Отладка эффективности OpenMP-программ.

Параллельное программирование с OpenMP

Бахтин Владимир Александрович
К.ф.-м.н., ведущий научный сотрудник Института прикладной математики им М.В.Келдыша РАН
Доцент кафедры системного программирования факультета ВМК МГУ им М.В. Ломоносова

Содержание

- Основные характеристики производительности
- Оптимизация последовательной программы
- 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 (Intel Vtune Amplifier)

Закон Амдала

- В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента.
- Ускорение, которое может быть получено на вычислительной системе из P процессоров, по сравнению с однопроцессорным решением не превышает величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

- α - доля от общего объёма вычислений, которая может быть получена только последовательными расчётами

Закон Амдала

α/p	10	100	1000
0	10	100	1000
10%	5.263	9.174	9.910
25%	3.077	3.883	3.988
40%	2.174	2.463	2.496

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 — время работы MASTER-нити) 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**).

Оптимизация последовательной программы

```
for (int i=0; i < n; i++) {
   for (int j=0; j < n; j++) {
        sum += a[i][j];
   }
}
```

```
for (int j=0; i < n; i++) {
   for (int i=0; j < n; j++) {
        sum += a[i][j];
   }
}
```

// Неправильный порядок доступа
// к массиву a

Оптимизация последовательной программы. Loop unrolling

```
for (int i=2; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] + a[i-1] + b[i-2];
}

// Накладные расходы
// на выполнение цикла
```

```
for (int i=2; i < n; i+=2) {
    a[i] = b[i] + 1;
    c[i] = a[i] + a[i-1] + b[i-2];
    a[i+1] = b[i+1] + 1;
    c[i+1] = a[i+1] + a[i] + b[i-1];
}
```

Оптимизация последовательной программы. Loop fusion

```
 for (int i=0; i < n; i++) {
    a[i] = 3 * b[i];
}

 for (int i=0; i < n; i++) {
    c[i] = 2 * c[i];
    d[i] = a[i] + 2;
}
```

```
// Элементы массива a
//  могут быть вытеснены из кэша
```

```
 for (int i=0; i < n; i++) {
    a[i] = 3 * b[i];
    c[i] = 2 * c[i];
    d[i] = a[i] + 2;
}
```

Оптимизация последовательной программы. Loop fission

```
for (int i=0; i < n; i++) {
    a[i] = exp (i/n);
    for (int j=0; j < n; j++) {
        b[j][i] = c[j][i] + d[i] * e[j];
    }
}

// Неправильный порядок доступа
// к массиву b и c
```

```
for (int i=0; i < n; i++) {
    a[i] = exp (i/n);
}

for (int j=0; j < n; j++) {
    for (int i=0; i < n; i++) {
        b[j][i] = c[j][i] + d[i] * e[j];
    }
}
```

Оптимизация последовательной программы. Loop tiling

```
for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        a[i][j] = b[j][i];
    }
}
```

```
for (int j1=0; j1 < n; j1+=nblockj) {
    for (int i=0; i < n; i++) {
        for (int j2=0; j2 < min(nblockj,n-j1); j2++) {
            a[i][j1+j2] = b[j1+j2][i];
    }
}
```

Оптимизация параллельной программы

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++ )
         temp[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] = temp[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++ )
         temp[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] = temp[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++ )
            temp[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] = temp[i][j];
   }
}
```

Накладные расходы на создание группы нитей

```
#include <stdio.h>
void work(int i) {}
int N=0;

scanf(“%d”,&N);
 #pragma omp parallel for if(N>100)
 for (int i=0; i < N; i++) {
        work(i);
 }
```

Накладные расходы на создание группы нитей

```
#include <omp.h>
void work_on_four_threads(int i) {}

 #pragma omp parallel num_threads(4)
{
    int iam = omp_get_thread_num ();
    work_on_four_threads(iam);
 }


omp_set_num_threads (4);
#pragma omp parallel
{
    int iam = omp_get_thread_num ();
    work_on_four_threads(iam);
 }
```

Несбалансированная нагрузка нитей

```
void work(int i, int j) {}
int num_threads = omp_get_max_threads();

/*N > num_threads */
#pragma omp for
 for (int i=0; i < N; i++) {
    for (int j=0; j < M; j++) {
        work(i, j);
    }
}

/*(N<=num_threads) && (M>N)*/
 for (int i=0; i < N; i++) {
    #pragma omp for
    for (int j=0; j < M; j++) {
        work(i, j);
    }
}
```

```
 /*OpenMP 3.0*/
#pragma omp for collapse(2)
for (int i=0; i < N; i++) {
    for (int j=0; j < M; j++) {
        work(i, j);
    }
 }
```

Вложенный параллелизм

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

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

```
 /*OpenMP 3.0*/
#pragma omp parallel for collapse(2)
for (int i=0; i < N; i++) {
    for (int j=0; j < M; j++) {
         work(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 = 0; i < 100; i++)
      A[i]=0.;
```

Балансировка нагрузки нитей. Клауза schedule

```
#pragma omp parallel for schedule(static, 10)
  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 1-10, 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**

Балансировка нагрузки нитей. Клауза schedule

```
#pragma omp parallel for schedule(dynamic, 15)
  for(int i = 0; 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.**

Балансировка нагрузки нитей. 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 = 0; 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 99 итерации**.

Балансировка нагрузки нитей. Клауза schedule

```
#pragma omp parallel for schedule(runtime)
  for(int i = 0; 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 функций системы поддержки:

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

Балансировка нагрузки нитей. Клауза schedule

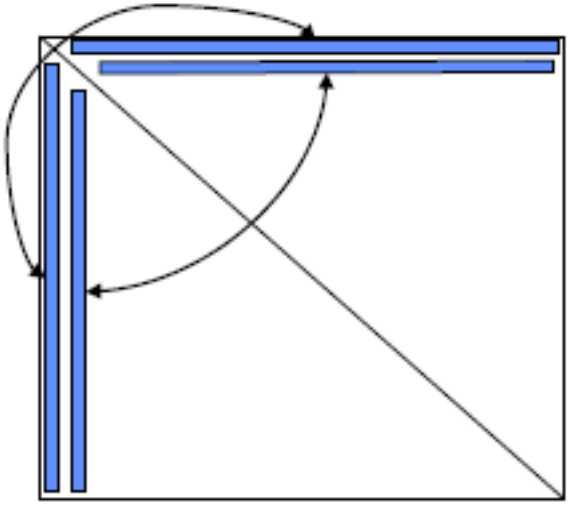
```
#pragma omp parallel for schedule(auto)
  for(int i = 0; i < 100; i++)
```

Способ распределения витков цикла между нитями определяется реализацией компилятора.

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

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

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.

Потери из-за синхронизации нитей

```
#define  Max(a,b) ((a)>(b)?(a):(b))
double MAXEPS   = 0.5;
double grid[L][L], temp[L][L],eps;
#pragma omp parallel default (none) shared (grid,temp,eps)
for (int it=0;it<ITMAX; it++) {
      #pragma omp barrier
      #pragma omp single
          eps= 0.;
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ ) {
            #pragma omp critical
               eps =  Max(fabs(temp[i][j]-grid[i][j]),eps);
            grid[i][j] = temp[i][j];
         }
      #pragma omp for
       for (int i=1; i<N-1; i++ )
          for (int j=1; j<N-1; j++ )
             temp[i][j] = 0.25 * ( grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
       if (eps < MAXEPS) break;
}
```

Потери из-за синхронизации нитей

```
#pragma omp for
for (int i=1; i<N-1; i++ )
    for (int j=1; j<N-1; j++ ) {
        #pragma omp critical
            eps =  Max(fabs(temp[i][j]-grid[i][j]),eps);
        grid[i][j] = temp[i][j];
    }

#pragma omp for
for (int i=1; i<N-1; i++ )
    for (int j=1; j<N-1; j++ ) {
        double tmp = fabs(temp[i][j]-grid[i][j]);
        #pragma omp flush (eps)
        if (tmp > eps) {
            #pragma omp critical
                eps =  Max(tmp,eps);
        }
        grid[i][j] = temp[i][j];
    }
```

