Технология параллельного программирования OpenMP

Бахтин Владимир Александрович
к.ф.-м.н., зав. сектором Института прикладной математики им М.В.Келдыша РАН
ассистент кафедры системного программирования факультета вычислительной математики и кибернетики Московского университета им. М.В. Ломоносова

Содержание

- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

OpenMP- модель параллелизма по управлению

- Основные понятия
- Выполнение OpenMP-программы
- Модель памяти в OpenMP
- Классы переменных
- Параллельная область

Обзор основных возможностей OpenMP

```
C$OMP FLUSH
C$OMP THREADPRIVATE(/ABC/)
C$OMP PARALLEL DO SHARED(A,B,C
CALL OMP_INIT_LOCK (LCK)
C$OMP SINGLE PRIVATE(X)
SET
C$OMP PARALLEL DO ORDERED PRI
C$OMP PARALLEL  REDUCTION (+:
#pragma omp parallel for private(a, b)
C$OMP BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO LASTPRIVATE(XX)
nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

OpenMP: API для написания многонитевых приложений

- Множество директив компилятора, набор функции библиотеки системы поддержки, переменные окружения
- Облегчает создание многонитиевых программ на Фортране, C и C++
- Обобщение опыта создания параллельных программ для SMP и DSM систем за последние 20 лет

Директивы и клаузы

Спецификации параллелизма в OpenMP представляют собой директивы вида:

#pragma omp название-директивы*[клауза[[,]клауза]...]*

Например:

#pragma omp parallel default (none) shared (i,j)

Исполняемые директивы:

- ***barrier***
- ***taskwait***
- ***flush***

Описательная директива:

- ***threadprivate***

Структурный блок

Действие остальных директив распространяется на структурный блок:

#pragma omp название-директивы*[клауза[[,]клауза]...]*

{

структурный блок

}

Структурный блок: блок кода с одной точкой входа и одной точкой выхода.

```
#pragma omp parallel
{
  …
  mainloop: res[id] = f (id);
  if (res[id] != 0) goto mainloop;
  …
  exit (0);
}
```

Структурный блок

```
#pragma omp parallel
{
  …
  mainloop: res[id] = f (id);
  …
}
if (res[id] != 0) goto mainloop;
```

Не структурный блок

Компиляция OpenMP-программы

Производитель	Компилятор	Опция компиляции
GNU	gcc	-fopenmp
IBM	XL C/C++ / Fortran	-qsmp=omp
Sun Microsystems	C/C++ / Fortran	-xopenmp
Intel	C/C++ / Fortran	-openmp /Qopenmp
Portland Group	C/C++ / Fortran	-mp
Microsoft	Visual Studio 2008 C++	/openmp

Условная компиляция OpenMP-программы

```
#include <stdio.h>
int main()
{
#ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
#endif
    return 0;
}
```

В значении переменной _OPENMP зашифрован год и месяц (yyyymm) версии стандарта OpenMP, которую поддерживает компилятор.

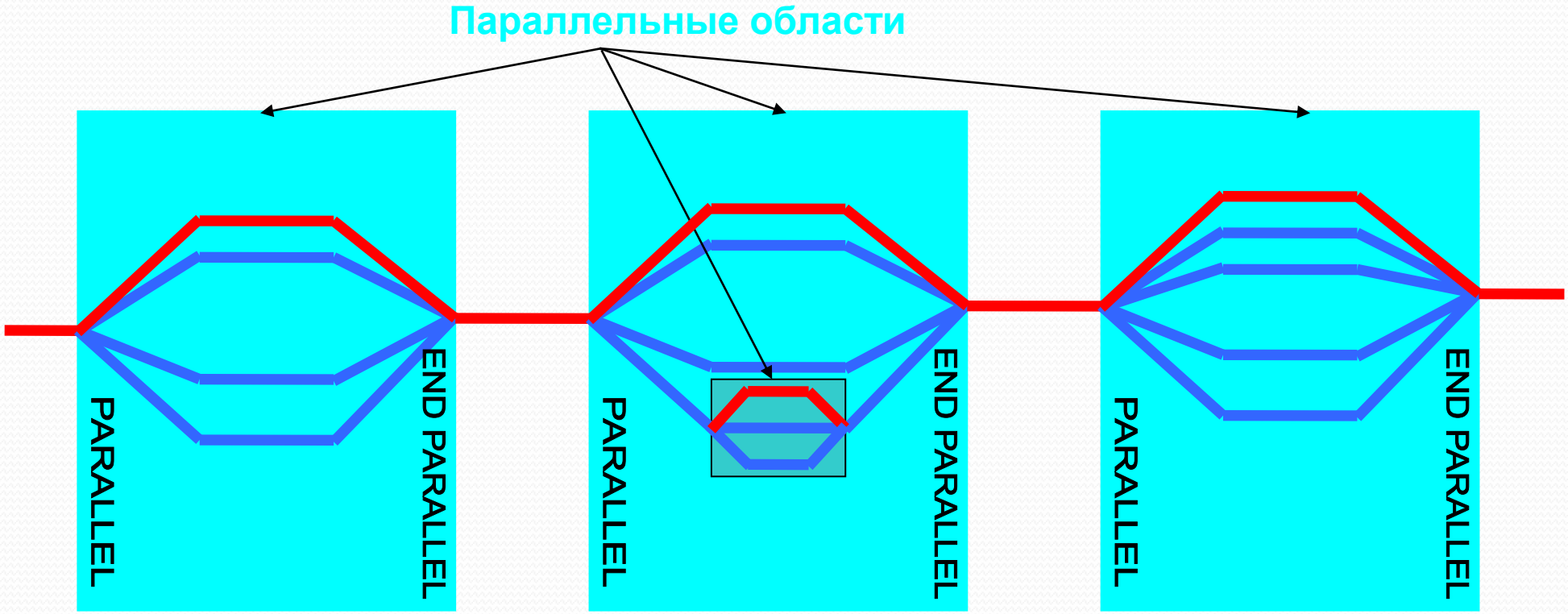
Использование функций поддержи выполнения OpenMP-программ (OpenMP API runtime library)

```
#include <stdio.h>
#include <omp.h> // Описаны прототипы всех функций и типов
int main()
{
#pragma omp parallel
  {
    int id = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    printf("Thread (%d) of (%d) threads alive\n", id, numt);
   }
  return 0;
}
```

Выполнение OpenMP-программы

Fork-Join параллелизм:

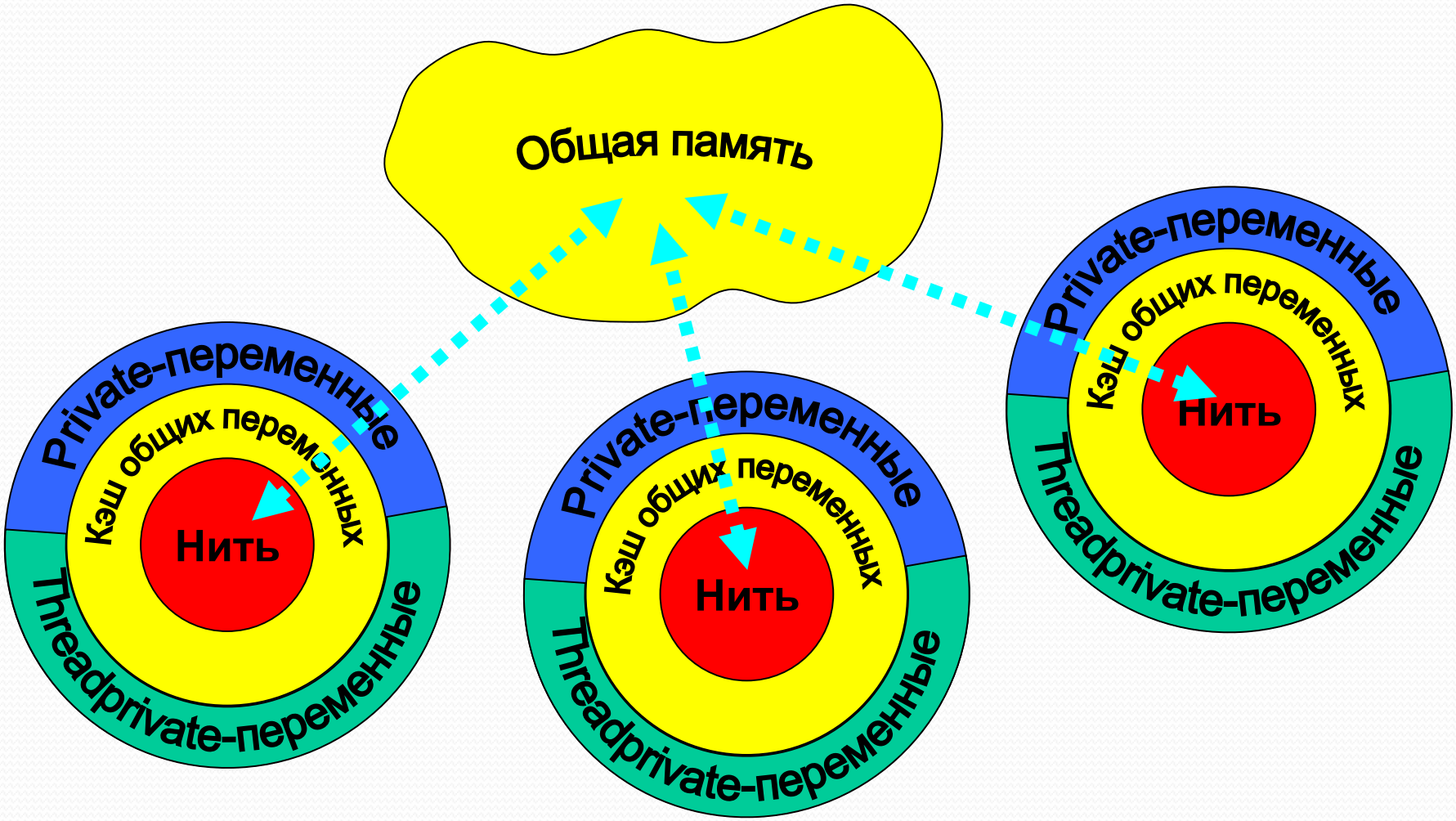
- Главная (master) нить порождает группу (team) нитей по мере небходимости.
- Параллелизм добавляется инкрементально.



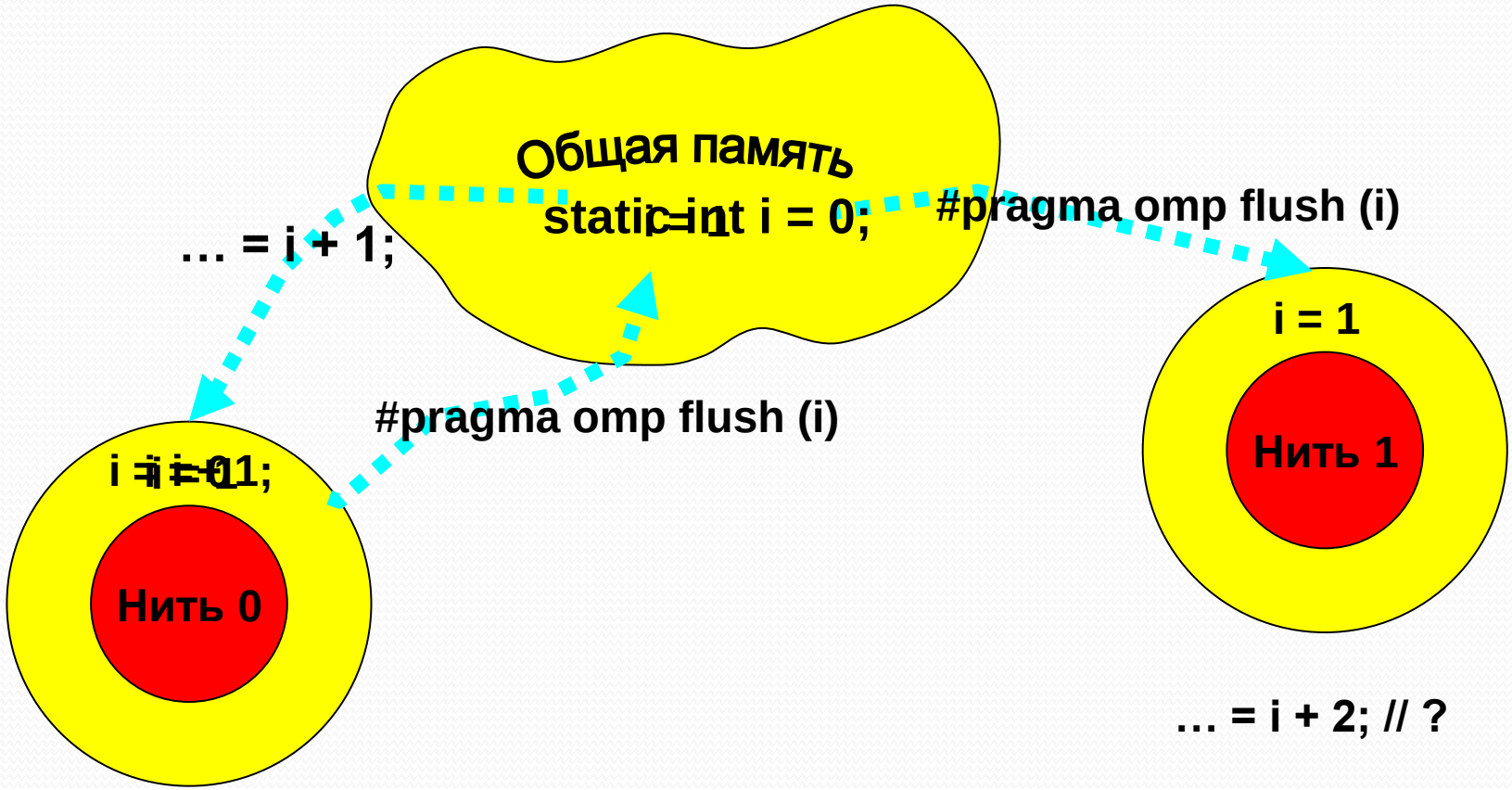
Фрагмент программы

