

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

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

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

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);  
    }  
}
```



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

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_поток_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  
нитеями 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.

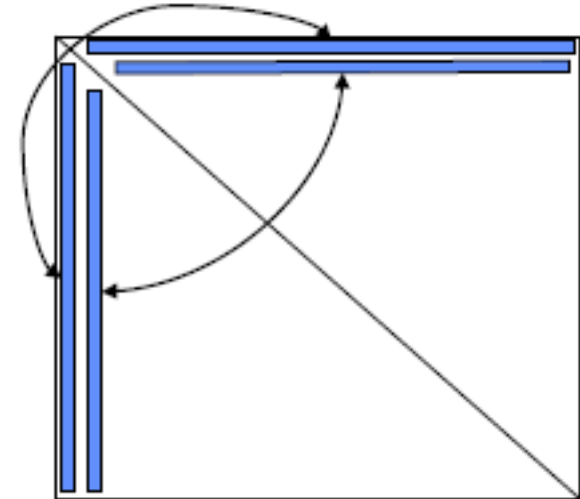
На этапе компиляции программы или во время ее выполнения определяется оптимальный способ распределения.

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];
  }
```

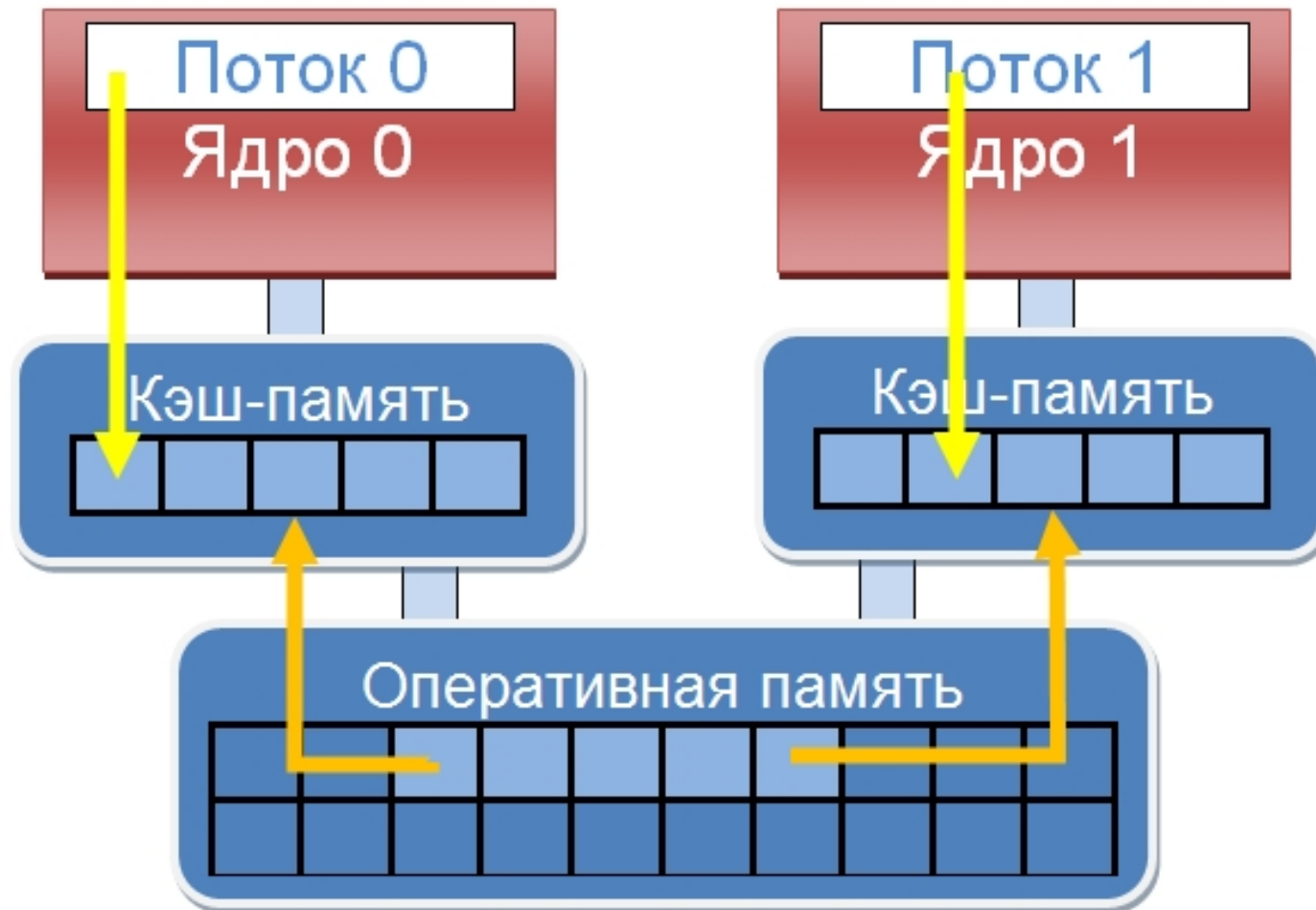
Локализация данных

```
#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
```