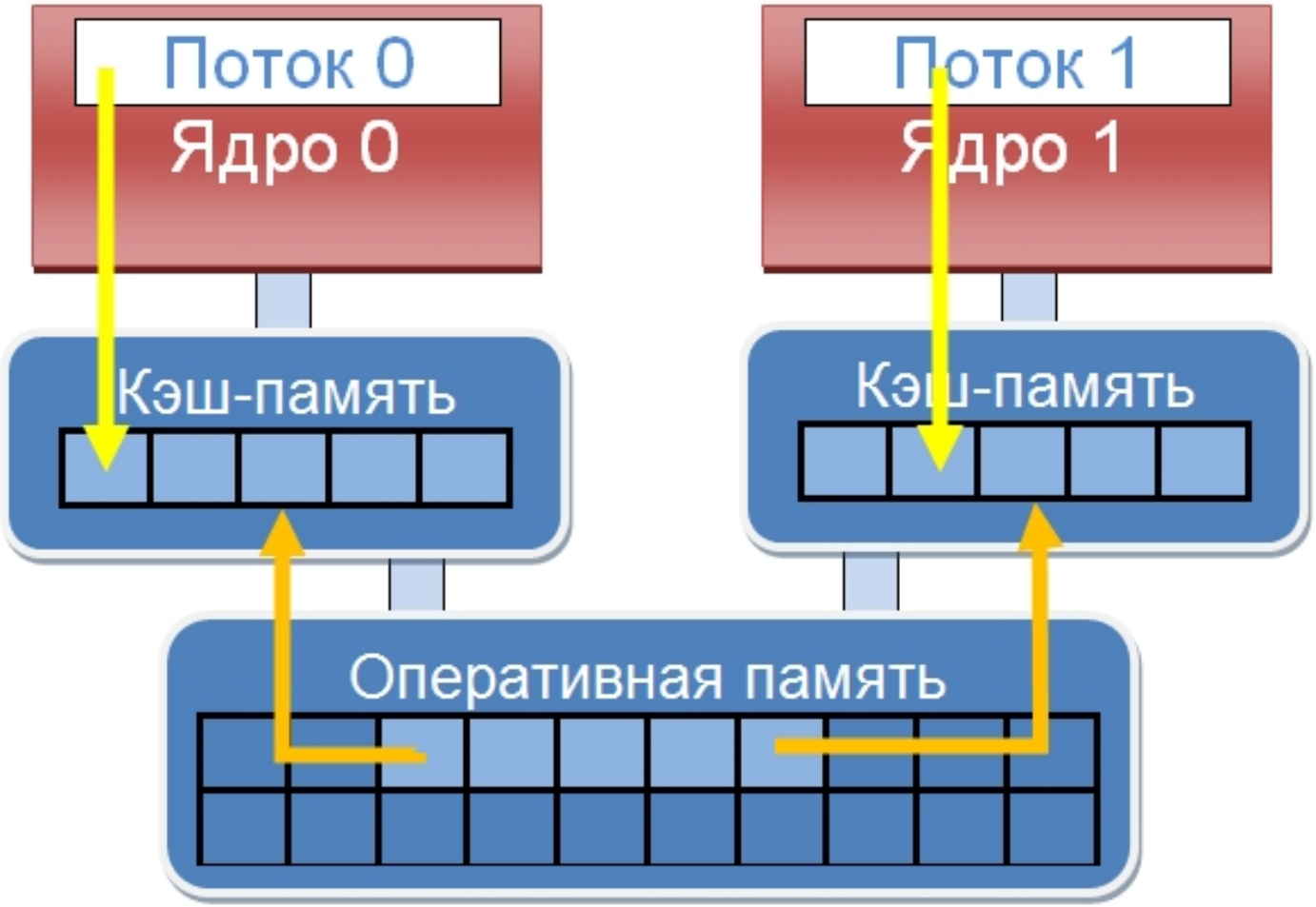
Локализация данных

```
#define  Max(a,b) ((a)>(b)?(a):(b))
double MAXEPS   = 0.5;
double grid[L][L], temp[L][L],eps;
#pragma omp parallel default (none) shared (grid,temp,eps)
for (int it=0;it<ITMAX; it++) {
      double localeps=0.;
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ ) {
            localeps =  Max(fabs(temp[i][j]-grid[i][j]),localeps);
            grid[i][j] = temp[i][j];
         }
      #pragma omp single
         eps= 0.;
      #pragma omp critical
        eps =  Max(eps,localeps);
      #pragma omp for
       for (int i=1; i<N-1; i++ )
          for (int j=1; j<N-1; j++ )
             temp[i][j] = 0.25 * ( grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
       if (eps < MAXEPS) break;
}
```

Локализация данных

```
double grid[L][L], temp[L][L],eps;
int num_threads = omp_get_max_threads();
double *localeps = (double *)malloc(num_threads*sizeof(double));
#pragma omp parallel default (none) shared (grid,temp,eps,localeps)
for (int it=0;it<ITMAX; it++) {
      int iam = omp_get_thread_num ();
      localeps[iam]=0.;
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ ) {
            localeps[iam] =  Max(fabs(temp[i][j]-grid[i][j]),localeps[iam]);
            grid[i][j] = temp[i][j];
         }
     #pragma omp single nowait
        for (int i=0, eps = 0.;i<num_threads;i++) eps =  Max(localeps[i],eps);
      #pragma omp for
       for (int i=1; i<N-1; i++ )
          for (int j=1; j<N-1; j++ )
             temp[i][j] = 0.25 * ( grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
       if (eps < MAXEPS) break;
}
```

False-Sharing



False Sharing. Array Padding

```
double grid[L][L], temp[L][L],eps;
double localeps[NUM_THREADS][CACHE_LINE_SIZE];
#pragma omp parallel default (none) shared (grid,temp,eps,localeps)
for (int it=0;it<ITMAX; it++) {
      int iam = omp_get_thread_num ();
      localeps[iam][0]=0.;
      #pragma omp for
      for (int i=1; i<N-1; i++ )
         for (int j=1; j<N-1; j++ ) {
            localeps[iam][0] =  Max(fabs(temp[i][j]-grid[i][j]),localeps[iam][0]);
            grid[i][j] = temp[i][j];
         }
     #pragma omp single nowait
        for (int i=0, eps = 0.;i<NUM_THREADS;i++) eps =  Max(localeps[i][0],eps);
      #pragma omp for
       for (int i=1; i<N-1; i++ )
          for (int j=1; j<N-1; j++ )
             temp[i][j] = 0.25 * ( grid[i-1][j] + grid[i+1][j] + grid[i][j-1] + grid[i][j+1]);
      if (eps < MAXEPS) break;
}
```

Потери из-за синхронизации нитей

```
#pragma omp parallel shared(a) firstprivate(b,c)
{
        ...
        #pragma omp critical
        {
           a+=2*b;
           c+=b*b;
        }
        ...
}
```

Потери из-за синхронизации нитей

```
#pragma omp parallel shared(a) firstprivate(b,c)
{
        ...
        #pragma omp critical
        {
           a+=2*b;
           c+=b*b;
        }
        ...
}
```

```
pragma omp parallel shared(a) firstprivate(b,c)
{
        ...
        #pragma omp critical
           a+=2*b;
         c+=b*b;
}
```

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:

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

Потери из-за синхронизации нитей

```
#pragma omp parallel shared(a) firstprivate(b,c)
{
        ...
        #pragma omp critical
        {
           a+=2*b;
           c+=b*b;
        }
        ...
}
```

```
pragma omp parallel shared(a) firstprivate(b,c)
{
        ...
        #pragma omp atomic
           a+=2*b;
         c+=b*b;
}
```

Распределение циклов с зависимостью по данным. Организация конвейерного выполнения цикла.

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

Распределение циклов с зависимостью по данным.

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

Распределение циклов с зависимостью по данным.

```
#pragma omp parallel reduction(max:eps) shared(A)
    #pragma omp for ordered(2)
        for (int i = 1; i < N; i++)
            for (int j = 1; j < N;  j++)
            {
                double e =  A[i][j];
#pragma omp ordered depend(sink:i-1,j)  depend(sink:i,j-1)
                A[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.0;
#pragma omp ordered depend(source)
                eps = Max(eps, fabs(e - A[i][j]));
            }
```

Переменная OMP_WAIT_POLICY.

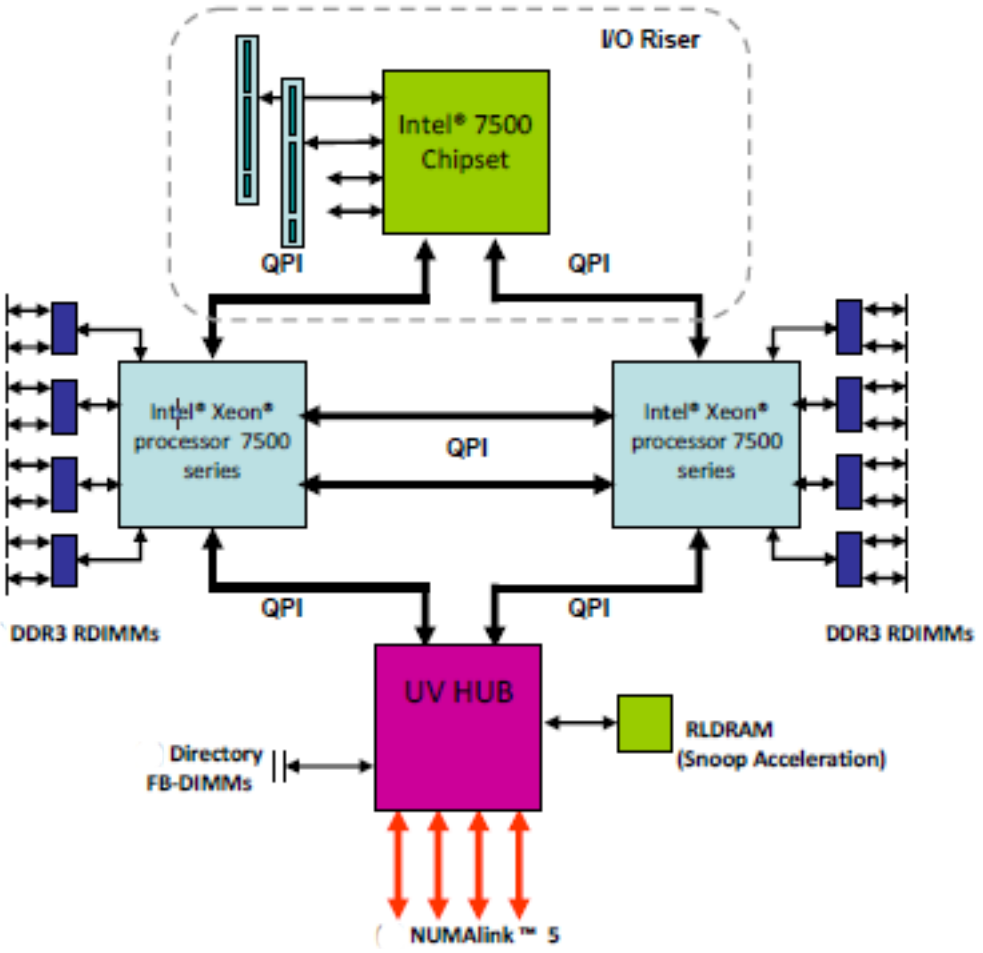
Подсказка OpenMP-компилятору о желаемом поведении нитей во время ожидания.

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

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)

Системы с неоднородным доступом к памяти (NUMA)



- Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти.
- Модули объединены с помощью высокоскоростного коммутатора.
- Поддерживается единое адресное пространство.
- Доступ к локальной памяти в несколько раз быстрее, чем к удаленной.