```
int a, b, c, d, x, y;              // переменные
int *p, *q;                        // указатели
int f(int *p, int *q);             // прототип функции
a = x * (x - 1);                   // a хранится в регистре
b = y * (y + 1);                   // b хранится в регистре
c = a * a + a * b + b * b;         // будет использовано позднее
d = a * b * c;                     // будет использовано позднее
p = &a;                            // получает адрес a
q = &b;                            // получает адрес b
x = f(p, q);                       // вызов функции\
```

Модель памяти в OpenMP



Модель памяти в OpenMP



Консистентность памяти в OpenMP

Корректная последовательность работы нитей с переменной:

Нить0 записывает значение переменной - write(var)

Нить0 выполняет операцию синхронизации – flush (var)

Нить1 выполняет операцию синхронизации – flush (var)

Нить1 читает значение переменной – read (var)

Директива flush:

#pragma omp flush *[(list)]* - для Си

!$omp flush *[(list)]* - для Фортран

Консистентность памяти в OpenMP

1. Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).
2. Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке, определяемом программой.
3. Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

Директива flush

#pragma omp flush *[(список переменных)]*

По умолчанию все переменные приводятся в консистентное состояние **(#pragma omp flush**):

- При барьерной синхронизации
- При входе и выходе из конструкций **parallel**, **critical** и **ordered**.
- При выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**), если не указана клауза **nowait**.
- При вызове **omp_set_lock** и **omp_unset_lock**.
- При вызове **omp_test_lock**, **omp_set_nest_lock**, **omp_unset_nest_lock**
- и **omp_test_nest_lock**, если изменилось состояние семафора.

При входе и выходе из конструкции **atomic** выполняется **#pragma omp flush(x)**, где x – переменная, изменяемая в конструкции **atomic**.

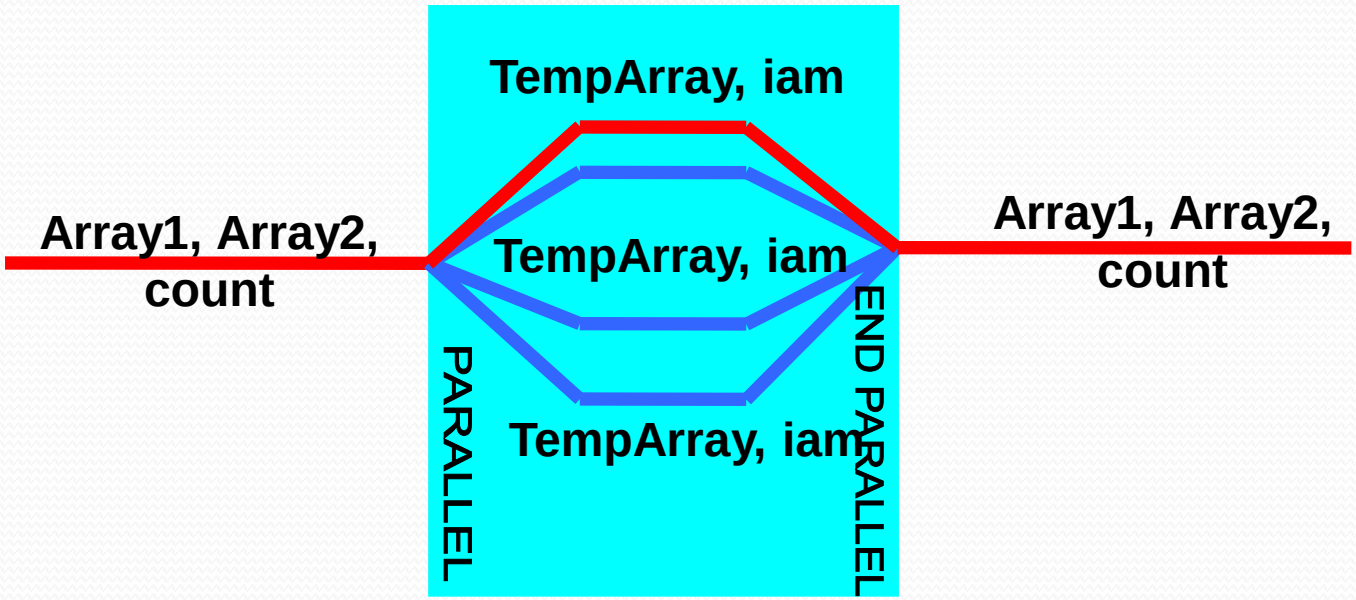
Классы переменных

- В модели программирования с разделяемой памятью:
  - Большинство переменных по умолчанию считаются **shared**
- Глобальные переменные совместно используются всеми нитями (shared) :
  - Фортран: COMMON блоки, SAVE переменные, MODULE переменные
  - Си: file scope, static
  - Динамически выделяемая память (ALLOCATE, malloc, new)
- Но не все переменные являются разделяемыми ...
  - Стековые переменные в подпрограммах (функциях), вызываемых из параллельного региона, являются **private**.
  - Переменные объявленные внутри блока операторов параллельного региона являются приватными.
  - Счетчики циклов витки которых распределяются между нитями при помощи конструкций **for** и **parallel for**.

Классы переменных

```
double Array1[100];
int main() {
    int Array2[100];
    #pragma omp parallel
    {
        int iam = omp_get_thread_num ();
        work(Array2);
     }
     printf("%d\n", Array2[0]);
}
```

```
extern double Array1[10];
void work(int *Array) {
    double TempArray[10];
    static int count;
...
}
```



Классы переменных

Можно изменить класс переменной при помощи конструкций:

- **shared** (список переменных)
- **private** (список переменных)
- **firstprivate** (список переменных)
- **lastprivate** (список переменных)
- **threadprivate** (список переменных)
- **default** (**private** | **shared** | **none**)

Конструкция private

- Конструкция «private(var)» создает локальную копию переменной «var» в каждой из нитей.
  - Значение переменной не инициализировано
  - Приватная копия не связана с оригинальной переменной
  - В OpenMP 2.5 значение переменной «var» не определено после завершения параллельной конструкции

```
sum = -1.0;
#pragma omp parallel for private (i,j,sum)
for (i=0; i< m; i++)
{
   sum = 0.0;
   for (j=0; j< n; j++)
     sum +=b[i][j]*c[j];
   a[i] = sum;
}
printf (“sum=%f\n", sum);
```

Конструкция firstprivate

- «firstprivate» является специальным случаем «private»
  - Инициализирует каждую приватную копию соответствующим значением из главной (master) нити.

```
BOOL FirstTime=TRUE;
#pragma omp parallel for firstprivate(FirstTime)
for (row=0; row<height; row++)
{
  if (FirstTime == TRUE)  { FirstTime = FALSE;   FirstWork (row); }
 AnotherWork (row);
}
```

Конструкция lastprivate

- lastprivate передает значение приватной переменной, посчитанной на последней итерации в глобальную переменную.

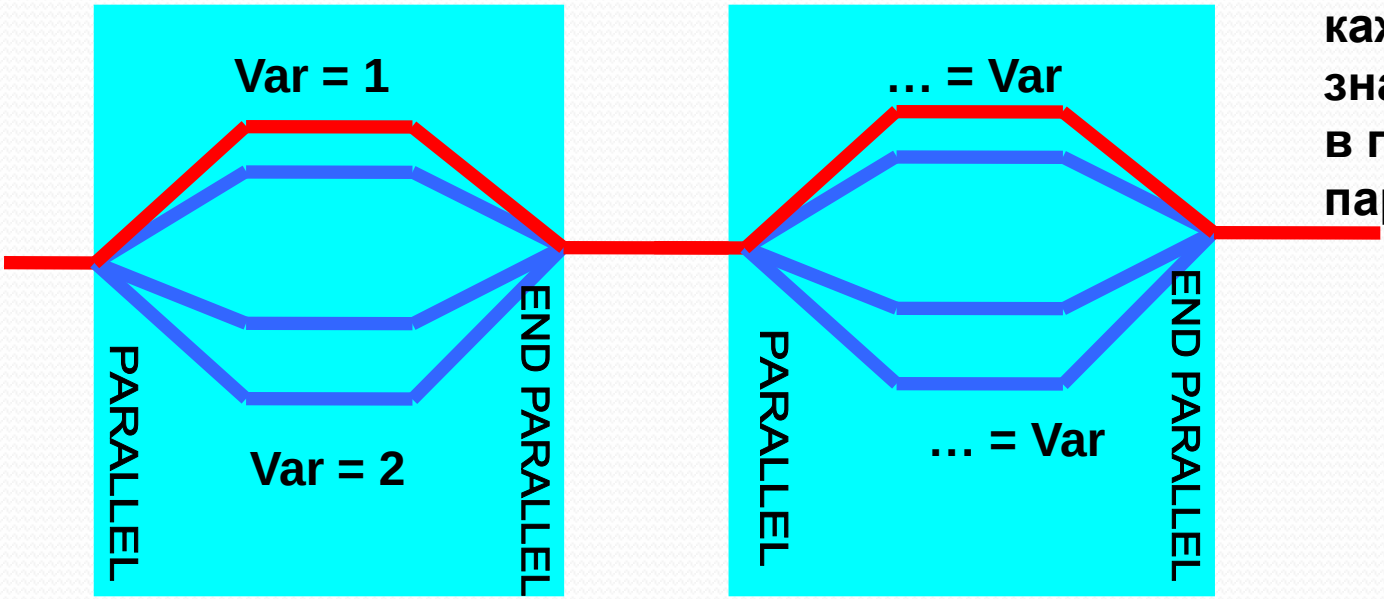
```
int i;
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];