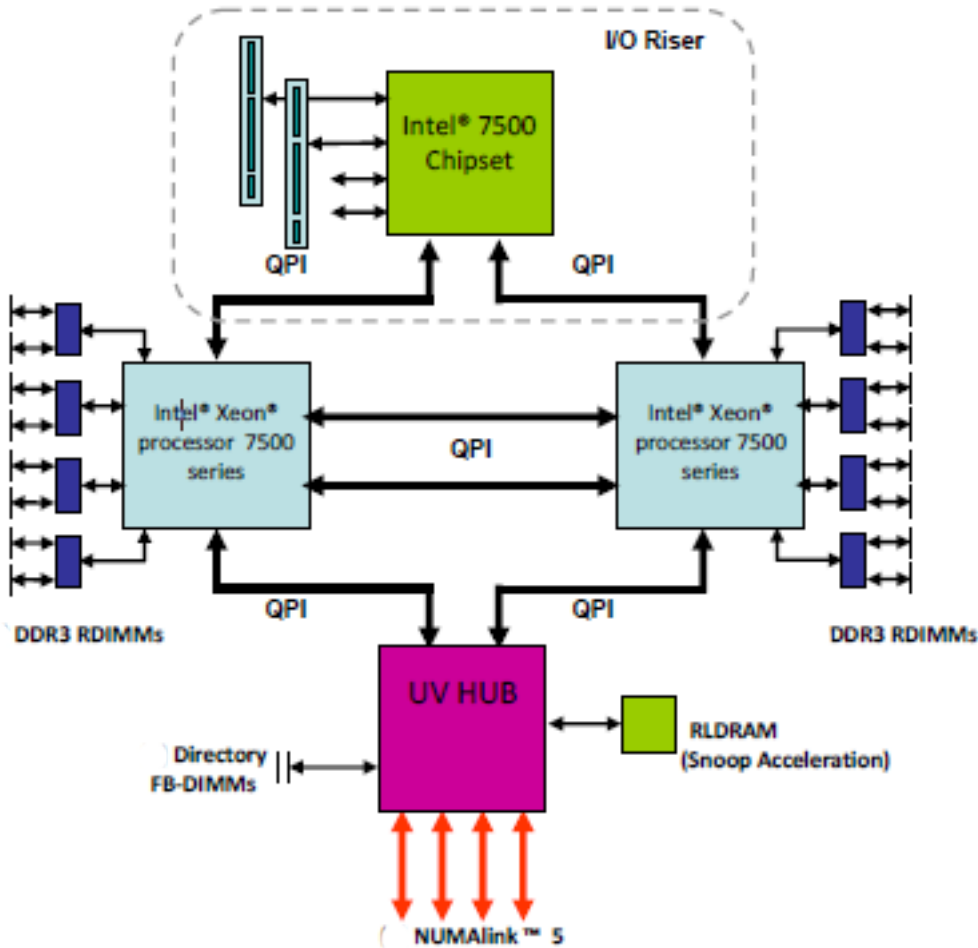
```
setenv SUNW_MP_THR_IDLE SLEEP
```

```
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 (UltraViolet) 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 algorithm
      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`

Tuning Profiler

- Test
 - TP: test.exe , Open
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]

Reference Run: A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009]

Configure Intel® 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

Buttons: OK, Отмена, Применить, Справка

Whole Program Estimated Speedups

Processors	Speedup
1	1.0
2	~3.2

Legend: Synchronized (blue), Parallel overheads (yellow)

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.
```

Tuning Profiler

- Test
 - TP: test.exe , OpenMP®
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - 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

