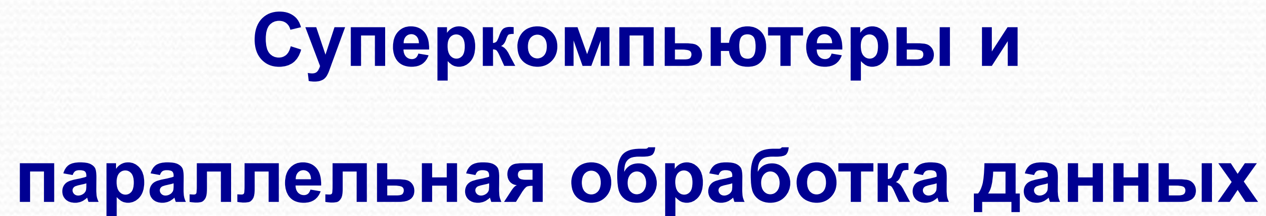


Бахтин Владимир Александрович
к.ф.-м.н., ведущий научный сотрудник
Института прикладной математики им М.В.Келдыша РАН
кафедра системного программирования
факультет вычислительной математики и кибернетики
Московского университета им. М.В. Ломоносова

Распределение 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**

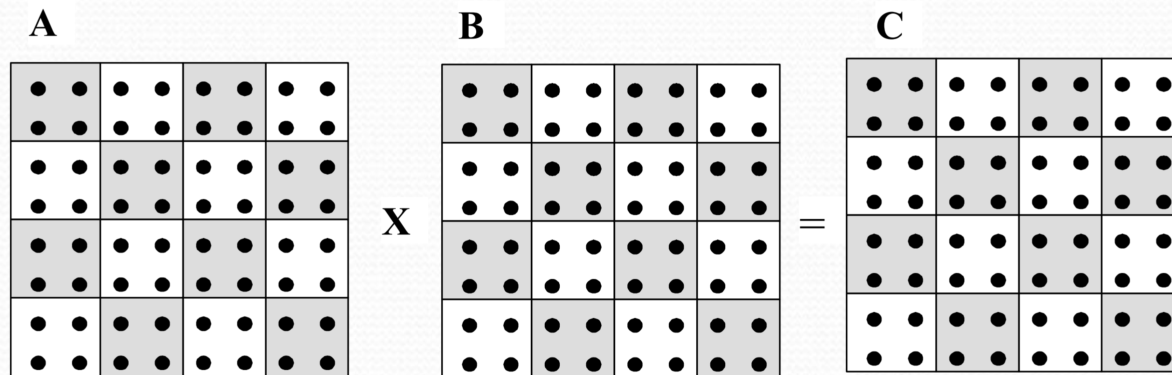
```
int i, y, a[100];

#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);
}
```

Блочное умножение матриц



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

Блочное умножение матриц

```
void matmul (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];
         }
      }
   }
}
```

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;
#pragma omp parallel
#pragma omp single
  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);
   }
}
```

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

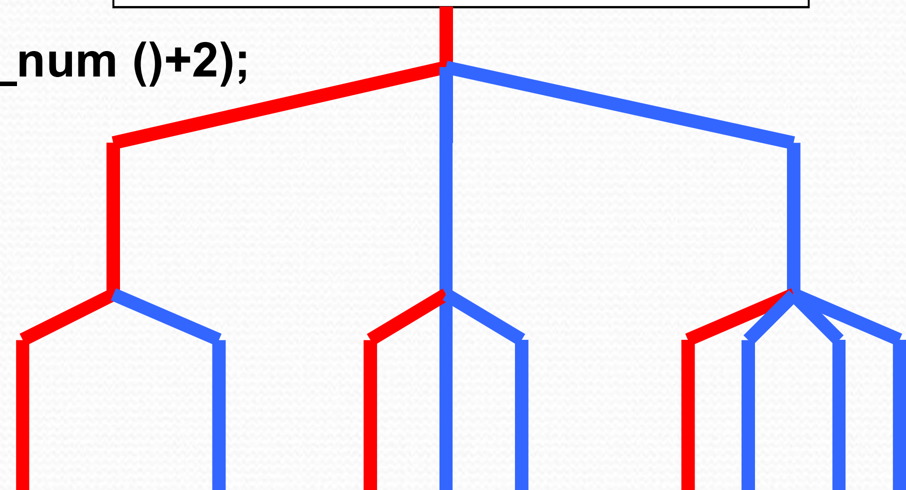
Internal Control Variables. nthreads-var