}
a[i]=b[i];   /*i == n-1*/
```

Директива threadprivate

Отличается от применения конструкции **private**:

- **private** скрывает глобальные переменные
- **threadprivate** – переменные сохраняют глобальную область видимости внутри каждой нити

#pragma omp threadprivate (Var)



Если количество нитей не изменилось, то каждая нить получит значение, посчитанное в предыдущей параллельной области.

Конструкция default

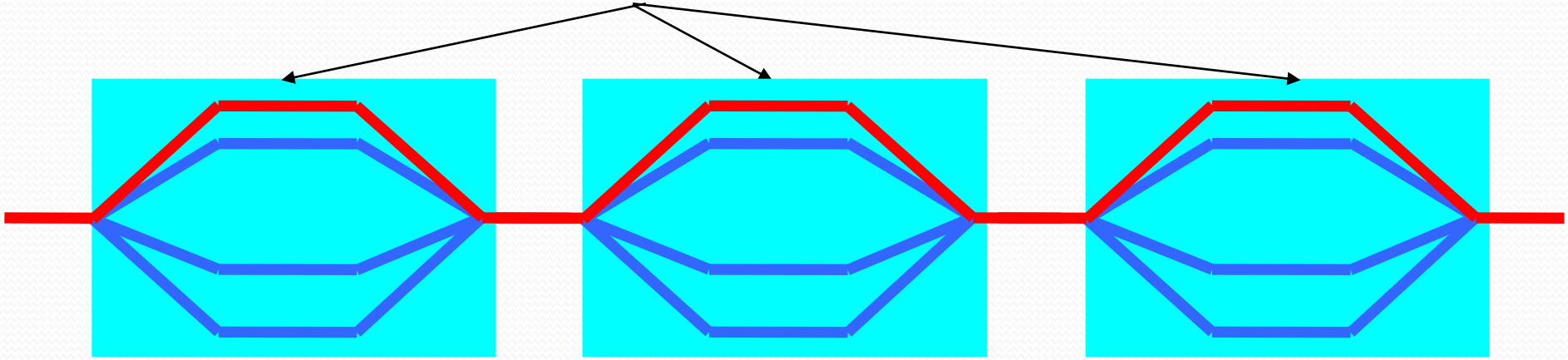
Меняет класс переменной по умолчанию:

- **default (shared)** – действует по умолчанию
- **default (private)** – есть только в Fortran
- **default (firstprivate)** – есть только в Fortran OpenMP 3.1
- **default (none)** – требует определить класс для каждой переменной

```
itotal = 100
#pragma omp parallel
private(np,each)
{
np = omp_get_num_threads()
each = itotal/np
……
}
```

```
itotal = 100
#pragma omp parallel default(none)
private(np,each) shared (itotal)
{
np = omp_get_num_threads()
each = itotal/np
……
}
```

Параллельная область (директива parallel)

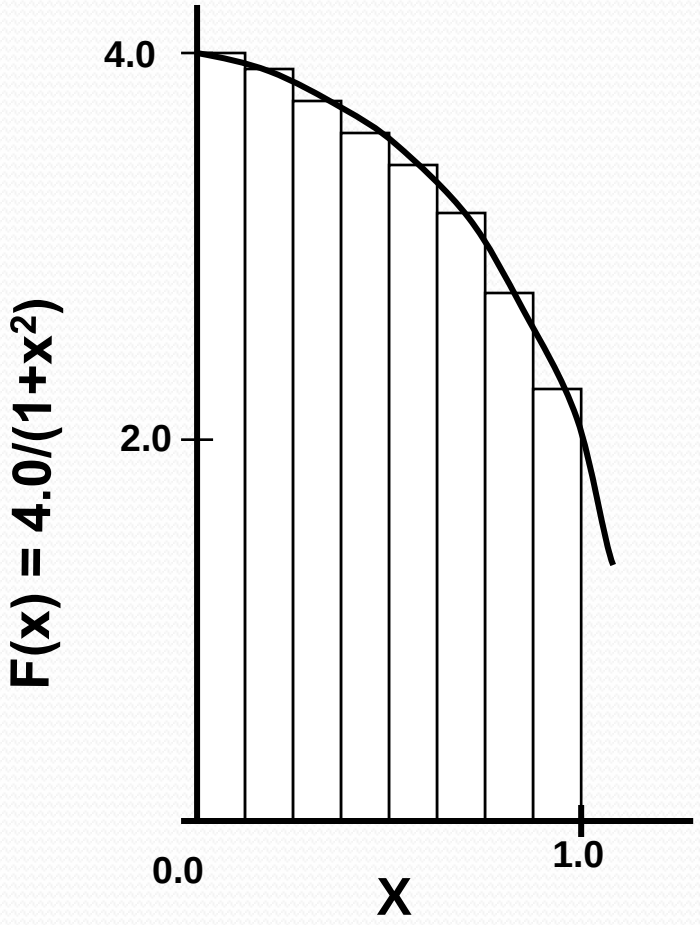


#pragma omp parallel *[клауза[[,] клауза] ...]*
структурный блок

где клауза одна из :

- **default(shared | none)**
- **private(***list***)**
- **firstprivate(***list***)**
- **shared(***list***)**
- **reduction(***operator***: *list***)**
- **if(***scalar-expression***)**
- **num_threads(***integer-expression***)**
- **copyin(***list***)**

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. Последовательная программа

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
    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. Программа с ошибкой

```
#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, 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));  // Ошибка
      }
    }
    pi = h * sum;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Конфликт доступа к данным

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

Thread0: sum = sum + val; && Thread1: sum = sum + val;

Время	Thread 0	Thread 1
1	LOAD R1,sum	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R1,sum
4		LOAD R2,val
5		ADD R1,R2
6		STORE R1,sum
7	STORE R1,sum	

Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

Вычисление числа π на OpenMP

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 32
int main ()
{
   int n =100000, i;
   double pi, h, sum[NUM_THREADS], x;
   h   = 1.0 / (double) n;
#pragma omp parallel default (none) private (i,x) shared (n,h,sum)
   {
      int id = omp_get_thread_num();
      int numt = omp_get_num_threads();
      for (i = id + 1, sum[id] = 0.0; i <= n;  i=i+numt)
      {
            x = h * ((double)i - 0.5);
            sum[id] += (4.0 / (1.0 + x*x));
      }
    }
    for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * h;
    printf("pi is approximately %.16f", pi);
    return 0;
}
```

Вычисление числа π на OpenMP. Клауза reduction

```
#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;
}
```

Клауза reduction

reduction(operator:list)

- Внутри паралельной области для каждой переменной из списка list создается копия этой переменной. Эта переменная инициализируется в соответствии с оператором operator (например, 0 для «+»).
- Для каждой нити компилятор заменяет в параллельной области обращения к редукционной переменной на обращения к созданной копии.
- По завершении выполнения параллельной области осуществляется объединение полученных результатов.

Оператор	Начальное значение
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0
max	Least number in reduction list item type
min	Largest number in reduction list item type

Клауза if

if(*scalar-expression*)

В зависимости от значения *scalar-expression* для выполнения структурного блока будет создана группа нитей или он будет выполняться одной нитью.

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int n = 0;
  printf("Enter the number of intervals: (0 quits) ");
  scanf("%d",&n);
#pragma omp parallel if (n>10)
   {
      int id = omp_get_thread_num ();
      func (n, id);
   }
   return 0;
}
```

Клауза num_threads

num_threads(*integer-expression)*

integer-expression задает максимально возможное число нитей, которые будут созданы для выполнения структурного блока

```
#include <omp.h>
int main()
{
  int n = 0;
  printf("Enter the number of intervals: (0 quits) ");
  scanf("%d",&n);
 omp_set_dynamic(1);
#pragma omp parallel num_threads(10)
   {
      int id = omp_get_thread_num ();
      func (n, id);
   }
   return 0;
}
```

Клауза copyin

copyin(list)

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

```
#include <stdlib.h>
float* work;
int size;
float tol;
#pragma omp threadprivate(work,size,tol)
void build()
{
   int i;
   work = (float*)malloc( sizeof(float)*size );
   for( i = 0; i < size; ++i ) work[i] = tol;
}

int main()
{
   read_from_file (&tol, &size);
#pragma omp parallel copyin(tol,size)
   build();
}
```

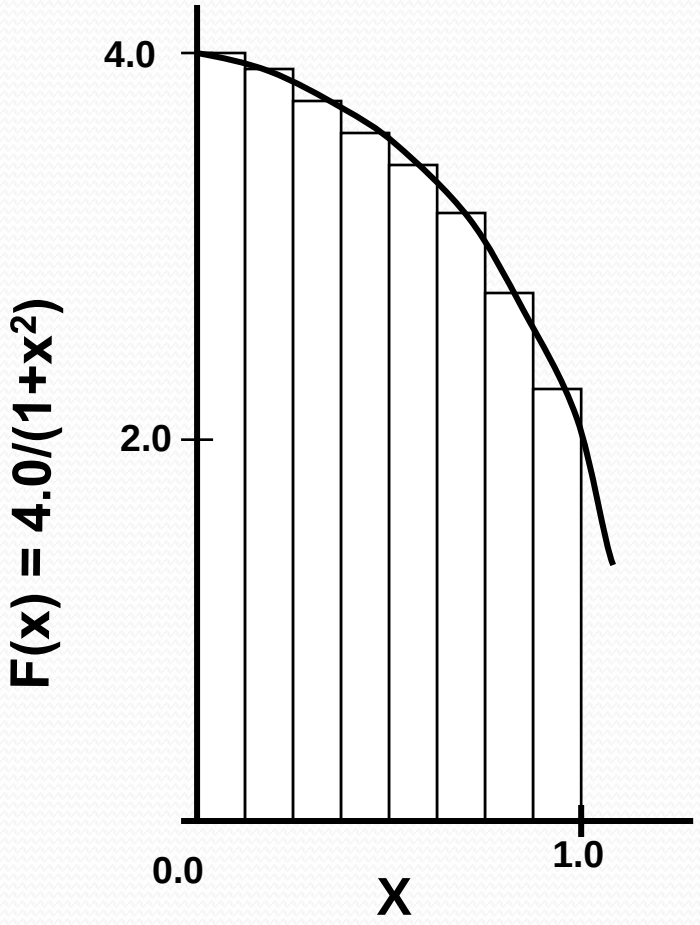
Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

Конструкции распределения работы

- Распределение витков циклов (директива for)
- Распределение нескольких структурных блоков между нитями (директива section)
- Выполнение структурного блока одной нитью (директива single)
- Задачи (директива task)

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. Последовательная программа

```
#include <stdio.h>
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
    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 =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)
   {
      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**
- **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

Вложенность конструкций распределения работы

```
void work(int i, int j) {}
void wrong1(int n)
{
#pragma omp parallel default(shared)
  {
    int i, j;
    #pragma omp for
    for (i=0; i < n; i++) {
      /* incorrect nesting of loop regions */
      #pragma omp for
        for (j=0; j < n; j++)
          work(i, j);
    }
  }
}
```

Вложенность конструкций распределения работы

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

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.

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).**

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 *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]);
   }
}
```

Если количество итераций у циклов совпадает и параметры клаузы schedule совпадают (STATIC + *число_итераций*).