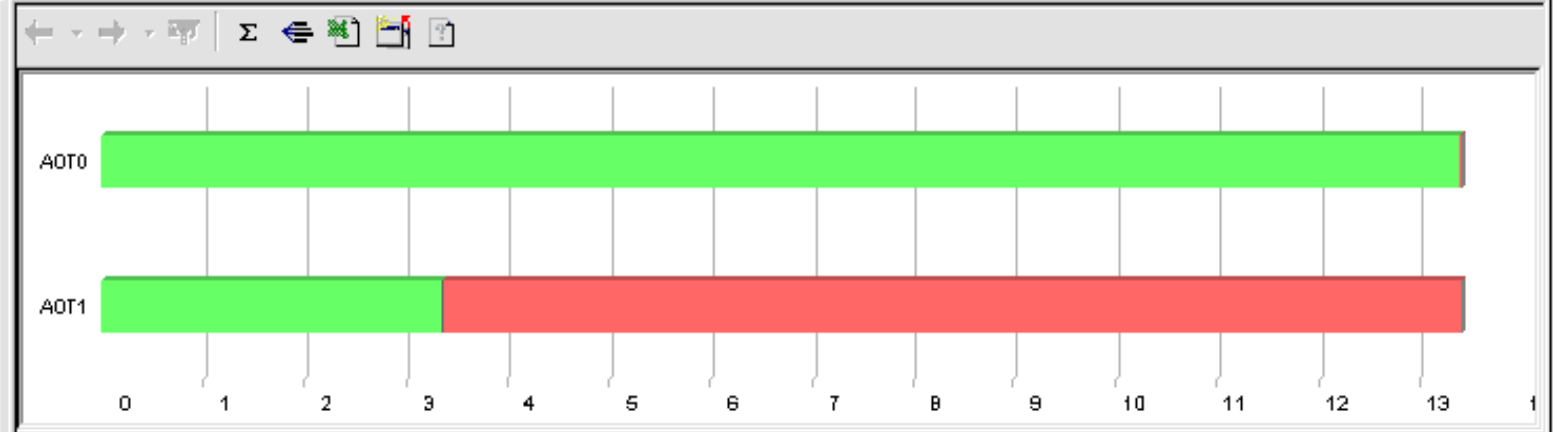
Intel® Thread Profiler OpenMP®

```

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.
    
```

Tuning Browser

- Test
 - TP: test.exe , Open
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]



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

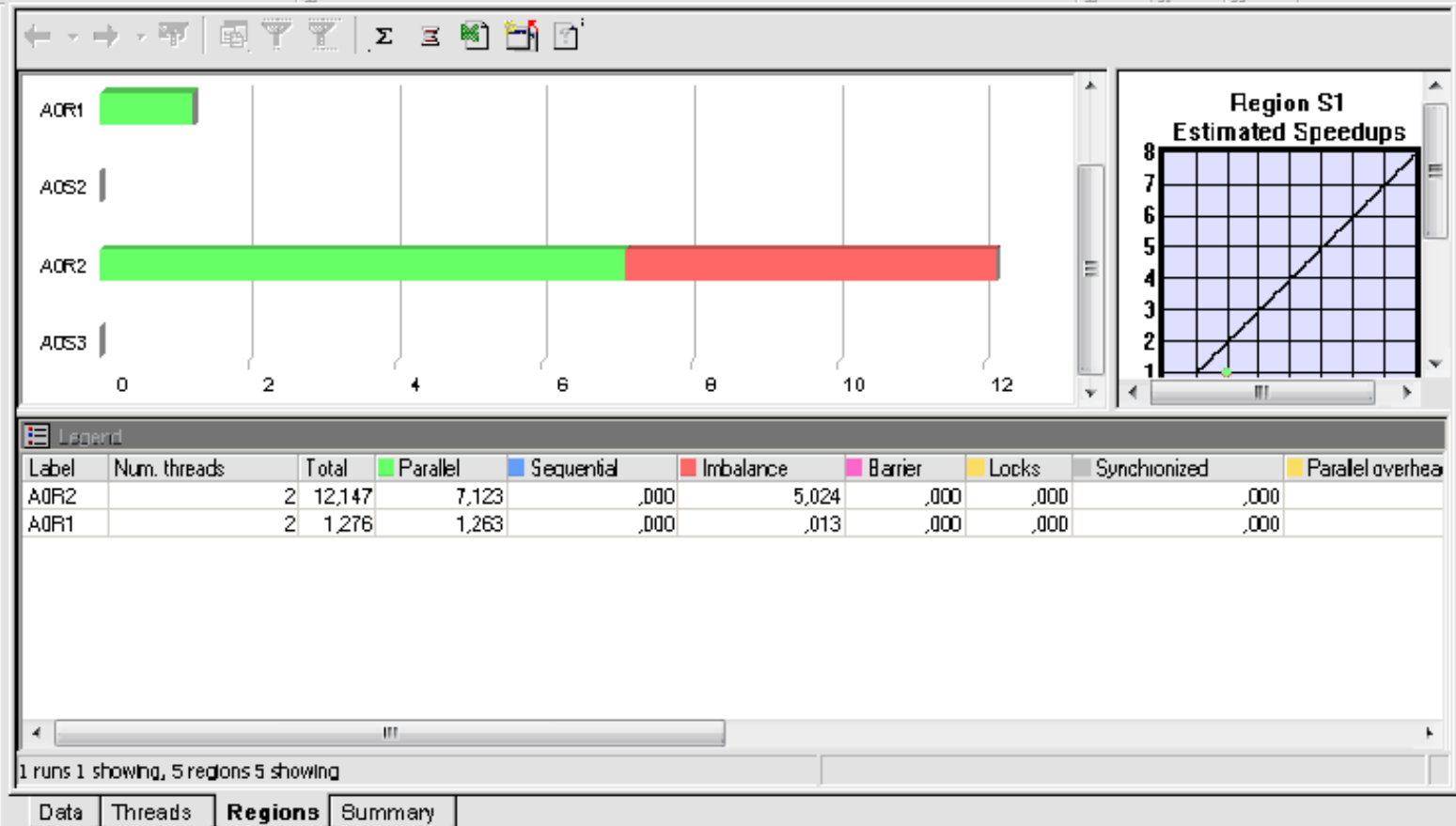
Intel® Thread Profiler OpenMP®

```

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.
    
```