```
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

Internal Control Variables. wait-policy-var

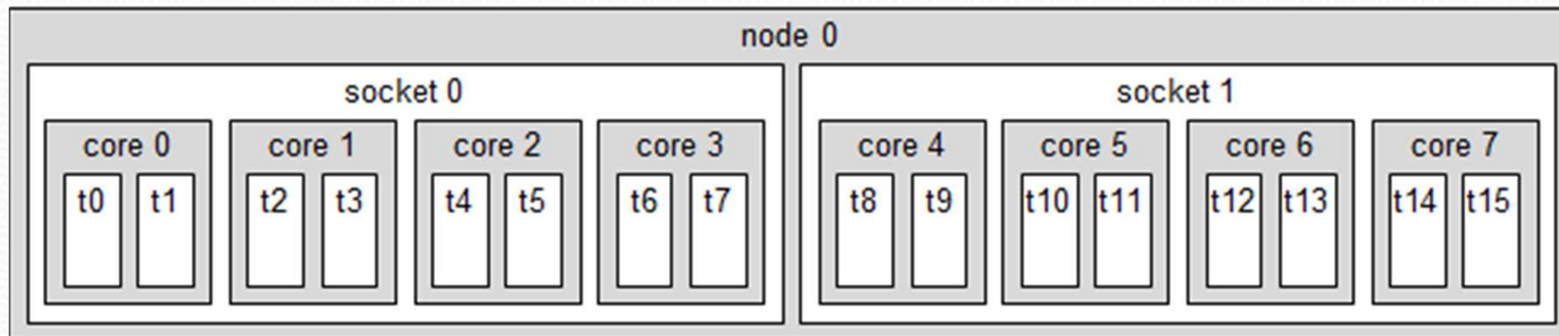
Подсказка OpenMP-компилятору о желаемом поведении нитей во время ожидания.
Начальное значение: зависит от реализации.
Существует одна копия этой переменной для всей программы.

Значение переменной можно изменить:
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive

IBM AIX
SPINLOOPTIME=100000
YIELDLOOPTIME=40000

Internal Control Variables. place-partition-var

Распределение OpenMP-нитей по ядрам.
Начальное значение: зависит от реализации.
Существует одна копия этой переменной для всей программы.



setenv OMP_PLACES "threads(2)"
setenv OMP_PLACES "{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"
setenv OMP_PLACES "{0:2},{2:2},{4:2} ,{6:2},{8:2},{10:2},{12:2},{14:2}"
setenv OMP_PLACES "{0:2}:8:2"

Internal Control Variables. Приоритеты

клауза	вызов функции	переменная окружения	ICV
	omp_set_dynamic()	OMP_DYNAMIC	*dyn-var*
	omp_set_nested()	OMP_NESTED	*nest-var*
num_threads	omp_set_num_threads()	OMP_NUM_THREADS	*nthreads-var*
schedule	omp_set_schedule()	OMP_SCHEDULE	*run-sched-var*
schedule			*def-sched-var*
		OMP_STACKSIZE	*stacksize-var*
		OMP_WAIT_POLICY	*wait-policy-var*
		OMP_THREAD_LIMIT	*thread-limit-var*
	omp_set_max_active_ levels()	OMP_MAX_ACTIVE_ LEVELS	*max-active-levels-var*

Система поддержки выполнения OpenMP-программ

int omp_get_num_threads(void);
-возвращает количество нитей в текущей параллельной области

```
#include <omp.h>
void work(int i);
void test()
{
     int np;
     np = omp_get_num_threads(); /* np == 1*/
     #pragma omp parallel private (np)
     {
        np = omp_get_num_threads();
        #pragma omp for schedule(static)
        for (int i=0; i < np; i++)
             work(i);
      }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_thread_num(void);

-возвращает номер нити в группе [0: omp_get_num_threads()-1]

```
#include <omp.h>
void work(int i);
void test()
{
    int iam;
    iam = omp_get_thread_num(); /* iam == 0*/
    #pragma omp parallel private (iam)
    {
       iam = omp_get_thread_num();
       work(iam);
     }
}
```

Система поддержки выполнения OpenMP-программ

```
int omp_get_num_procs(void);
```

-возвращает количество процессоров, на которых программа выполняется

```
#include <omp.h>
void work(int i);
void test()
{
    int nproc;
    nproc = omp_get_num_ procs();
   #pragma omp parallel num_threads(nproc)
    {
        int iam = omp_get_thread_num();
        work(iam);
     }
}
```

Система поддержки выполнения OpenMP-программ

Распределение OpenMP-нитей по ядрам.