Распределение витков цикла. Клауза nowait

```
void example(int n, float *a, float *b, float *z)
{
    int i;
    float sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait reduction (+: sum)
        for (i=0; i<n; i++) {
            c[i] = (a[i] + b[i]) / 2.0;
            sum += c[i];
        }
        #pragma omp for schedule(static) nowait
          for (i=0; i<n; i++)
              z[i] = sqrt(c[i]);
         #pragma omp barrier
            … = sum

    }
}
```

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>
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 ();
   }
}
```

Понятие задачи. Директива task (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.**

Задаются при помощи директивы:

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

структурный блок

где клауза одна из :

- **if (scalar-*expression*)**
- **final (scalar-*expression*)**
- **mergeable**
- **untied**
- **shared (*list*)**
- **private (*list*)**
- **firstprivate (*list*)**
- **default(shared | none)**

Использование директивы 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;
            }
          }
      }
}
```

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

Конструкции для синхронизации нитей

- Директива master
- Директива critical
- Директива atomic
- Семафоры
- Директива barrier
- Директива taskwait
- Директива flush
- Директива ordered

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);
   }
}
```

Взаимное исключение критических интервалов

При взаимодействии через общую память нити должны синхронизовать свое выполнение.

Thread0: pi = pi + val; && Thread1: pi = pi + val;

Время	Thread 0	Thread 1
1	LOAD R1,pi	
2	LOAD R2,val	
3	ADD R1,R2	LOAD R1,pi
4		LOAD R2,val
5		ADD R1,R2
6		STORE R1,pi
7	STORE R1,pi	

Результат зависит от порядка выполнения команд. Требуется взаимное исключение критических интервалов.

Взаимное исключение критических интервалов

Решение проблемы взаимного исключения должно удовлетворять требованиям:

- в любой момент времени только одна нить может находиться внутри критического интервала;
- если ни одна нить не находится в критическом интервале, то любая нить, желающая войти в критический интервал, должна получить разрешение без какой либо задержки;
- ни одна нить не должна бесконечно долго ждать разрешения на вход в критический интервал (если ни одна нить не будет находиться внутри критического интервала бесконечно).

Вычисление числа π. Последовательная программа

```
int main ()
{
    int n =100000, i;
    double pi, h, sum, x;
    h   = 1.0 / (double) n;
    sum = 0.0;
    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 с использованием критической секции

```
#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
       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;
}
```

Директива critical

```
int from_ list(float *a, int type);
void work(int i, float *a);

void example ()
{
#pragma omp parallel
    {
        float *x;
        int ix_next;
        #pragma omp critical (list0)
            ix_next = from_ list(x,0);
            work(ix_next, x);
        #pragma omp critical (list1)
            ix_next = from_ list(x,1);
            work(ix_next, x);
    }
}
```

Директива atomic

#pragma omp atomic
expression-stmt
где expression-stmt:

- **x binop= expr**
- **x++**
- **++x**
- **x--**
- **--x**

Здесь x – скалярная переменная, expr – выражение со скалярными типами, в котором не присутствует переменная x.

где binop - не перегруженный оператор:

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

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
      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;
}
```

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 на OpenMP 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
      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

```
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) {}
```

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 barrier

```
void work(int i, int j) {}
void wrong(int n)
{
#pragma omp parallel default(shared)
{
      int i;
      #pragma omp for
      for (i=0; i<n; i++) {
          work(i, 0);
          /* incorrect nesting of barrier region in a loop region */
          #pragma omp barrier
                work(i, 1);
           }
}
}
```

Директива barrier

```
void work(int i, int j) {}
void wrong(int n)
{
#pragma omp parallel default(shared)
{
      int i;
      #pragma omp critical
      {
          work(i, 0);
          /* incorrect nesting of barrier region in a critical region */
          #pragma omp barrier
                work(i, 1);
           }
}
}
```

Директива barrier

```
void work(int i, int j) {}
void wrong(int n)
{
#pragma omp parallel default(shared)
{
      int i;
      #pragma omp single
      {
          work(i, 0);
          /* incorrect nesting of barrier region in a single region */
          #pragma omp barrier
                work(i, 1);
           }
}
}
```

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 flush

#pragma omp flush *[(список переменных)]*

По умолчанию все переменные приводятся в консистентное состояние **(#pragma omp flush**):

- При барьерной синхронизации
- При входе и выходе из конструкций **parallel**, **critical** и **ordered**.
- При выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**), если не указана клауза **nowait**.
- При вызове **omp_set_lock** и **omp_unset_lock**.
- При вызове **omp_test_lock**, **omp_set_nest_lock**, **omp_unset_nest_lock**
- и **omp_test_nest_lock**, если изменилось состояние семафора.

При входе и выходе из конструкции **atomic** выполняется **#pragma omp flush(x)**, где x – переменная, изменяемая в конструкции **atomic**.

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

Система поддержки выполнения OpenMP-программ

- Внутренние переменные, управляющие выполнением OpenMP-программы (ICV-Internal Control Variables).
- Задание/опрос значений ICV-переменных.
- Функции работы со временем.

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*

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 всей программы.

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. run-sched-var

```
void work(int i);

int main () {
   omp_sched_t schedules [] = {omp_sched_static, omp_sched_dynamic,
omp_sched_guided, omp_sched_auto};
   omp_set_num_threads (4);
   #pragma omp parallel
   {
      omp_set_schedule (schedules[omp_get_thread_num()],0);
      #pragma omp parallel for schedule(runtime)
         for (int i=0;i<N;i++)  work (i);
   }
}
```

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

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. 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

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

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

Наиболее часто встречаемые ошибки в OpenMP-программах

- Трудно обнаруживаемые ошибки типа race condition (конфликт доступа к данным).
- Ошибки типа deadlock (взаимная блокировка нитей).
- Использование неинициализированных переменных.
- Автоматизированный поиск ошибок в OpenMP-программах при помощи Intel Thread Checker и Sun Studio Thread Analyzer.

Конфликт доступа к данным

Ошибка возникает при одновременном выполнении следующих условий:

1. Две или более нитей обращаются к одной и той же ячейке памяти.
2. По крайней мере, один из этих доступов к памяти является записью.
3. Нити не синхронизируют свой доступ к данной ячейки памяти.

При одновременном выполнении всех трех условий порядок доступа становится неопределенным.

Конфликт доступа к данным

Использование различных компиляторов (различных опций оптимизации, включение/отключение режима отладки при компиляции программы), применение различных стратегий планирования выполнения нитей в различных ОС, может приводить к тому, что в каких-то условиях (например, на одной вычислительной машине) ошибка не будет проявляться, а в других (на другой машине) – приводить к некорректной работе программы.

От запуска к запуску программа может выдавать различные результаты в зависимости от порядка доступа.

Отловить такую ошибку очень тяжело.

Причиной таких ошибок как правило являются:

- **неверное определение класса переменной,**
- **отсутствие синхронизации.**

Конфликт доступа к данным

```
#define N
float a[N], tmp
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<N;i++) {
        tmp= a[i]*a[i];
        a[i]=1-tmp;
    }
}
```

```
#define N
float a[N], sum
#pragma omp parallel
{
 #pragma omp for private(tmp)
    for(int i=0; i<N;i++) {
        tmp= a[i]*a[i];
        a[i]=1-tmp;
    }
}
```

Конфликт доступа к данным

file1.c

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
counter++;
return(counter);
}
```

file2.c

```
extern int counter;
// нет директивы threadprivate
int decrement_counter()
{
counter--;
return(counter);
}
```

Конфликт доступа к данным

```
#define N
float a[N], sum
#pragma omp parallel
{
    #pragma omp master
        sum = 0.0;
    #pragma omp for
     for(int i=0; i<N;i++) {
        #omp atomic
            sum += a[i];
     }
}
```

```
#define N
float a[N], sum
#pragma omp parallel
{
    #pragma omp master
        sum = 0.0;
    #pragma omp barrier
    #pragma omp for
     for(int i=0; i<N;i++) {
        #omp atomic
            sum += a[i];
     }
}
```

Взаимная блокировка нитей

```
#pragma omp parallel
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
        omp_set_lock (&lcka);
        omp_set_lock (&lckb);
        x = x + 1;
        omp_unset_lock (&lckb);
        omp_unset_lock (&lcka);
      } else {
        omp_set_lock (&lckb);
        omp_set_lock (&lcka);
         x = x + 2;
         omp_unset_lock (&lcka);
         omp_unset_lock (&lckb);
       }
   }
}
```

Взаимная блокировка нитей

```
#pragma omp parallel
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
        omp_set_lock (&lcka);
        while (x<0); /*цикл ожидания*/
        omp_unset_lock (&lcka);
     } else {
        omp_set_lock (&lcka);
        x++;
        omp_unset_lock (&lcka);
     }
  }
}
```

Взаимная блокировка нитей

```
#define N 10
int A[N],B[N], sum;
#pragma omp parallel default(shared) num_threads(10)
{
    int iam=omp_get_thread_num();
    if (iam ==0) {
      #pragma omp critical (update_a)
        #pragma omp critical (update_b)
          sum +=A[iam];
    } else {
      #pragma omp critical (update_b)
        #pragma omp critical (update_a)
          sum +=B[iam];
    }
}
```

Автоматизированный поиск ошибок. Intel Thread Checker. Intel Inspector XE 2011

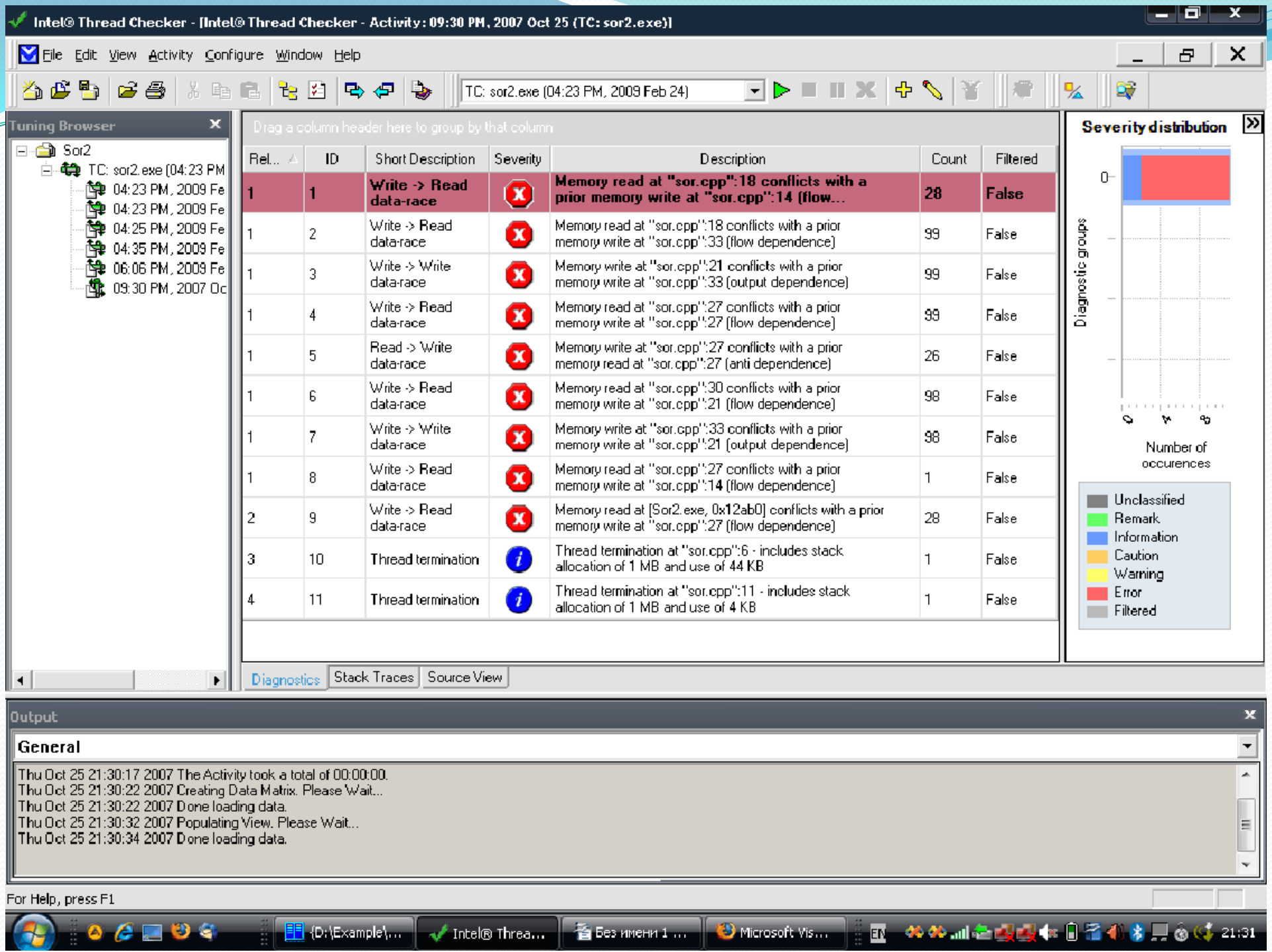
KAI Assure Threads (Kuck and Associates)