Системы с неоднородным доступом к памяти (NUMA)

SGI Altix UV (UltraVioloet) 1000

- 256 Intel® Xeon® quad-, six- or eight-core 7500 series (2048 cores)
- 16 TB памяти
- Interconnect Speed 15 ГБ/с, 1мкс

http://www.sgi.com/products/servers/altix/uv/

Оптимизация для систем типа NUMA

```
 #pragma omp parallel for
    for (int i=1; i<N-1; i++ )
        for (int j=1; j<N-1; j++ ) {       // first-touch algoritm
            temp[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++ )
           temp[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] = temp[i][j];
}
```

Intel Thread Profiler

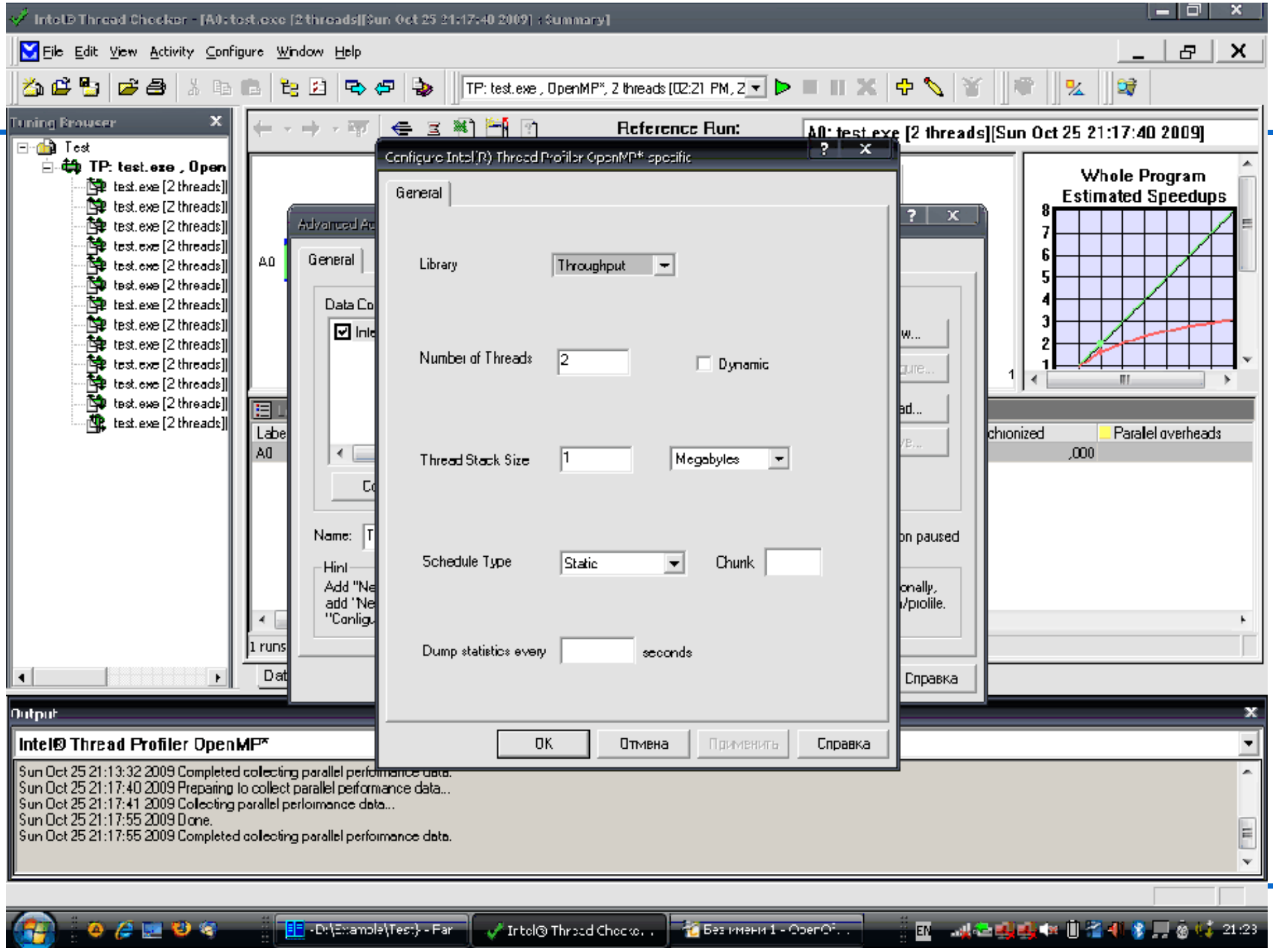
Предназначен для анализа производительности OpenMP-приложений или многопоточных приложений с использованием потоков Win32 API и POSIX.

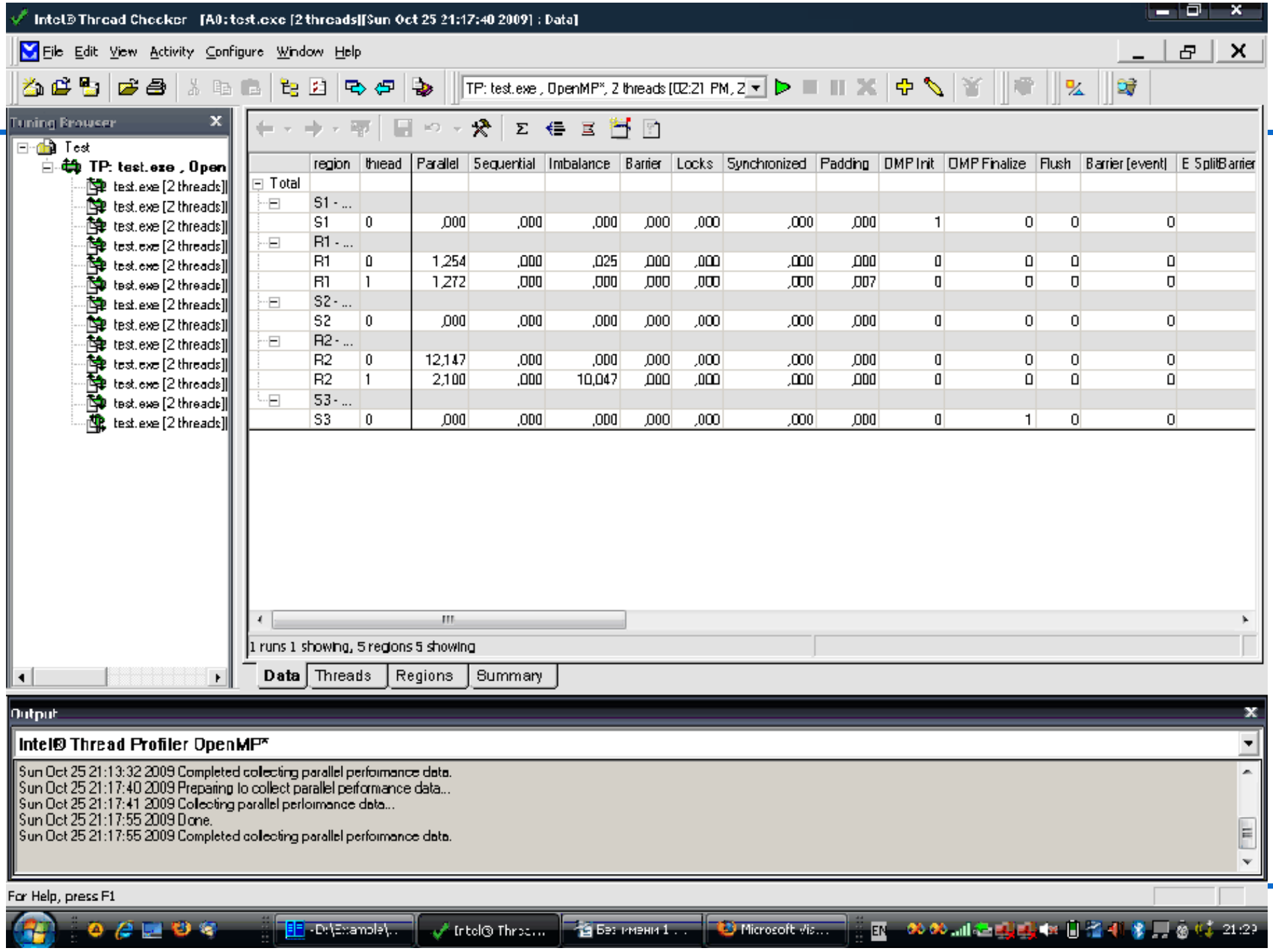
Визуализация выполнения потоков во времени помогает понять их функции и взаимодействие.