```
int omp_get_num_places(void);
int omp_get_place_num(void);

int main()
{
    int n_sockets, socket_num;
    omp_set_nested(1);                // or export OMP_NESTED=true
    omp_set_max_active_levels(2);   // or export OMP_MAX_ACTIVE_LEVELS=2
    n_sockets = omp_get_num_places();
    #pragma omp parallel num_threads(n_sockets) private(socket_num)
    {
            socket_num = omp_get_place_num();
            socket_init(socket_num);
    }
}
```

Система поддержки выполнения OpenMP-программ

Распределение OpenMP-нитей по ядрам.

```
int omp_get_place_num_procs(int place_num);

void socket_init(int socket_num)
{
    int n_procs;
    n_procs = omp_get_place_num_procs(socket_num);
    #pragma omp parallel num_threads(n_procs) proc_bind(close)
    {
            printf("Reporting in from socket num, thread num:  %d %d\n",
            socket_num,omp_get_thread_num() );
    }
}


void omp_get_place_proc_ids(int place_num, int  *ids);
```

Система поддержки выполнения OpenMP-программ

int omp_get_level(void)

- возвращает уровень вложенности для текущей параллельной области.

```
#include <omp.h>
void work(int i) {
 #pragma omp parallel
    {
       int ilevel = omp_get_level ();
    }
}
void test()
{
     int ilevel = omp_get_level (); /*ilevel==0*/
    #pragma omp parallel private (ilevel)
    {
        ilevel =  omp_get_level ();
        int iam = omp_get_thread_num();
        work(iam);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_active_level(void)

- возвращает количество активных параллельных областей (выполняемых 2-мя или более нитями).

```
#include <omp.h>
void work(int iam, int size) {
 #pragma omp parallel
    {
        int ilevel = omp_get_active_level ();
    }
}
void test()
{
    int size = 0;
    int ilevel = omp_get_active_level (); /*ilevel==0*/
    scanf("%d",&size);
    #pragma omp parallel if (size>10)
    {
        int iam = omp_get_thread_num();
         work(iam, size);
     }
}
```

Система поддержки выполнения OpenMP-программ

int omp_get_ancestor_thread_num (int level)

- для нити, вызвавшей данную функцию, возвращается номер нити-родителя, которая создала указанную параллельную область.

```
omp_get_ancestor_thread_num (0)  = 0

If (level==omp_get_level()) {
    omp_get_ancestor_thread_num (level) == omp_get_thread_num ();
}

If ((level<0)||(level>omp_get_level())) {
    omp_get_ancestor_thread_num (level) == -1;
}
```

Система поддержки выполнения OpenMP-программ

```
int omp_get_team_size(int level);
 - количество нитей в указанной параллельной области.

omp_get_team_size (0)  = 1

If (level==omp_get_level()) {
    omp_get_team_size (level) == omp_get_num _threads ();
}

If ((level<0)||(level>omp_get_level())) {
    omp_get_team_size (level) == -1;
}
```

Система поддержки выполнения OpenMP-программ. Функции работы со временем

double omp_get_wtime(void);

возвращает для нити астрономическое время в секундах, прошедшее с некоторого момента в прошлом. Если некоторый участок окружить вызовами данной функции, то разность возвращаемых значений покажет время работы данного участка. Гарантируется, что момент времени, используемый в качестве точки отсчета, не будет изменен за время выполнения программы.

```
double start;
double end;
start = omp_get_wtime();
/*... work to be timed ...*/
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

double omp_get_wtick(void);

- возвращает разрешение таймера в секундах (количество секунд между последовательными импульсами таймера).

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

Новые возможности OpenMP

- Векторизация кода
- Обработка исключительных ситуаций / cancellation constructs
- Поддержка ускорителей/сопроцессоров

Использование векторизации

