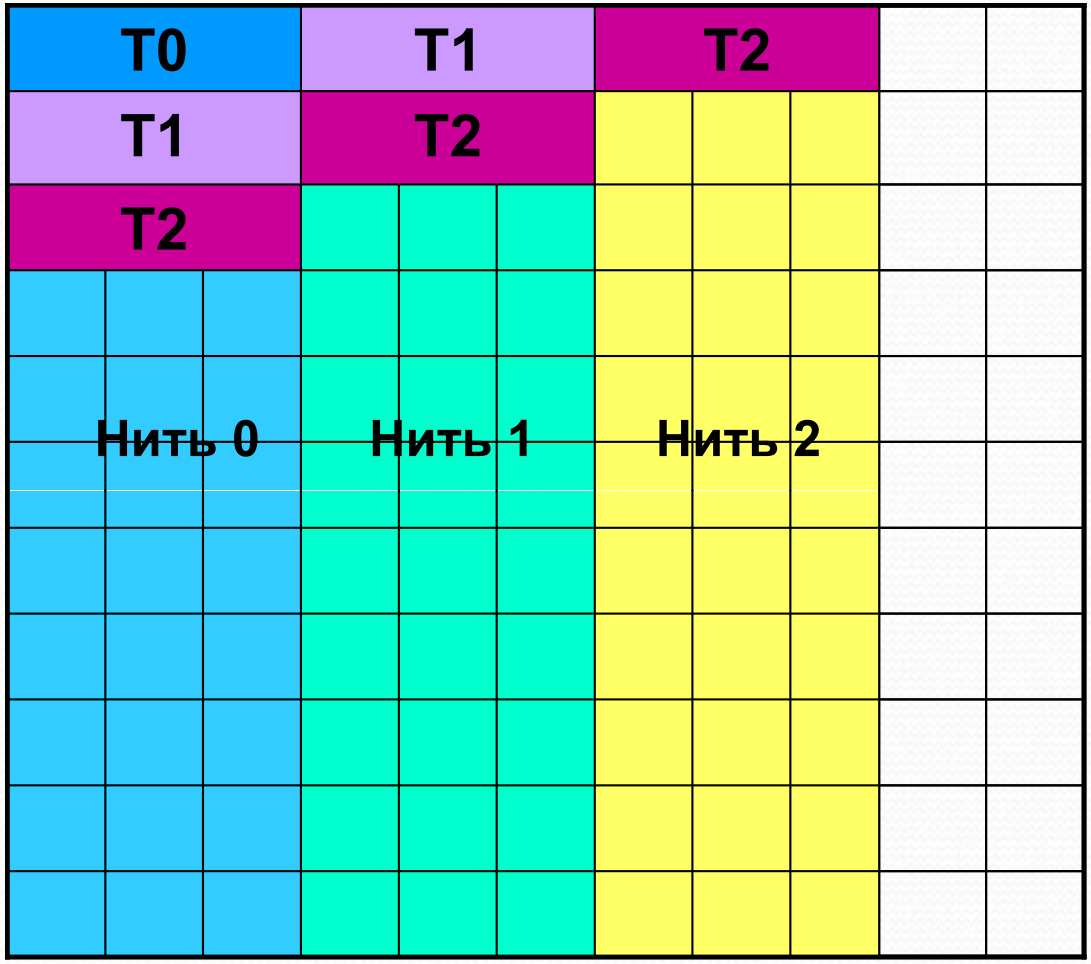
Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла

```
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;
```



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