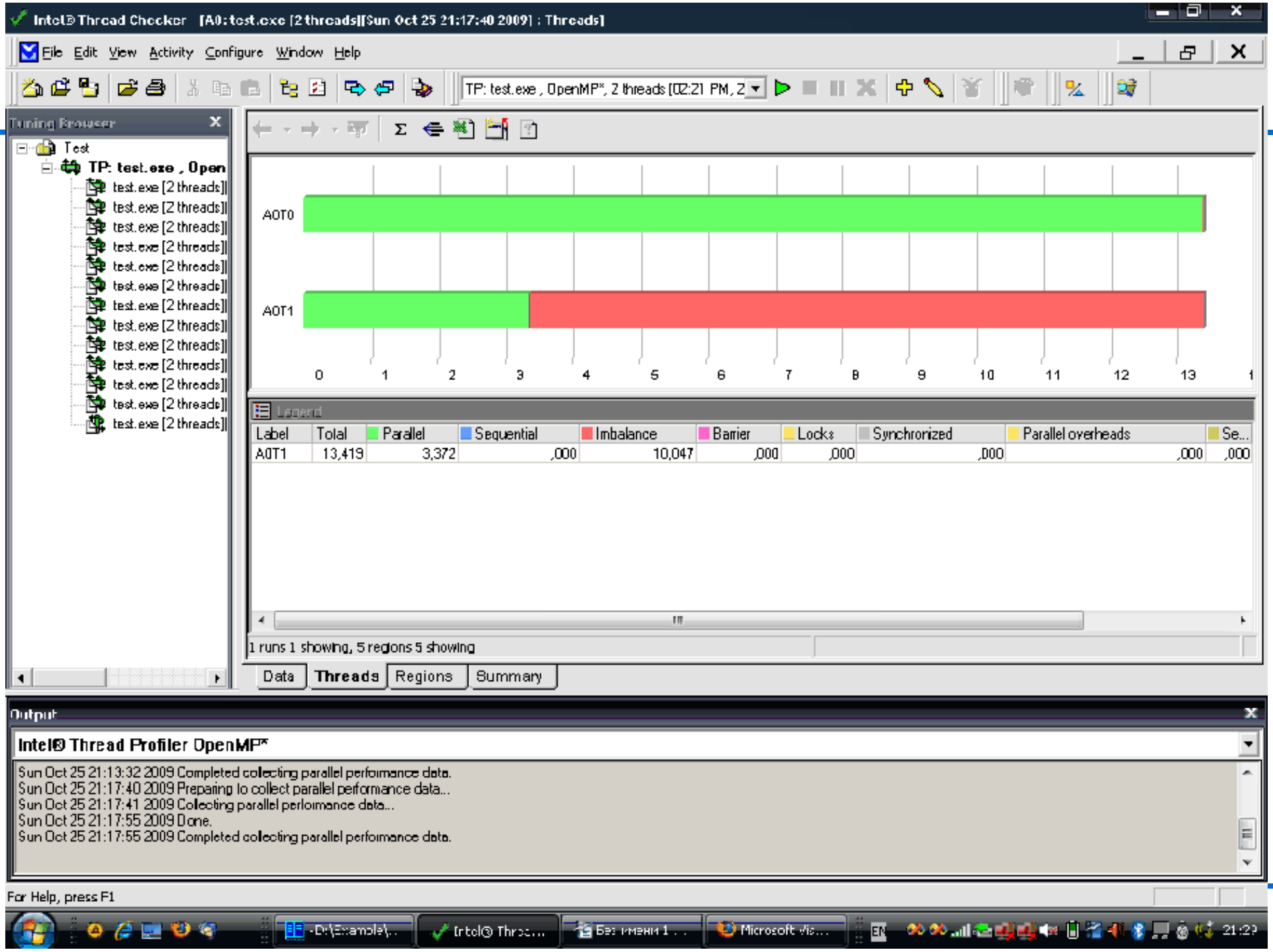
Инструмент указывает на узкие места, снижающие производительность.

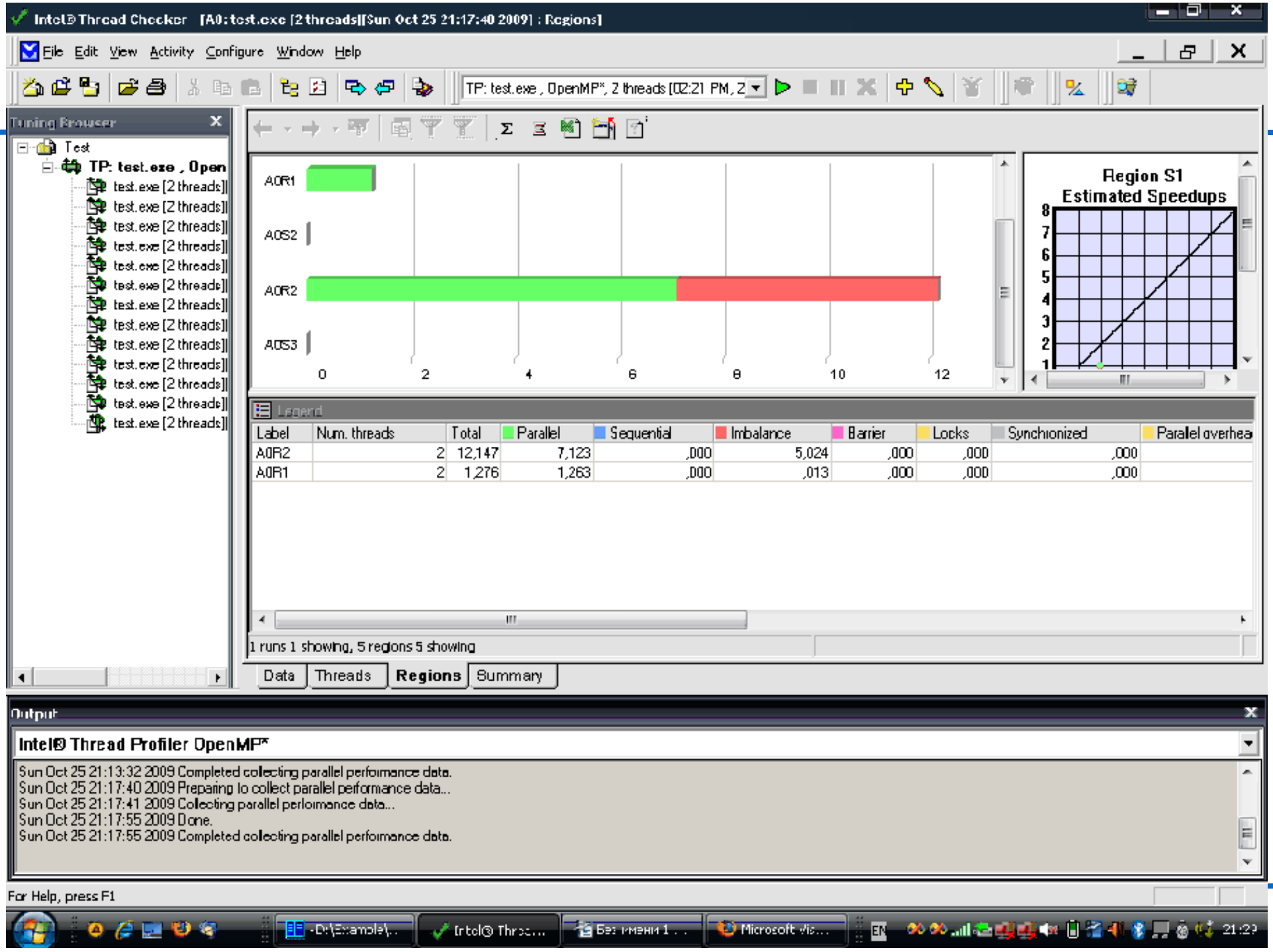
Инструментация программы:

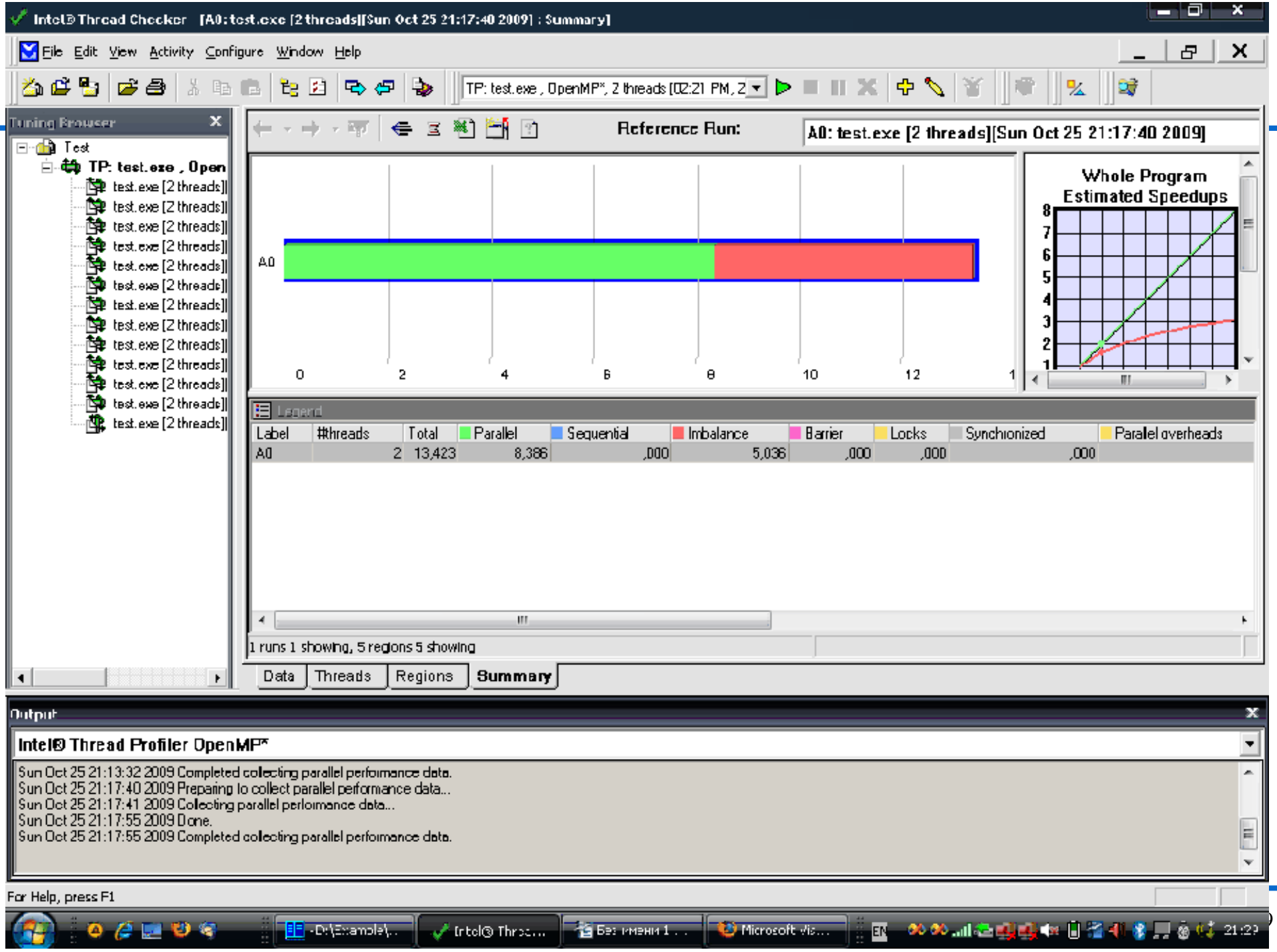
- **Linux: -g [-openmp-profile]**
- **Windows: /Zi [/-Qopenmp-profile], link with /fixed:no**


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.