```
void add_float (float *a, float *b, float *c, float *d, float *e, int n)
{
    for (int i=0; i<n; i++)
      a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Использование векторизации

```
void add_float (float *restrict a, float *restrict b, float *restrict c,
float *restrict d, float *restrict e, int n) // C99
{
    for (int i=0; i<n; i++)
      a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Использование векторизации. Спецификация simd

#pragma omp simd *[clause[[,] clause]..]*
 for-loops

#pragma omp for simd *[clause[[,] clause]..]*
 for-loops

где клауза одна из:

- **safelen (length)**
- **linear (list[:linear-step])**
- **aligned (list[:alignment])**
- **private (list)**
- **lastprivate (list)**
- **reduction (reduction-identifier: list)**
- **collapse (n)**

Использование векторизации

```
void add_float (float *a, float *b, float *c, float *d, float *e, int n)
{
  #pragma omp simd
  for (int i=0; i<n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}


void add_float (float *restrict a, float *restrict b, float *restrict c,
float *restrict d, float *restrict e, int n) // C99
{
   for (int i=0; i<n; i++)
    a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Использование векторизации. Спецификация declare simd

#pragma omp declare simd *[clause[[,] clause]..]*
function definition or declaration

где клауза одна из:

- **simdlen (length)**
  the largest size for a vector that the compiler is free to assume
- **linear (argument-list[:constant-linear-step])**
  in serial execution parameters are incremented by steps (induction variables with constant stride)
- **aligned (argument-list[:alignment])**
  all arguments in the argument-list are aligned on a known boundary not less than the specified alignment
- **uniform (argument-list)**
  shared, scalar parameters are broadcasted to all iterations
- **inbranch**
- **notinbranch**

Использование векторизации

```
#pragma omp declare simd notinbranch
float min(float a, float b) {
  return a < b ? a : b;
}

#pragma omp declare simd inbranch
float distance (float x, float y) {
  return (x - y) * (x - y);
}

#pragma omp parallel for simd
for (i=0; i<N; i++)
  d[i] = min (distance (a[i], b[i]), c[i]);
```

Cancellation Constructs

Директива

#pragma omp cancel ***clause[[,] clause]***

где ***clause*** *одна из:*

- **parallel**
- **sections**
- **for**
- **taskgroup**
- **if (*scalar-expression*)**

Директива

#pragma omp cancellation point ***clause[[,] clause]***

где ***clause*** *одна из:*

- **parallel**
- **sections**
- **for**
- **taskgroup**

Новая функция системы поддержки:

- **omp_get_cancellation**

Новая переменная окружения:

- **OMP_CANCELLATION**

Обработка исключительных ситуаций

```
void example() {
    std::exception *ex = NULL;
    #pragma omp parallel shared(ex)
    {
        #pragma omp for schedule(runtime)
        for (int i = 0; i < N; i++) {
            try {
                causes_an_exception();
            } catch (std::exception *e) {
                #pragma omp atomic write
                   ex = e; // still must remember exception for later handling
               #pragma omp cancel for // cancel worksharing construct
            }
        }
        if (ex) { // if an exception has been raised, cancel parallel region
           #pragma omp cancel parallel
        }
   }
   if (ex) {  // handle exception stored in ex
   }
}
```

Поиск в дереве (часть 1)

```
typedef struct binary_tree_s {
   int value;
   struct binary_tree_s *left, *right;
} binary_tree_t;

binary_tree_t *search_tree_parallel (binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
    {
#pragma omp taskgroup
        {
#pragma omp master
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}
```

Поиск в дереве (часть 2)

```
binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
    binary_tree_t *found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        } else {
            #pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_left = NULL;
                found_left = search_tree(tree->left, value, level + 1);
                if (found_left) {
                    #pragma omp atomic write
                      found = found_left;
                    #pragma omp cancel taskgroup
                }
            }
```

Поиск в дереве (часть 3)

```
        #pragma omp task shared(found) if(level < 10)
        {
            binary_tree_t *found_right = NULL;
            found_right = search_tree(tree->right, value, level + 1);
            if (found_right) {
                #pragma omp atomic write
                    found = found_right;
                #pragma omp cancel taskgroup
            }
        }
       #pragma omp taskwait
      }
    }
    return found;
}
```