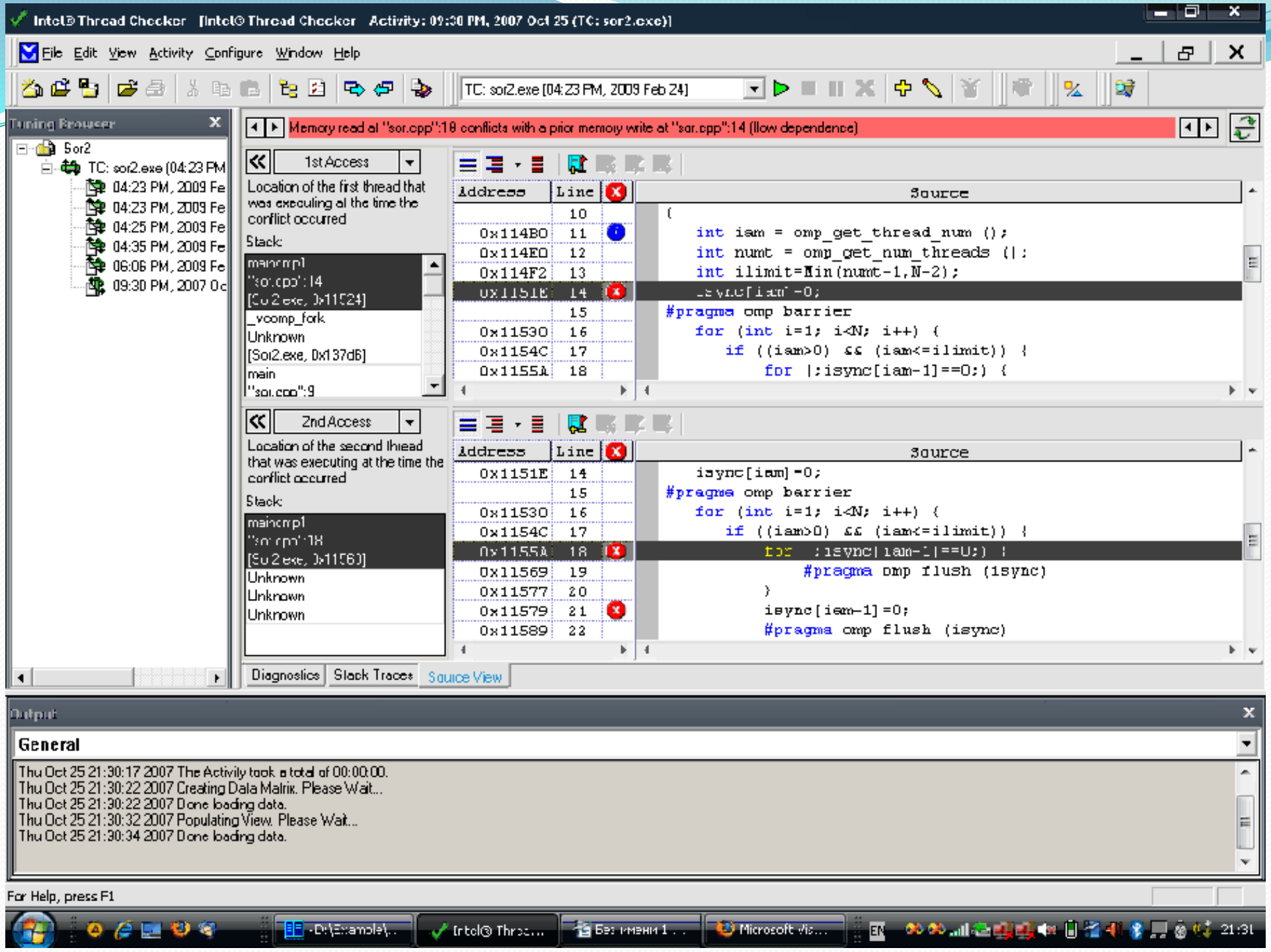
Анализ программы основан на процедуре инструментации.

***Инструментация* – вставка обращений для записи действий, потенциально способных привести к ошибкам: работа с памятью, вызовы операций синхронизации и работа с потоками.**

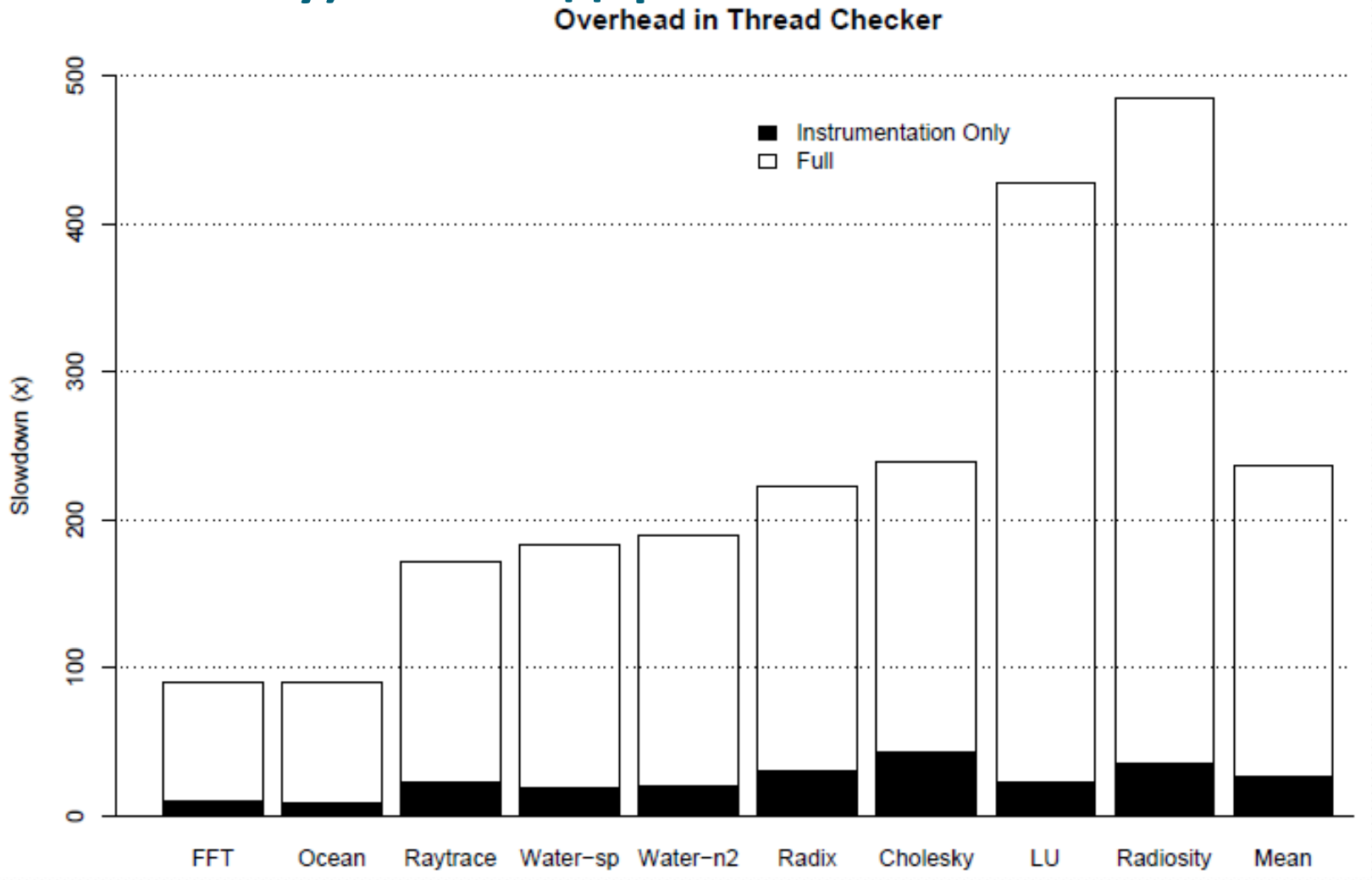
Может выполняться автоматически (бинарная инструментация) на уровне исполняемого модуля (а также dll-библиотеки)

и/или по указанию программиста на уровне исходного кода (компиляторная инструментация - Windows: /Qtcheck Linux:-tcheck).


Intel® Thread Checker - [Intel® Thread Checker - Activity: 09:30 PM, 2007 Oct 25 (TC: sor2.exe)]
File Edit View Activity Configure Window Help
TC: sor2.exe (04:23 PM, 2009 Feb 24)
Tuning Browser
Sor2
TC: sor2.exe (04:23 PM
04:23 PM, 2009 Fe
04:23 PM, 2009 Fe
04:25 PM, 2009 Fe
04:35 PM, 2009 Fe
06:06 PM, 2009 Fe
09:30 PM, 2007 Oc
Drag a column header here to group by that column
Rel... ID Short Description Severity Description Count Filtered
1 1 Write -> Read data-race Memory read at "sor.cpp":18 conflicts with a prior memory write at "sor.cpp":14 (flow... 28 False
1 2 Write -> Read data-race Memory read at "sor.cpp":18 conflicts with a prior memory write at "sor.cpp":33 (flow dependence) 99 False
1 3 Write -> Write data-race Memory write at "sor.cpp":21 conflicts with a prior memory write at "sor.cpp":33 (output dependence) 99 False
1 4 Write -> Read data-race Memory read at "sor.cpp":27 conflicts with a prior memory write at "sor.cpp":27 (flow dependence) 99 False
1 5 Read -> Write data-race Memory write at "sor.cpp":27 conflicts with a prior memory read at "sor.cpp":27 (anti dependence) 26 False
1 6 Write -> Read data-race Memory read at "sor.cpp":30 conflicts with a prior memory write at "sor.cpp":21 (flow dependence) 98 False
1 7 Write -> Write data-race Memory write at "sor.cpp":33 conflicts with a prior memory write at "sor.cpp":21 (output dependence) 98 False
1 8 Write -> Read data-race Memory read at "sor.cpp":27 conflicts with a prior memory write at "sor.cpp":14 (flow dependence) 1 False
2 9 Write -> Read data-race Memory read at [Sor2.exe, 0x12ab0] conflicts with a prior memory write at "sor.cpp":27 (flow dependence) 28 False
3 10 Thread termination Thread termination at "sor.cpp":6 - includes stack allocation of 1 MB and use of 44 KB 1 False
4 11 Thread termination Thread termination at "sor.cpp":11 - includes stack allocation of 1 MB and use of 4 KB 1 False
Diagnostics
Stack Traces
Source View
Severity distribution
0
Diagnostics
Number of occurences
Unclassified
Remark
Information
Caution
Warning
Error
Filtered
Output
General
Thu Oct 25 21:30:17 2007 The Activity took a total of 00:00:00.
Thu Oct 25 21:30:22 2007 Creating Data Matrix. Please Wait...
Thu Oct 25 21:30:22 2007 Done loading data.
Thu Oct 25 21:30:32 2007 Populating View. Please Wait...
Thu Oct 25 21:30:34 2007 Done loading data.
For Help, press F1
{D:\Example\...
Intel® Threa...
Без имени 1 ...
Microsoft Vis...
EN
21:31