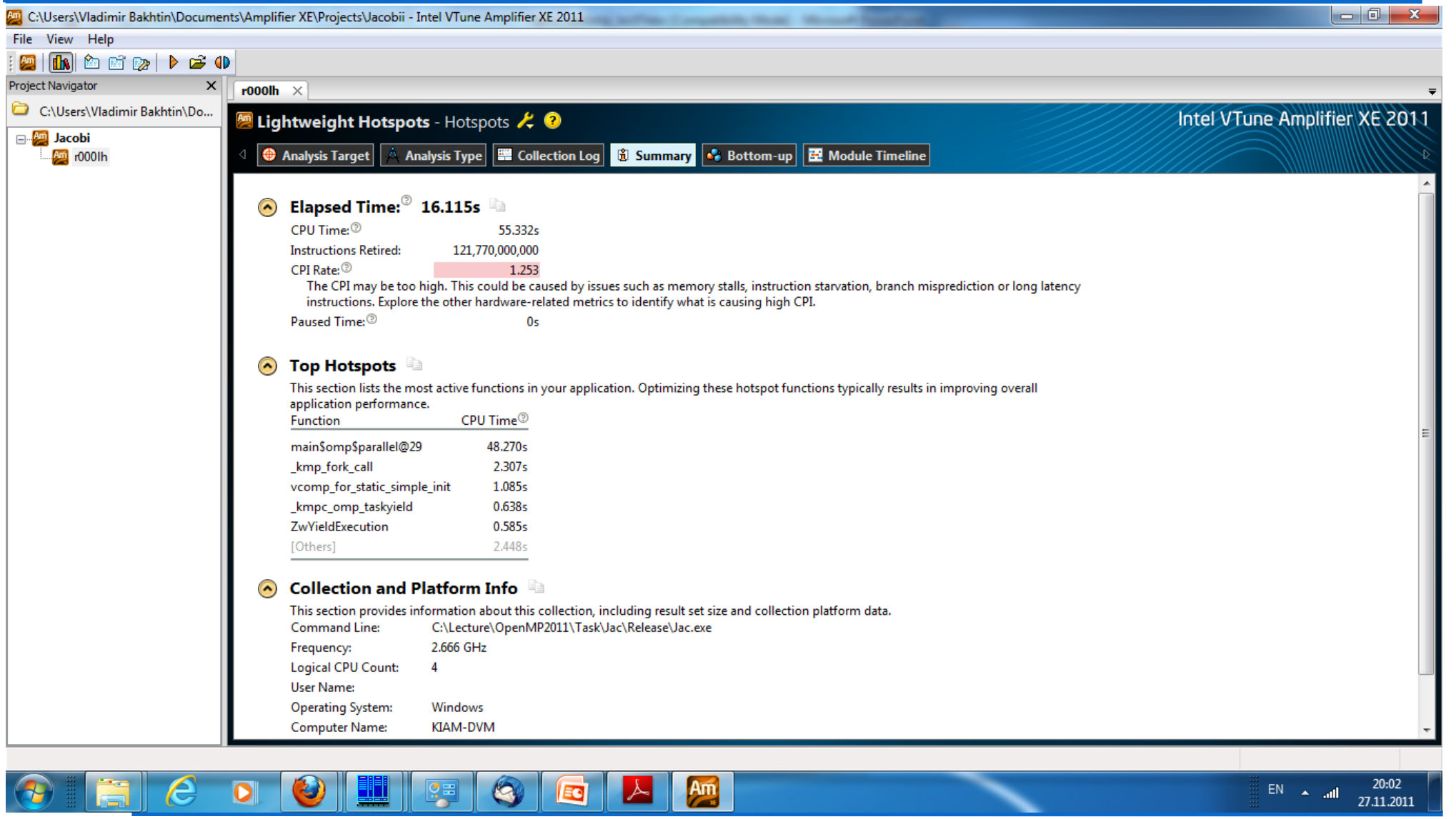

Intel® Thread Checker [A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Data]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
region thread Parallel Sequential Imbalance Barrier Locks Synchronized 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...
Без имени 1 - ...
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

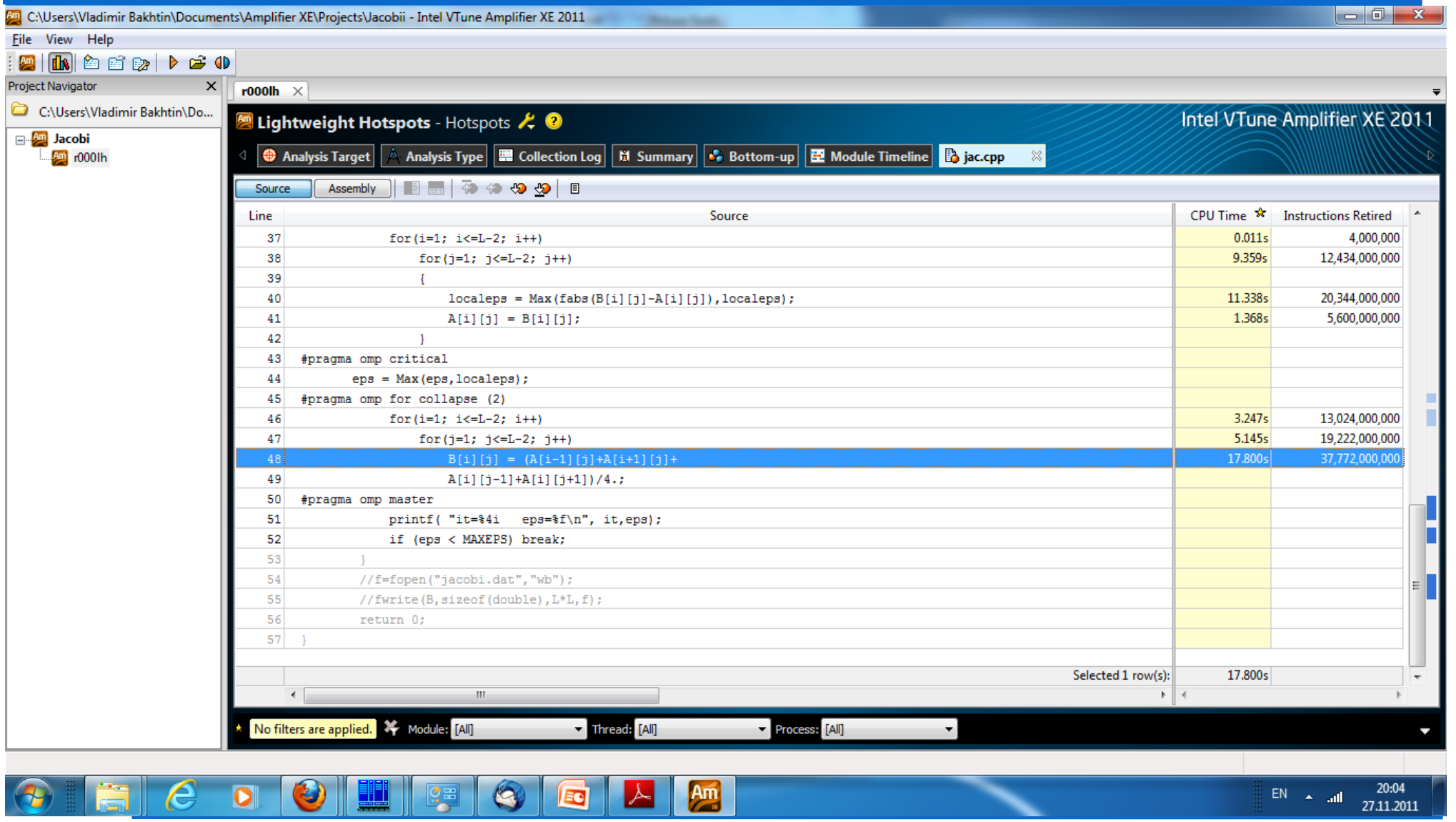

Intel® Thread Checker [A0:test.exe [2 threads][Sun Oct 25 21:17:40 2009] : Regions]
File Edit View Activity Configure Window Help
TP: test.exe , OpenMP*, 2 threads [02:21 PM, 2
Tuning Browser
Test
TP: test.exe , Open
test.exe [2 threads]
A0R1
A0S2
A0R2
A0S3
0 2 4 6 8 10 12
Region S1
Estimated Speedups
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
21:29