Intel® Thread Checker [Intel® Thread Checker Activity: 09:30 PM, 2007 Oct 25 (TC: sor2.exe)]
File Edit View Activity Configure Window Help
TC: sor2.exe [04:23 PM, 2009 Feb 24]
Tuning Browser
Sor2
TC: sor2.exe (04:23 PM
04:23 PM, 2009 Fe
04:23 PM, 2009 Fe
04:25 PM, 2009 Fe
04:35 PM, 2009 Fe
06:06 PM, 2009 Fe
09:30 PM, 2007 Oc
Memory read at "sor.cpp":18 conflicts with a prior memory write at "sor.cpp":14 (flow dependence)
1st Access
Location of the first thread that was executing at the time the conflict occurred
Stack:
main
"sor.cpp":14
[Sor2.exe, 0x11524]
_vcomp_fork
Unknown
[Sor2.exe, 0x137d6]
main
"sor.cpp":9
Address
Line
Source
10
{
0x114B0 11 int iam = omp_get_thread_num ();
0x114E0 12 int numt = omp_get_num_threads ();
0x114F2 13 int ilimit=Min(numt-1,N-2);
0x1151E 14 isync[iam]=0;
15 #pragma omp barrier
0x11530 16 for (int i=1; i<N; i++) {
0x1154C 17 if ((iam>0) && (iam<=ilimit)) {
0x1155A 18 for (;isync[iam-1]==0;) {
2nd Access
Location of the second thread that was executing at the time the conflict occurred
Stack:
main
"sor.cpp":18
[Sor2.exe, 0x11560]
Unknown
Unknown
Unknown
0x1151E 14 isync[iam]=0;
15 #pragma omp barrier
0x11530 16 for (int i=1; i<N; i++) {
0x1154C 17 if ((iam>0) && (iam<=ilimit)) {
0x1155A 18 for (;isync[iam-1]==0;) {
0x11569 19 #pragma omp flush (isync)
0x11577 20 }
0x11579 21 isync[iam-1]=0;
0x11589 22 #pragma omp flush (isync)
Diagnostics
Stack Traces
Source View
Output
General
Thu Oct 25 21:30:17 2007 The Activity took a total of 00:00:00.
Thu Oct 25 21:30:22 2007 Creating Data Matrix. Please Wait...
Thu Oct 25 21:30:22 2007 Done loading data.
Thu Oct 25 21:30:32 2007 Populating View. Please Wait...
Thu Oct 25 21:30:34 2007 Done loading data.
For Help, press F1

Пакет тестов SPLASH-2 (Stanford Parallel Applications for Shared Memory) на 4-х ядерной машине



http://iacoma.cs.uiuc.edu/iacoma-papers/asid06.pdf

Автоматизированный поиск ошибок. Sun Studio Thread Analyzer (Oracle Solaris Studio)

Инструментация программы:

cc -xinstrument=datarace -g -xopenmp=noopt test.c

Накопление информации о программе:

export OMP_NUM_THREADS=2

collect -r race ./a.out

collect -r deadlock ./a.out

collect -r all ./a.out

Получение результатов анализа программы

tha test.1.er => GUI

er_print test.1.er => интерфейс командной строки

Автоматизированный поиск ошибок. Sun Studio Thread Analyzer

```
if (iam==0) {
data = …
user_lock ();
} else {
user_lock ();
    … = data;
}
```

```
if (iam==0) {
ptr1 = mymalloc(sizeof(data_t));
ptr1->data = ...
...
myfree(ptr1);
} else {
ptr2 = mymalloc(sizeof(data_t));
ptr2->data = ...
...
myfree(ptr2);
}
```

Может выдавать сообщения об ошибках там где их нет

Intel Thread Checker и Sun Studio Thread Analyzer

Программа	Jacobian Solver		Sparse Matrix-Vector multiplication		Adaptive Integration Solver	
	Mem	MFLOP/s	Mem	MFLOP/s	Mem	Time
Intel on 2 Threads	5	621	40	929	4	5.0 s
Intel Thread Checker on 2 Threads	115	0.9	1832	3.5	30	9.5 s
Intel Thread Checker –tcheck	115	3.1	-	-	-	-
Sun on 2 Threads	5	600	50	550	2	8.4 s
Sun Thread Analyzer on 2 Threads	125	1.1	2020	0.8	17	8.5 s

http://www.fz-juelich.de/nic-series/volume38/terboven.pdf

Содержание

- Современные направления развития параллельных вычислительных систем
- OpenMP – модель параллелизма по управлению
- Конструкции распределения работы
- Конструкции для синхронизации нитей
- Система поддержки выполнения OpenMP-программ. Переменные окружения, управляющие выполнением OpenMP-программы
- Наиболее часто встречаемые ошибки в OpenMP-программах. Функциональная отладка OpenMP-программ
- Отладка эффективности OpenMP-программ

Отладка эффективности OpenMP-программ

- Основные характеристики производительности
- Стратегии распределения витков цикла между нитями (клауза schedule)
- Отмена барьерной синхронизации по окончании выполнения цикла (клауза nowait)
- Локализация данных
- Задание поведения нитей во время ожидания (переменная OMP_WAIT_POLICY)
- Оптимизация OpenMP-программы при помощи Intel Thread Profiler

Основные характеристики производительности

Полезное время - время, которое потребуется для выполнения программы на однопроцессорной ЭВМ.

Общее время использования процессоров равно произведению **времени выполнения** программы на многопроцессорной ЭВМ (максимальное значение среди времен выполнения программы на всех используемых ею процессорах) на **число используемых процессоров.**

Главная характеристика эффективности параллельного выполнения - **коэффициент эффективности** равен отношению полезного времени к общему времени использования процессоров.

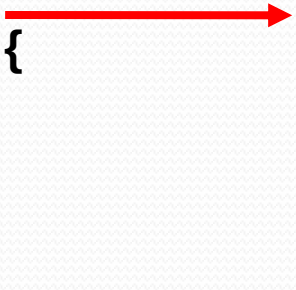
Разница между общим временем использования процессоров и полезным временем представляет собой **потерянное время.**

Существуют следующие составляющие **потерянного времени**:

- накладные расходы на создание группы нитей;
- потери из-за простоев тех нитей, на которых выполнение программы завершилось раньше, чем на остальных (**несбалансированная нагрузка нитей**);
- потери из-за синхронизации нитей (например, из-за чрезмерного использования общих данных);
- потери из-за недостатка параллелизма, приводящего к дублированию вычислений на нескольких процессорах (**недостаточный параллелизм**).

Накладные расходы на создание группы нитей

```
void work(int i, int j) {}

 for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        work(i, j);
    }
}
```



```
for (int i=0; i < n; i++) {
 #pragma omp parallel for
    for (int j=0; j < n; j++) {
        work(i, j);
    }
}
```

```
 #pragma omp parallel for
 for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        work(i, j);
    }
}
```

Алгоритм Якоби

```
for (int it=0;it<ITMAX; it++) {
   for (int i=1; i<N-1; i++ )
      for( int j=1; j<N-1; j++ )
         tmp[i][j] = 0.25 * ( grid[i-1][j] +
            grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
   for (int i=0; i<N; i++ )
      for( int j=0; j<N; j++ )
         grid[i][j] = tmp[i][j];
}
```

```
for (int it=0;it<ITMAX; it++) {
   #pragma omp parallel for
   for (int i=1; i<N-1; i++ )
      for (int j=1; j<N-1; j++ )
         tmp[i][j] = 0.25 * ( grid[i-1][j] +
            grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
   #pragma omp parallel for
   for (int i=1; i<N-1; i++ )
      for (int j=1; j<N-1; j++ )
         grid[i][j] = tmp[i][j];
}
```

```
#pragma omp parallel
{
   for (int it=0;it<ITMAX; it++) {
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ )
            tmp[i][j] = 0.25 * ( grid[i-1][j] +
               grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ )
            grid[i][j] = tmp[i][j];
   }
}
```

Балансировка нагрузки нитей. Клауза schedule

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

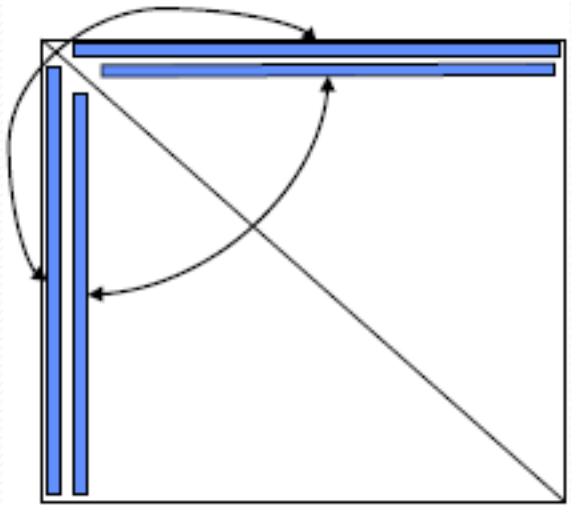
Где *алгоритм планирования* один из:

- **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 schedule(static)
  for(int i = 1; i <= 100; i++)
     A[i]=0.;
```

Балансировка нагрузки нитей. Клауза schedule

```
#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;
      }
```



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

Отмена барьерной синхронизации по окончании выполнения цикла. Клауза nowait

```
void example(int n, float *a, float *b, float *z, int n)
{
    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]);
    }
}
```

Локализация данных

```
#pragma omp parallel shared (var)
{
    <критическая секция>
    {
        var = …
    }
}
```

Модификация общей переменной в параллельной области должна осуществляться в критической секции (critical/atomic/omp_set_lock).

Если локализовать данную переменную (например, private(var)), то можно сократить потери на синхронизацию нитей.

Переменная OMP_WAIT_POLICY

Подсказка OpenMP-компилятору о желаемом поведении нитей во время ожидания.

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

Sun Studio
setenv SUNW_MP_THR_IDLE SPIN
setenv SUNW_MP_THR_IDLE SLEEP
setenv SUNW_MP_THR_IDLE SLEEP(2s)
setenv SUNW_MP_THR_IDLE SLEEP(20ms)
setenv SUNW_MP_THR_IDLE SLEEP(150mc)

Оптимизация для DSM-систем