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

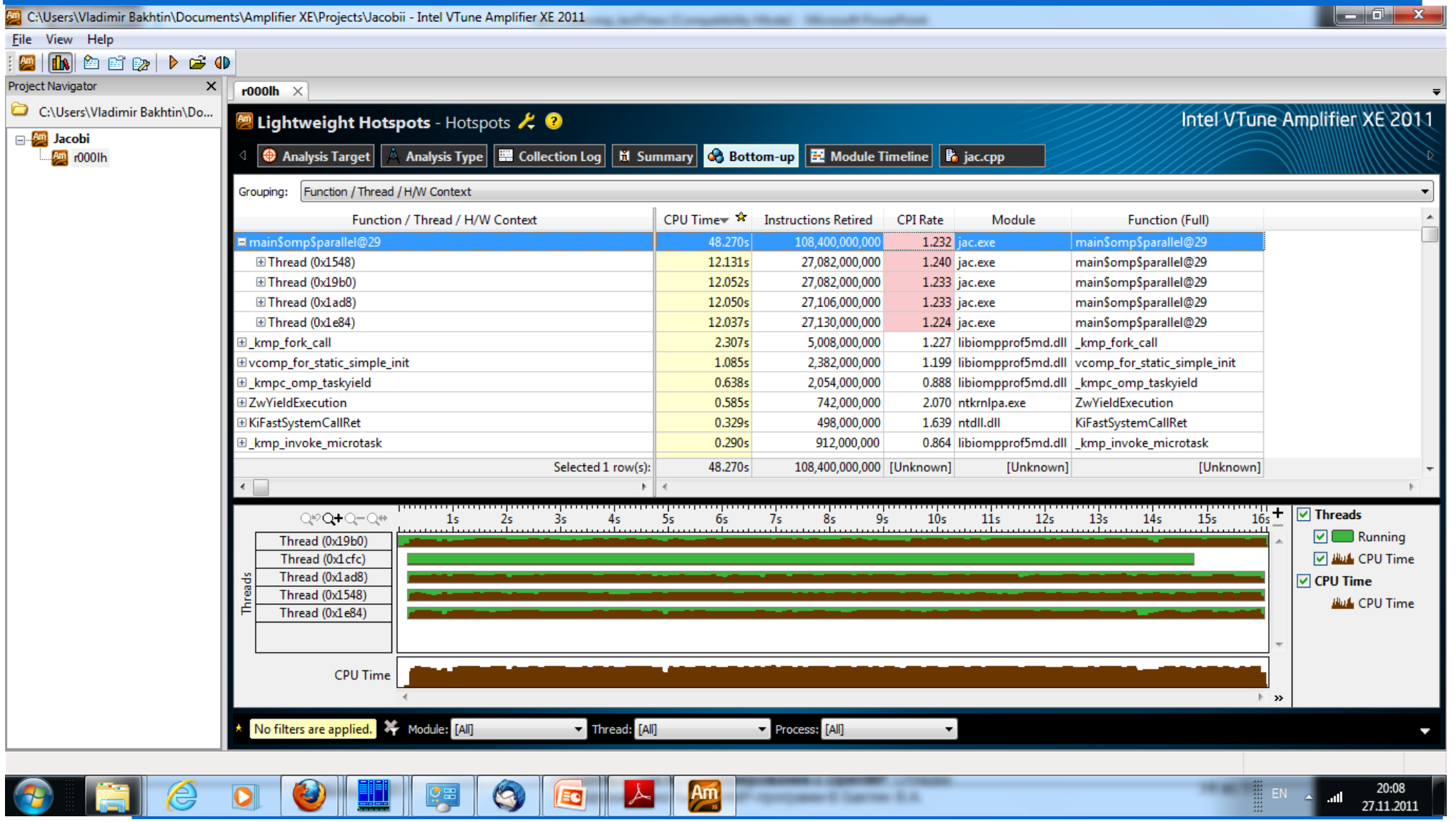
Intel Vtune Amplifier XE 2018



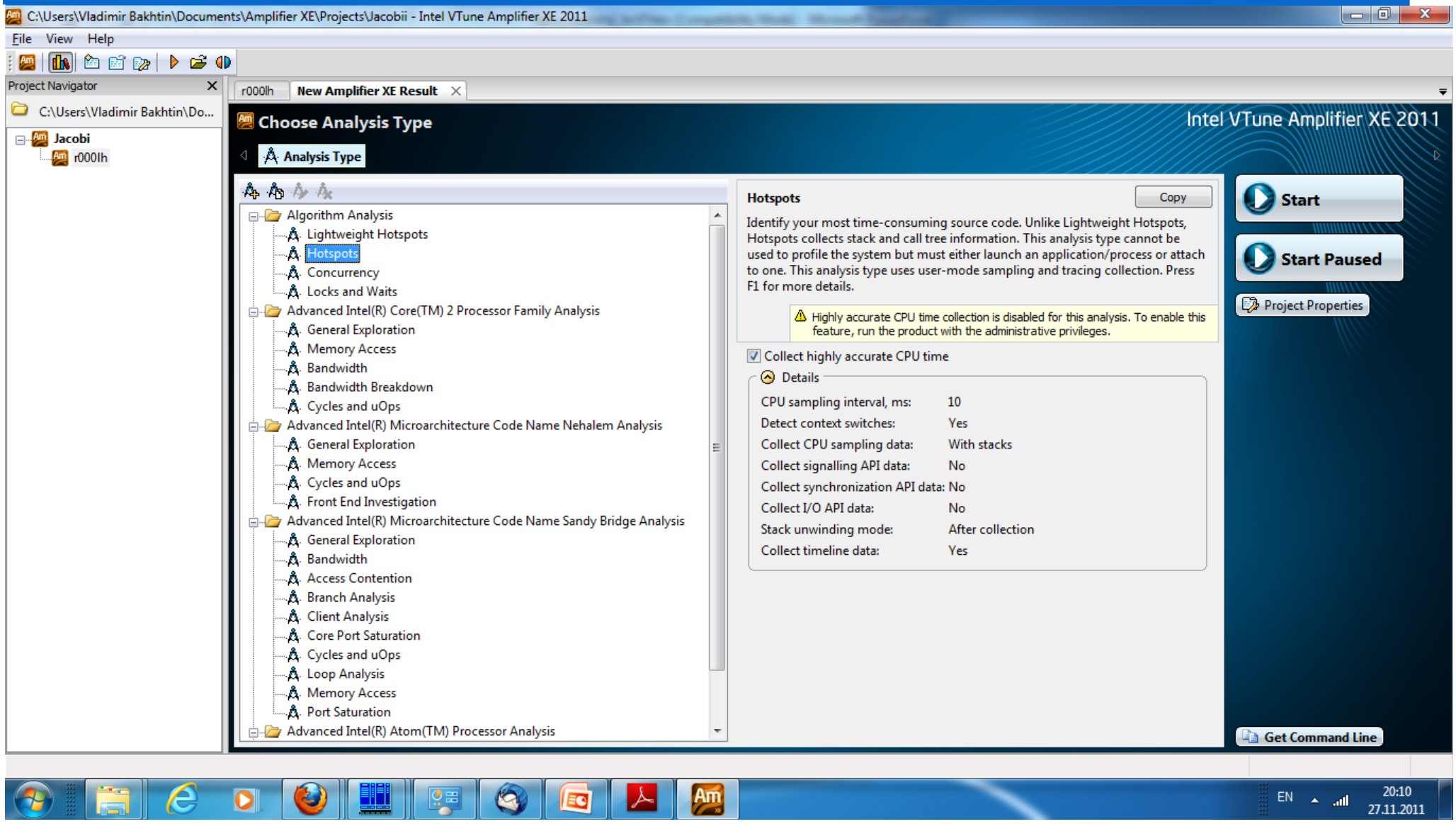
Intel Vtune Amplifier XE 2018



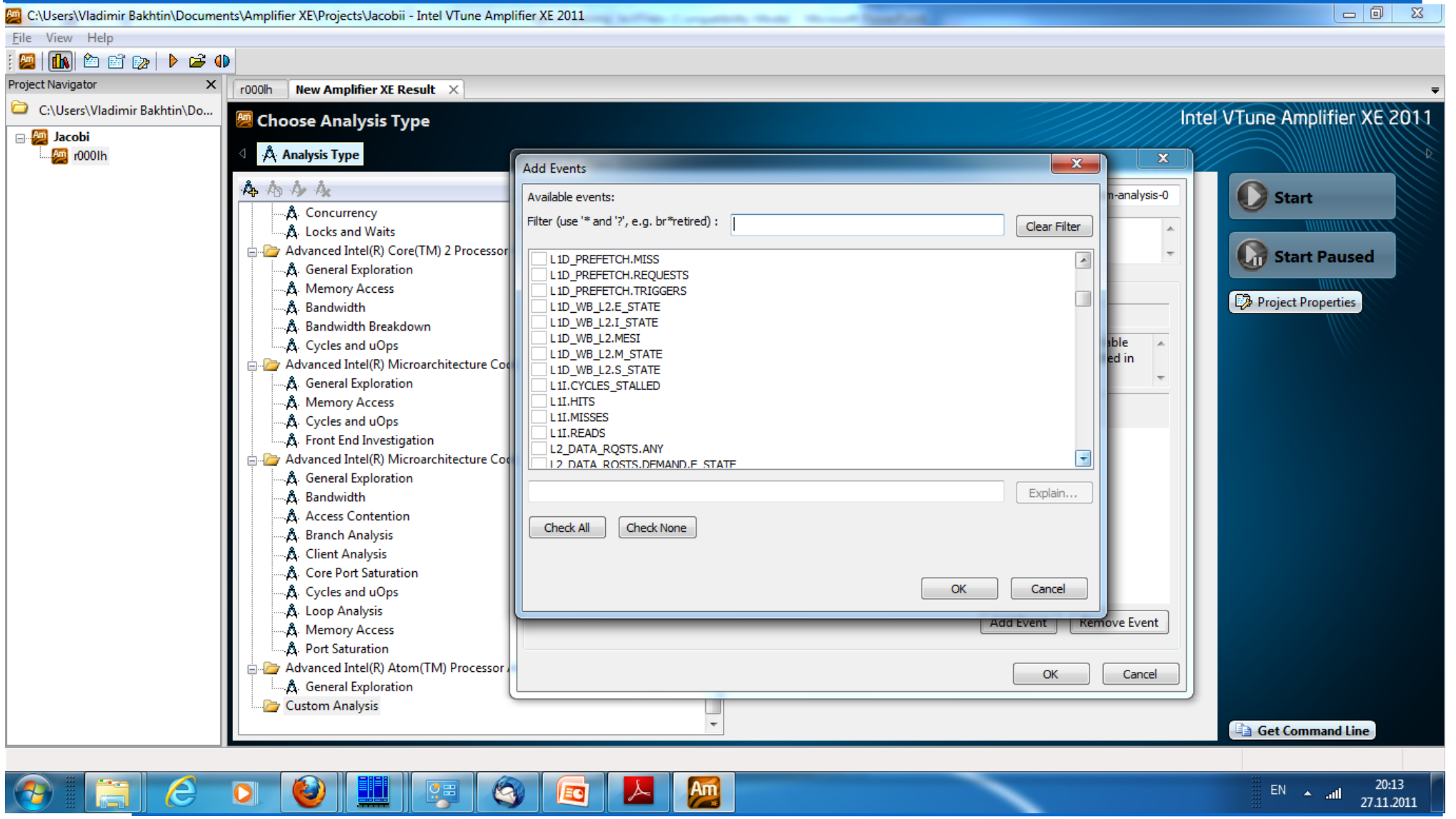
Intel Vtune Amplifier XE 2018



Intel Vtune Amplifier XE 2018

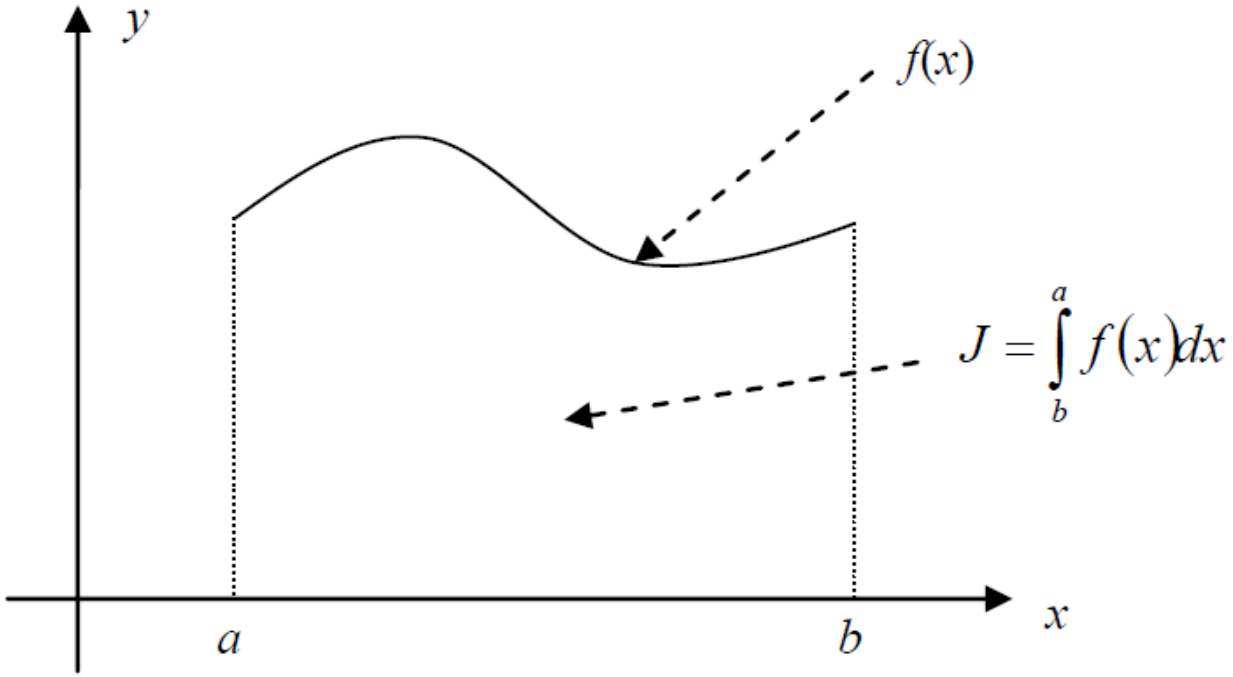


Intel Vtune Amplifier XE 2018



Пример. Вычисление определенного интеграла

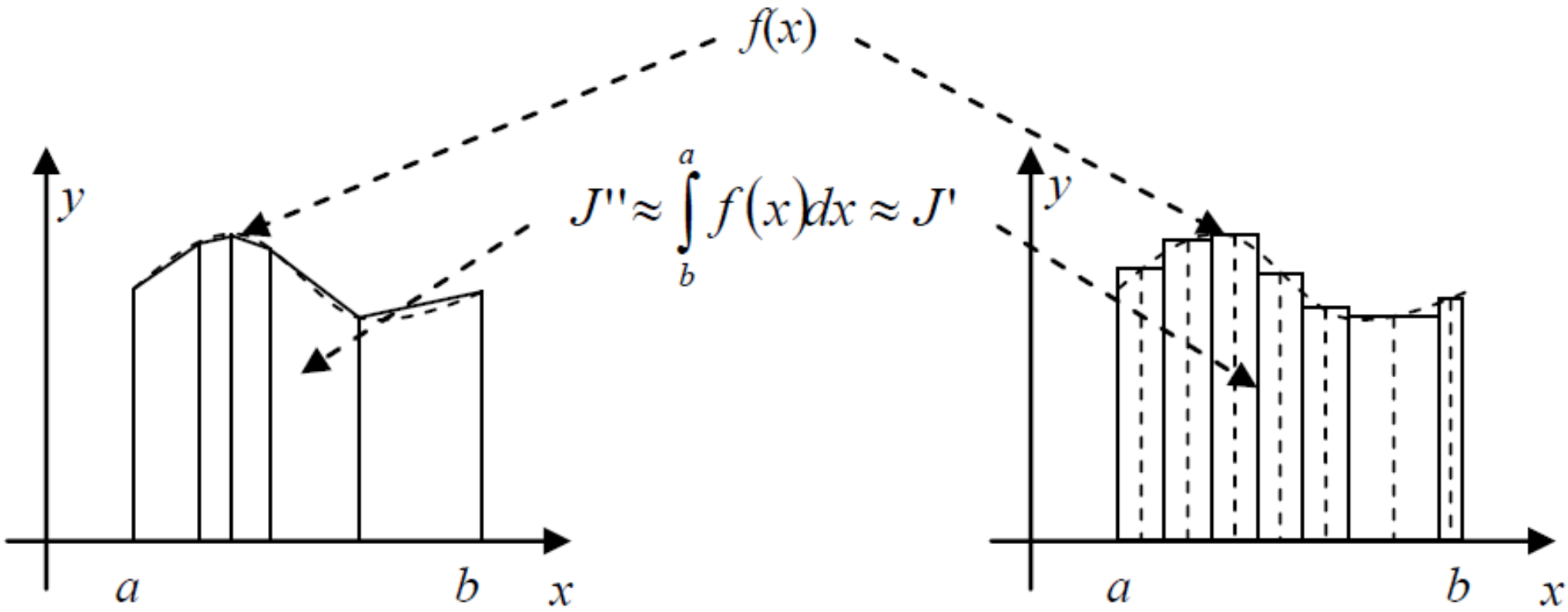
Пусть на отрезке *[a, b] задана непрерывная неотрицательная функция. В этом случае значение определенного интеграла от f(x) на отрезке [a, b] совпадает с площадью фигуры, ограниченной графиком функции, осью Ox и прямыми x = a, x = b.*



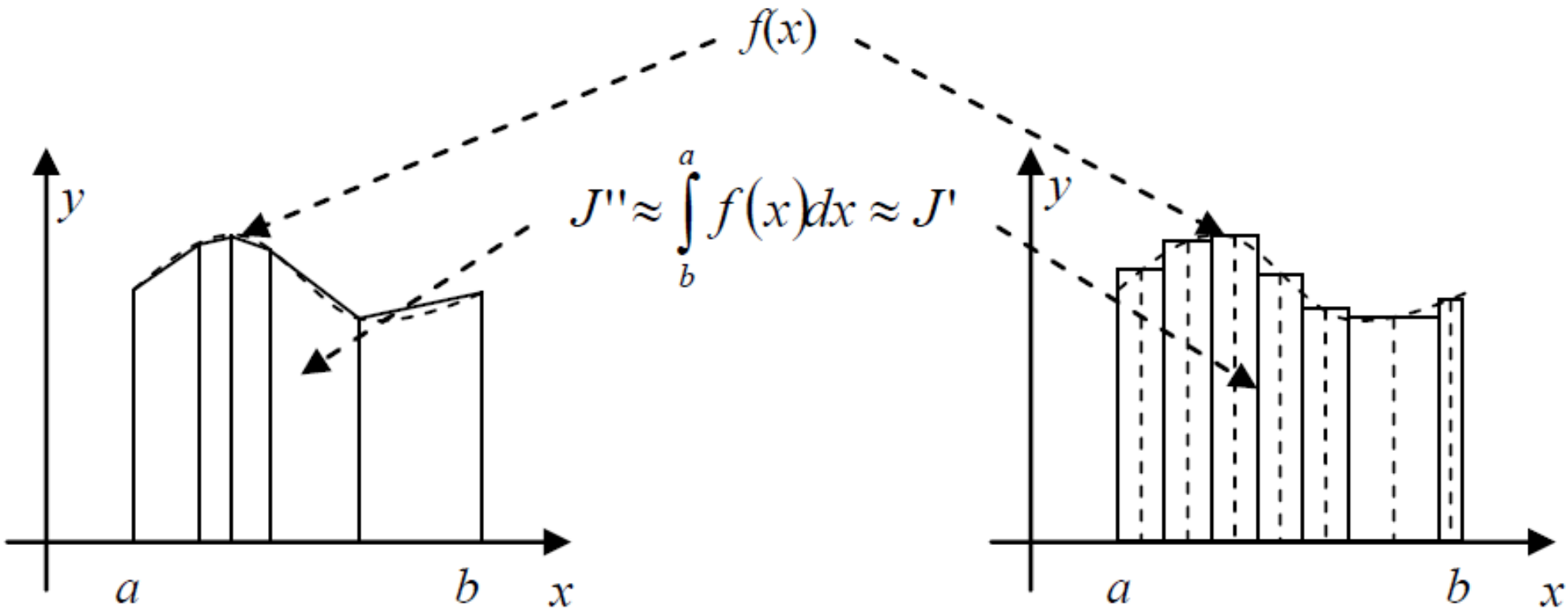
Спасибо Е.А. Козинову и А.В. Сысоеву:

https://software.intel.com/sites/default/files/m/d/4/1/d/8/LW_Integral_doc.pdf

Численное вычисление интеграла методами трапеций и прямоугольников



Численное вычисление интеграла методами трапеций и прямоугольников



Численное вычисление интеграла методом прямоугольников и методом параллелепипедов

$$\begin{cases} J = \int_b^a f(x)dx \approx h\sum_{i=0}^{N-1} f(x_i) \\ x_i = a + ih + \frac{h}{2} \end{cases},$$

где *N*-количество отрезков интегрирования по оси *x*,

а *h=(b-a)/N*

$$\begin{cases} J = \int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y)dxdy \approx h_1 h_2 \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} f(x_i, y_i) \\ x_i = a_1 + ih_1 + \frac{h_1}{2}, y_i = a_2 + ih_2 + \frac{h_2}{2} \end{cases},$$