```
#pragma omp parallel for
   for (int i=1; i<N-1; i++ )
      for (int j=1; j<N-1; j++ ) {      // first-touch algoritm
          tmp[i][j] = 0.;
          grid[i][j] = 1. + i + j;
      }
for (int it=0;it<ITMAX; it++) {
   #pragma omp parallel for
   for (int i=1; i<N-1; i++ )
      for (int j=1; j<N-1; j++ )
         tmp[i][j] = 0.25 * ( grid[i-1][j] +
            grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
   #pragma omp parallel for
   for (int i=1; i<N-1; i++ )
      for (int j=1; j<N-1; j++ )
         grid[i][j] = tmp[i][j];
}
```

Intel Thread Profiler. Intel Vtune Amplifier XE 2011

Предназначен для анализа производительности OpenMP-приложений или многопоточных приложений с использованием потоков Win32 API и POSIX.

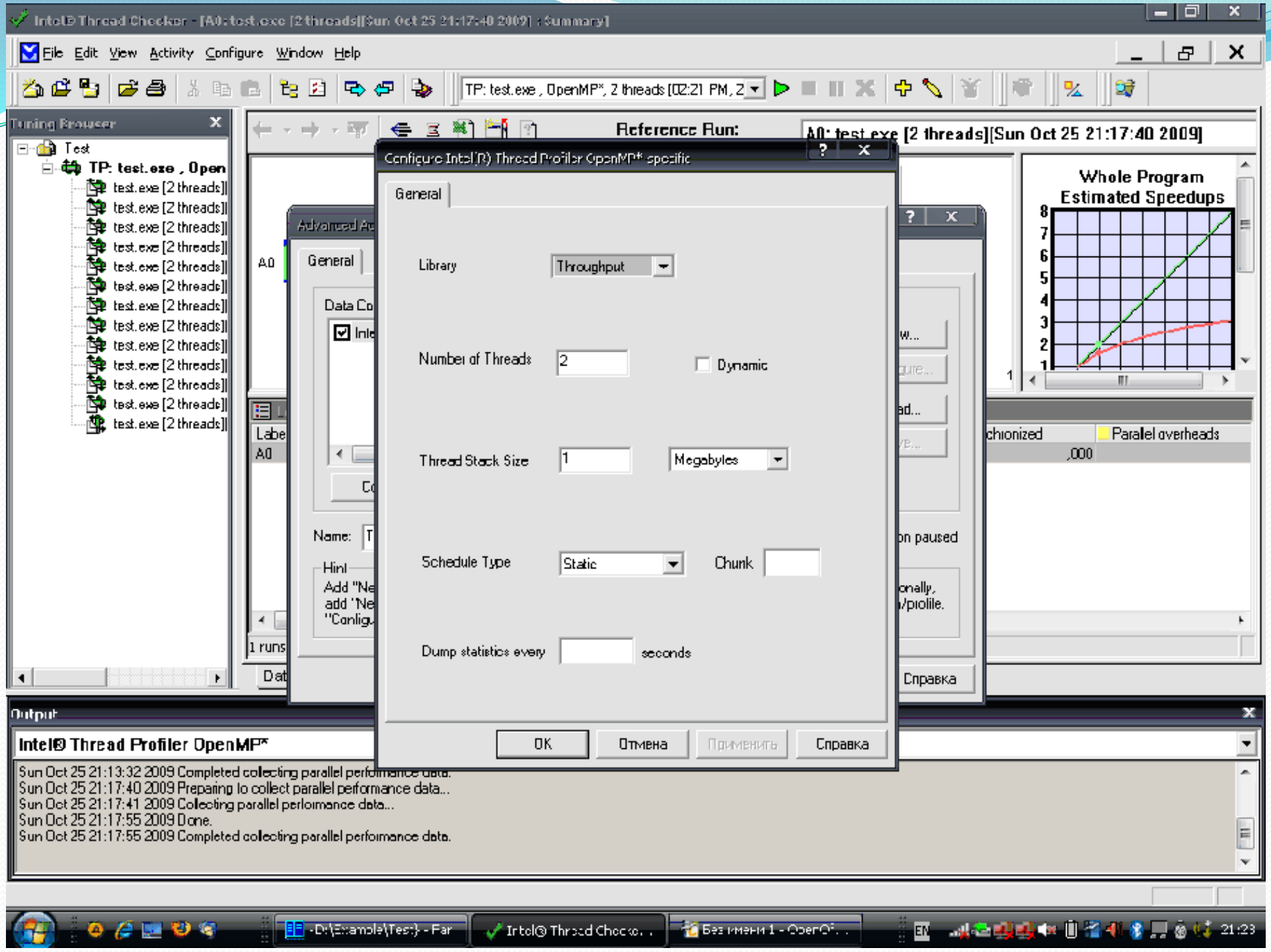
Визуализация выполнения потоков во времени помогает понять их функции и взаимодействие.

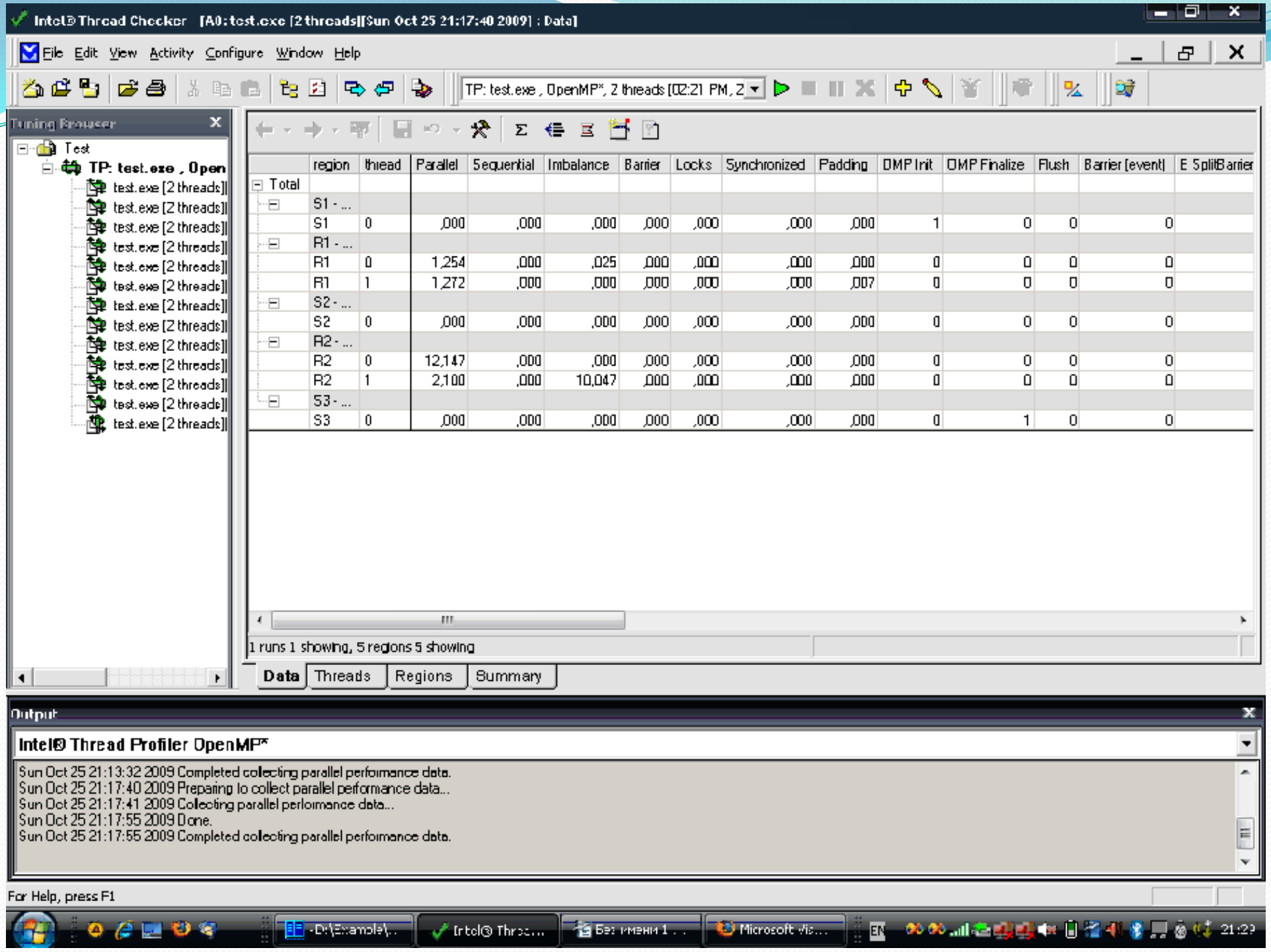
Инструмент указывает на узкие места, снижающие производительность.

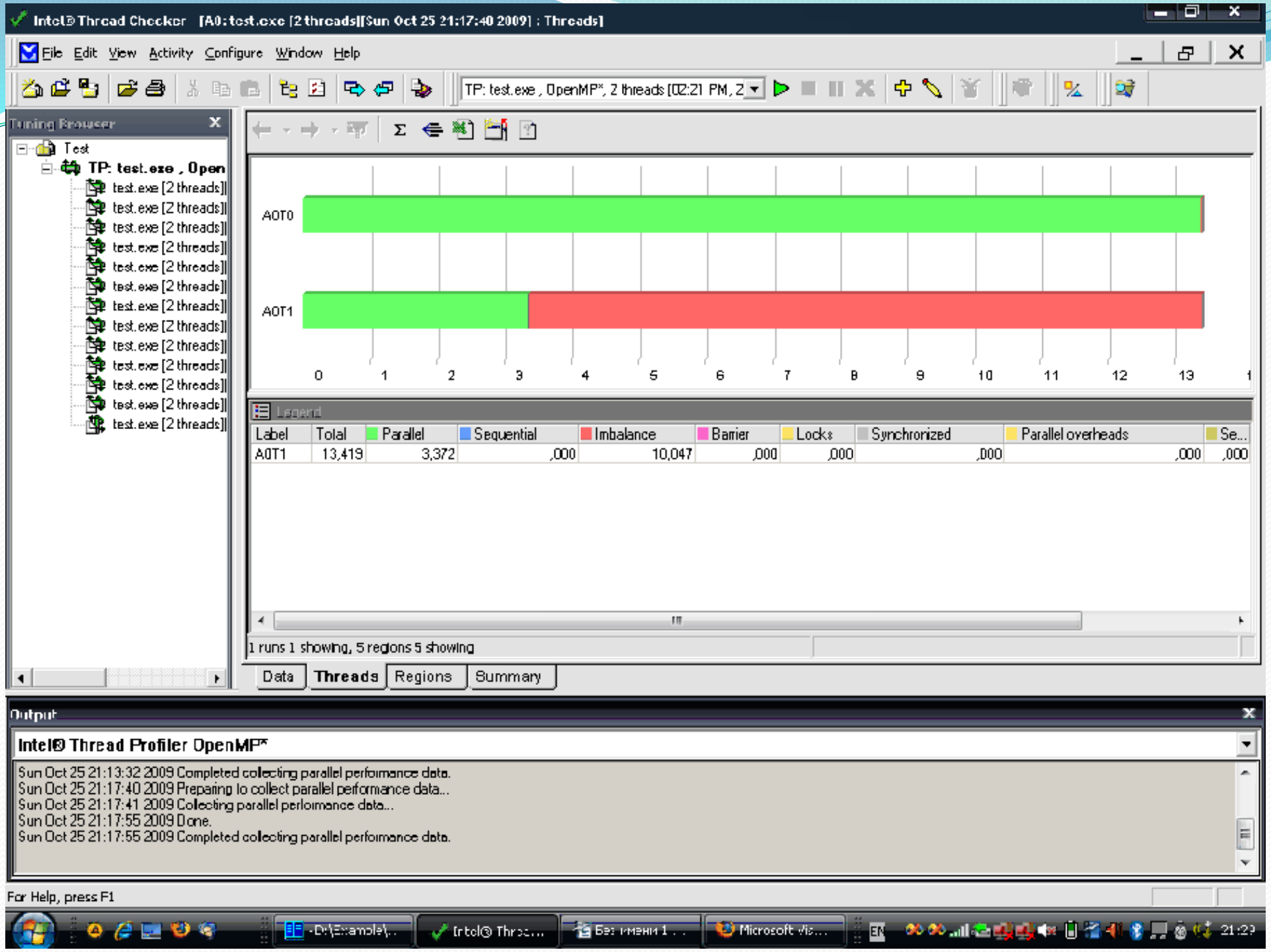
Инструментация программы:

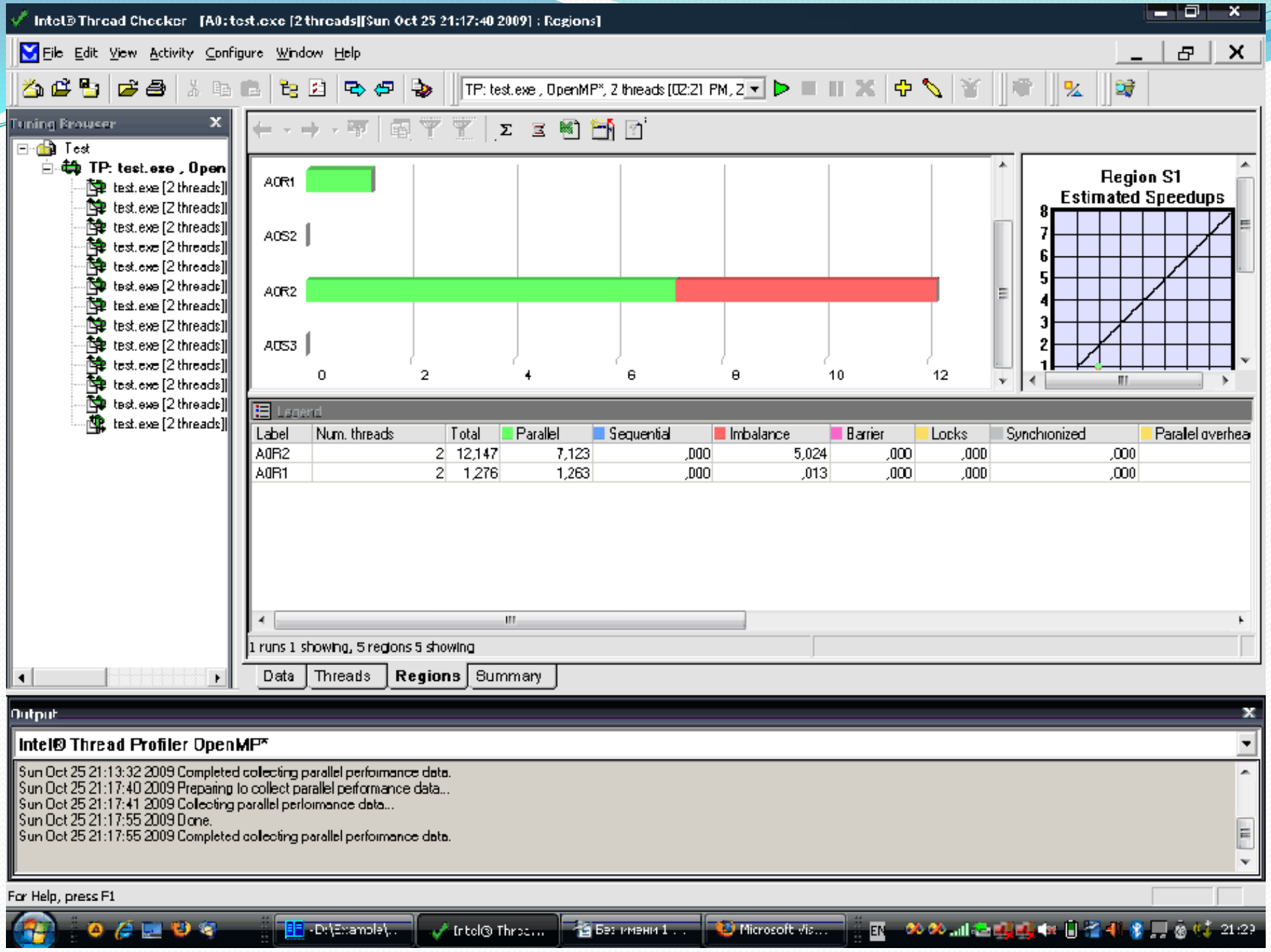
Linux: -g [-openmp_profile]

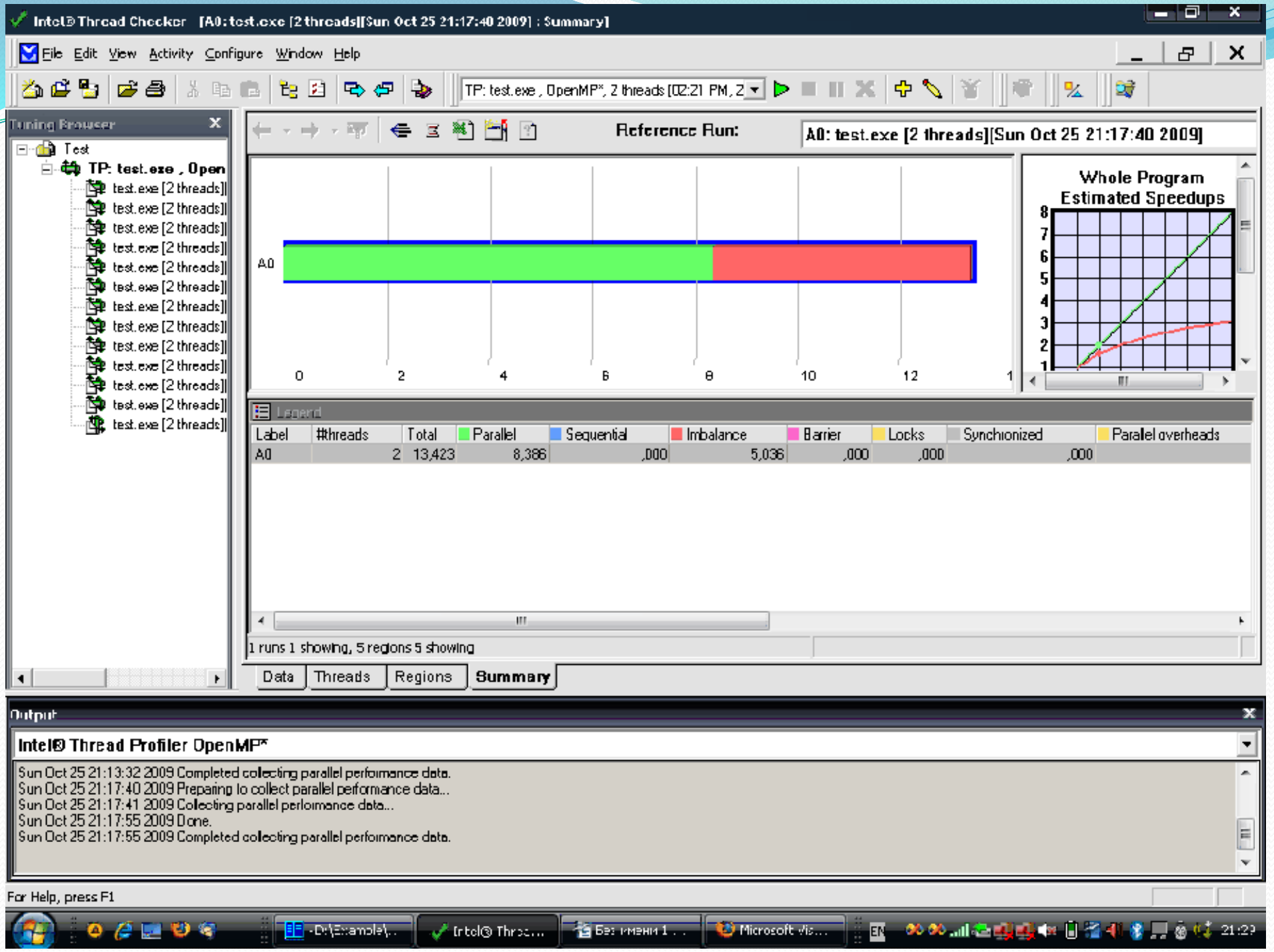
Windows: /Zi [/-Qopenmp_profile], link with /fixed:no


Intel® Thread Checker - [A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Summary]
File Edit View Activity Configure Window Help
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
Reference Run:
A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009]
Whole Program
Estimated Speedups
Configure Intel(R) Thread Profiler OpenMP* specific
General
Library
Throughput
Number of Threads
2
Dynamic
Thread Stack Size
1
Megabytes
Schedule Type
Static
Chunk
Dump statistics every
seconds
OK
Отмена
Применить
Справка
Output
Intel® Thread Profiler OpenMP*
Sun Oct 25 21:13:32 2009 Completed collecting parallel performance data.
Sun Oct 25 21:17:40 2009 Preparing to collect parallel performance data...
Sun Oct 25 21:17:41 2009 Collecting parallel performance data...
Sun Oct 25 21:17:55 2009 Done.
Sun Oct 25 21:17:55 2009 Completed collecting parallel performance data.
Parallel overheads
21:23


Intel® Thread Checker [A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Data]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
region thread Parallel Sequential Imbalance Barrier Locks Synchronized Padding OMP Init OMP Finalize Flush Barrier (event) E SplitBarrier
Total
S1 - ...
S1 0 ,000 ,000 ,000 ,000 ,000 ,000 ,000 1 0 0 0
R1 - ...
R1 0 1,254 ,000 ,025 ,000 ,000 ,000 ,000 0 0 0 0
R1 1 1,272 ,000 ,000 ,000 ,000 ,000 ,007 0 0 0 0
S2 - ...
S2 0 ,000 ,000 ,000 ,000 ,000 ,000 ,000 0 0 0 0
R2 - ...
R2 0 12,147 ,000 ,000 ,000 ,000 ,000 ,000 0 0 0 0
R2 1 2,100 ,000 10,047 ,000 ,000 ,000 ,000 0 0 0 0
S3 - ...
S3 0 ,000 ,000 ,000 ,000 ,000 ,000 ,000 0 1 0 0
1 runs 1 showing, 5 regions 5 showing
Data Threads Regions Summary
Output
Intel® Thread Profiler OpenMP*
Sun Oct 25 21:13:32 2009 Completed collecting parallel performance data.
Sun Oct 25 21:17:40 2009 Preparing to collect parallel performance data...
Sun Oct 25 21:17:41 2009 Collecting parallel performance data...
Sun Oct 25 21:17:55 2009 Done.
Sun Oct 25 21:17:55 2009 Completed collecting parallel performance data.
For Help, press F1
D:\Example\...
Intel® Thread...
Microsoft Vis...
EN
21:22


Intel® Thread Checker [A0:test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Threads]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
A0T0
A0T1
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Legend
Label Total Parallel Sequential Imbalance Barrier Locks Synchronized Parallel overheads Se...
A0T1 13,419 3,372 ,000 10,047 ,000 ,000 ,000 ,000 ,000
1 runs 1 showing, 5 regions 5 showing
Data Threads Regions Summary
Output
Intel® Thread Profiler OpenMP*
Sun Oct 25 21:13:32 2009 Completed collecting parallel performance data.
Sun Oct 25 21:17:40 2009 Preparing to collect parallel performance data...
Sun Oct 25 21:17:41 2009 Collecting parallel performance data...
Sun Oct 25 21:17:55 2009 Done.
Sun Oct 25 21:17:55 2009 Completed collecting parallel performance data.
For Help, press F1


Intel® Thread Checker [A0:test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Regions]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
A0R1
A0S2
A0R2
A0S3
0 2 4 6 8 10 12
Region S1
Estimated Speedups
8 7 6 5 4 3 2 1
Legend
Label Num. threads Total Parallel Sequential Imbalance Barrier Locks Synchronized Parallel overhea
A0R2 2 12,147 7,123 ,000 5,024 ,000 ,000 ,000
A0R1 2 1,276 1,263 ,000 ,013 ,000 ,000 ,000
1 runs 1 showing, 5 regions 5 showing
Data Threads Regions Summary
Output
Intel® Thread Profiler OpenMP*
Sun Oct 25 21:13:32 2009 Completed collecting parallel performance data.
Sun Oct 25 21:17:40 2009 Preparing to collect parallel performance data...
Sun Oct 25 21:17:41 2009 Collecting parallel performance data...
Sun Oct 25 21:17:55 2009 Done.
Sun Oct 25 21:17:55 2009 Completed collecting parallel performance data.
For Help, press F1
D:\Example\...
Intel® Thread...
Microsoft Vis...
EN
21:22


Intel® Thread Checker [A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Summary]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
Reference Run:
A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009]
A0
0 2 4 6 8 10 12
Whole Program Estimated Speedups
8 7 6 5 4 3 2 1
Legend
Label #threads Total Parallel Sequential Imbalance Barrier Locks Synchronized Parallel overheads
A0 2 13,423 8,386 ,000 5,036 ,000 ,000 ,000
1 runs 1 showing, 5 regions 5 showing
Data Threads Regions Summary
Output
Intel® Thread Profiler OpenMP*
Sun Oct 25 21:13:32 2009 Completed collecting parallel performance data.
Sun Oct 25 21:17:40 2009 Preparing to collect parallel performance data...
Sun Oct 25 21:17:41 2009 Collecting parallel performance data...
Sun Oct 25 21:17:55 2009 Done.
Sun Oct 25 21:17:55 2009 Completed collecting parallel performance data.
For Help, press F1

Литература

- **OpenMP Application Program Interface Version 3.1, July 2011.** http://ww.openmp.org/mp-documents/OpenMP3.1.pdf
- **Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009.** http://parallel.ru/info/parallel/openmp/OpenMP.pdf
- **Э. Таненбаум, М. ван Стеен.** Распределенные системы. Принципы и парадигмы. – СПб. Питер, 2003
- **Воеводин В.В., Воеводин Вл.В.** Параллельные вычисления. – СПб.: БХВ-Петербург, 2002.
- **Презентация** ftp://ftp.keldysh.ru/K_student/MSU2011/MSU2011_MPI_OpenMP2.pdf

Автор

Бахтин Владимир Александрович, кандидат физико-математических наук, заведующий сектором Института прикладной математики им. М.В. Келдыша РАН, ассистент кафедры системного программирования факультета вычислительной математики и кибернетики Московского университета им. М.В. Ломоносова
bakhtin@keldysh.ru