где *M*-количество отрезков интегрирования по оси *y*,

h1=(b1-a1)/N,

h2=(b2-a2)/M

Численное вычисление интеграла методом параллелепипедов

$$\begin{cases} f(x,y) = \dfrac{e^{\sin(\pi x)\cdot\cos(\pi y)} + 1}{(b_1 - a_1)\cdot(b_2 - a_2)}, \text{где} \\ (x,y) \in [a_1, b_1] \times [a_2, b_2] \\ a_1 = 0, b_1 = 16 \\ a_2 = 0, b_2 = 16 \end{cases}.$$

Численное вычисление интеграла методом параллелепипедов. Последовательная программа

```
void integral(const double a1, const double b1, const double a2, const double b2, const double h, double *res)
{
	int i, j, n1, n2;
	// локальная переменная для подсчета интеграла
	double sum;
	// координата точки сетки по оси x
	double x;
	// координата точки сетки по оси y
	double y;
	// количество точек сетки интегрирования : n1 - по координате x,  n2 - по координате y
	n1 = (int)((b1 - a1) / h);
	n2 = (int)((b2 - a2) / h);
	sum = 0.0;
	for(i = 0; i < n1; i++)
	{
		for(j = 0; j < n2; j++) {
			// вычисление координат точек
			x = a1 + i * h + h / 2;
			y = a2 + j * h + h / 2;
			// вычисление интеграла
			sum += ((exp(sin(x * PI) * cos(y * PI)) + 1) / ((b1 - a1) * (b2 - a2))) * h * h;
		}
	}
	*res = sum;
}
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 3.589169

Численное вычисление интеграла методом параллелепипедов. Параллельная програма (вариант 1)

```
void integral(const double a1, const double b1, const double a2, const double b2, const double h, double *res)
{
	int i, j, n1, n2;
	// локальная переменная для подсчета интеграла
	double sum;
	// координата точки сетки по оси x
	double x;
	// координата точки сетки по оси y
	double y;
	// количество точек сетки интегрирования : n1 - по координате x,  n2 - по координате y
	n1 = (int)((b1 - a1) / h);
	n2 = (int)((b2 - a2) / h);
	sum = 0.0;
#pragma omp parallel for private(i,j,x,y) reduction(+:sum)
	for(i = 0; i < n1; i++)
	{
		for(j = 0; j < n2; j++) {
			// вычисление координат точек
			x = a1 + i * h + h / 2;
			y = a2 + j * h + h / 2;
			// вычисление интеграла
			sum += ((exp(sin(x * Pl) * cos(y * Pl)) + 1) / ((b1 - a1) * (b2 - a2))) * h * h;
		}
	}
	*res = sum;
}
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 0.681427

Численное вычисление интеграла методом параллелепипедов. Параллельная програма (вариант 2)

```
void integral(const double a1, const double b1, const double a2, const double b2, const double h, double *res)
{
	int i, j, n1, n2;
	// локальная переменная для подсчета интеграла
	double sum;
	// координата точки сетки по оси x
	double x;
	// координата точки сетки по оси y
	double y;
	// количество точек сетки интегрирования : n1 - по координате x,  n2 - по координате y
	n1 = (int)((b1 - a1) / h);
	n2 = (int)((b2 - a2) / h);
	sum = 0.0;
#pragma omp parallel for collapse(2) private(i,j,x,y) reduction(+:sum)
	for(i = 0; i < n1; i++)
	{
		for(j = 0; j < n2; j++) {
			// вычисление координат точек
			x = a1 + i * h + h / 2;
			y = a2 + j * h + h / 2;
			// вычисление интеграла
			sum += ((exp(sin(x * Pl) * cos(y * Pl)) + 1) / ((b1 - a1) * (b2 - a2))) * h * h;
		}
	}
	*res = sum;
}
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 1.417066

Численное вычисление интеграла. Последовательная программа (вариант 2)

```
void integral(const double a1, const double b1, const double a2, const double b2, const double h, double *res)
{
        int i, j, n1, n2;
        double sum, x, y;
        double *sinx;
        double *cosy;
         n1 = (int)((b1 - a1) / h);
        n2 = (int)((b2 - a2) / h);
        sinx = new double [n1];
        for(i = 0; i < n1; i++)
        {
                x = a1 + i * h + h / 2;
                sinx[i] = sin(x * PI); // вычисление значений sin(x * pi)
        }
        cosy = new double [n2];
        for(j = 0; j < n2; j++)
        {
                y = a2 + j * h + h / 2;
                cosy[j] = cos(y * PI); // вычисление значений cos(y * pi)
        }
        sum = 0.0;
        for(i = 0; i < n1; i++)
        {
                for(j = 0; j < n2; j++) {
                    sum += ((exp(sinx[i] * cosy[j]) + 1) / ((b1 - a1) * (b2 - a2))) * h * h;
                }
        }
       delete [] sinx;
       delete [] cosy;
       *res = sum;
}
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 1.06952

Численное вычисление интеграла. Последовательная программа (вариант 3)

```
#define BLOCK_SIZE 2000
// правильность размера блока
assert((n1 % BLOCK_SIZE) == 0);
assert((n2 % BLOCK_SIZE) == 0);
// Выделение памяти
sinx = new double [BLOCK_SIZE];
cosy = new double [BLOCK_SIZE];
// проход по всем блокам
for(ii = 0; ii < n1; ii += BLOCK_SIZE) { // вычисление значений sin(x
        for(i = 0; i < BLOCK_SIZE; i++) {
                x = a1 + i * h + ii + h / 2;
                sinx[i] = sin(x * PI);
        }
        for(jj = 0; jj < n2; jj += BLOCK_SIZE) { // вычисление значений cos(y * pi)
                for(j = 0; j < BLOCK_SIZE; j++) {
                        y = a2 + j * h + jj + h / 2;
                        cosy[j] = cos(y * PI);
                }
                for(i = 0; i < BLOCK_SIZE; i++) {
                        for(j = 0; j < BLOCK_SIZE; j++) {
                                // вычисление интеграла
                                sum += (exp(sinx[i] * cosy[j]) + 1)/((b1 - a1) * (b2 - a2)) * h * h;

                        }
                }
        }
}
…
*res = sum;
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 1.056832

Численное вычисление интеграла. Последовательная программа (вариант 4)

```
#define BLOCK_SIZE 2000
 double value;
// правильность размера блока
assert((n1 % BLOCK_SIZE) == 0);
assert((n2 % BLOCK_SIZE) == 0);
// Выделение памяти
 sinx = new double [BLOCK_SIZE];
cosy = new double [BLOCK_SIZE];
value =  h * h / ((b1 - a1) * (b2 - a2));
// проход по всем блокам
for(ii = 0; ii < n1; ii += BLOCK_SIZE) { // вычисление значений sin(x * pi)
        for(i = 0; i < BLOCK_SIZE; i++) {
                x = a1 + i * h + ii + h / 2;
                sinx[i] = sin(x * PI);
        }
        for(jj = 0; jj < n2; jj += BLOCK_SIZE) { // вычисление значений cos(y * pi)
                for(j = 0; j < BLOCK_SIZE; j++) {
                        y = a2 + j * h + jj + h / 2;
                        cosy[j] = cos(y * PI);
                }
                for(i = 0; i < BLOCK_SIZE; i++) {
                        for(j = 0; j < BLOCK_SIZE; j++) {
                                // вычисление интеграла
                                sum += (exp(sinx[i] * cosy[j]) + 1) * value;

                        }
                }
        }
}
…
*res = sum;
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 0.988739

Численное вычисление интеграла. Параллельная программа (вариант 2)

```
// проход по всем блокам
#pragma omp parallel private(ii,jj,i,j,x,y) reduction(+:sum)
for(ii = 0; ii < n1; ii += BLOCK_SIZE) { // вычисление значений sin(x * pi)
        #pragma omp for
        for(i = 0; i < BLOCK_SIZE; i++) {
                x = a1 + i * h + ii + h / 2;
                sinx[i] = sin(x * PI);
        }
        for(jj = 0; jj < n2; jj += BLOCK_SIZE) { // вычисление значений
                #pragma omp for
                for(j = 0; j < BLOCK_SIZE; j++) {
                        y = a2 + j * h + jj + h / 2;
                        cosy[j] = cos(y * PI);
                }
                #pragma omp for
                for(i = 0; i < BLOCK_SIZE; i++) {
                        for(j = 0; j < BLOCK_SIZE; j++) {
                                // вычисление интеграла
                                sum += (exp(sinx[i] * cosy[j]) + 1) * value;

                        }
                }
        }
}
…
*res = sum;
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 0.267862

Численное вычисление интеграла. Параллельная программа (вариант 3)

```
        // проход по всем блокам
#pragma omp parallel private(ii,jj,i,j,x,y) reduction(+:sum)
{
        double *sinx = new double [BLOCK_SIZE];
        double *cosy = new double [BLOCK_SIZE];
        #pragma omp for nowait
        for(ii = 0; ii < n1; ii += BLOCK_SIZE) { // вычисление значений sin(x
          for(i = 0; i < BLOCK_SIZE; i++) {
            x = a1 + i * h + ii + h / 2;
            sinx[i] = sin(x * PI);
          }
          for(jj = 0; jj < n2; jj += BLOCK_SIZE) { // вычисление значений cos(y * pi)
            for(j = 0; j < BLOCK_SIZE; j++) {
              y = a2 + j * h + jj + h / 2;
              cosy[j] = cos(y * PI);
            }
            for(i = 0; i < BLOCK_SIZE; i++) {
                for(j = 0; j < BLOCK_SIZE; j++) {
                        // вычисление интеграла
                        sum += (exp(sinx[i] * cosy[j]) + 1) * val;
                }
            }
          }
        }
        delete [] sinx;
        delete [] cosy;
}
*res = sum;
```

Intel Core i7-3770 CPU@3.40GHz
4 cores x 2 threads
h=0.001
execution time: 0.265425

Спасибо за внимание!

Вопросы?

Контакты

- **Бахтин В.А.**, кандидат физ.-мат. наук, заведующий сектором, Институт прикладной математики им. М.В.Келдыша РАН

bakhtin@keldysh.ru