Tuning Profiler

- Test
 - TP: test.exe , Open
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]



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.
    
```


Tuning Browser

- Test
 - TP: test.exe , Open
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]
 - test.exe [2 threads]

Reference Run: A0: test.exe [2 threads][Sun Oct 25 21:17:40 2009]

Whole Program Estimated Speedups

Label	#threads	Total	Parallel	Sequential	Imbalance	Barrier	Locks	Synchronized	Parallel overheads
A0	2	13,423	8,386	,000	5,036	,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.
    
```

Intel Vtune Amplifier XE 2018

The screenshot displays the Intel VTune Amplifier XE 2011 interface. The main window shows the 'Summary' view for a project named 'Jacobi' (r000lh). The 'Elapsed Time' is 16.115s. The 'CPU Time' is 55.332s, 'Instructions Retired' is 121,770,000,000, and the 'CPI Rate' is 1.253. A note indicates that the CPI rate may be too high due to memory stalls or instruction starvation. The 'Top Hotspots' section lists the most active functions, with 'main\$omp\$parallel@29' having the highest CPU time at 48.270s. The 'Collection and Platform Info' section provides details about the collection, including the command line, frequency (2.666 GHz), logical CPU count (4), user name, operating system (Windows), and computer name (KIAM-DVM).

Elapsed Time: 16.115s

- CPU Time: 55.332s
- Instructions Retired: 121,770,000,000
- CPI Rate: 1.253
- Paused Time: 0s

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify what is causing high CPI.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
main\$omp\$parallel@29	48.270s
_kmp_fork_call	2.307s
vcomp_for_static_simple_init	1.085s
_kmpc_omp_taskyield	0.638s
ZwYieldExecution	0.585s
[Others]	2.448s

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

- Command Line: C:\Lecture\OpenMP2011\Task\Jac\Release\Jac.exe
- Frequency: 2.666 GHz
- Logical CPU Count: 4
- User Name:
- Operating System: Windows
- Computer Name: KIAM-DVM

Intel Vtune Amplifier XE 2018

The screenshot shows the Intel VTune Amplifier XE 2011 interface. The main window displays a performance analysis of a Jacobi program. The table below shows the source code lines and their corresponding performance metrics.

Line	Source	CPU Time *	Instructions Retired
37	for(i=1; i<=L-2; i++)	0.011s	4,000,000
38	for(j=1; j<=L-2; j++)	9.359s	12,434,000,000
39	{		
40	localeps = Max(fabs(B[i][j]-A[i][j]), localeps);	11.338s	20,344,000,000
41	A[i][j] = B[i][j];	1.368s	5,600,000,000
42	}		
43	#pragma omp critical		
44	eps = Max(eps, localeps);		
45	#pragma omp for collapse (2)		
46	for(i=1; i<=L-2; i++)	3.247s	13,024,000,000
47	for(j=1; j<=L-2; j++)	5.145s	19,222,000,000
48	B[i][j] = (A[i-1][j]+A[i+1][j]+	17.800s	37,772,000,000
49	A[i][j-1]+A[i][j+1])/4.;		
50	#pragma omp master		
51	printf("it=%4i eps=%f\n", it, eps);		
52	if (eps < MAXEPS) break;		
53	}		
54	//f=fopen("jacobi.dat", "wb");		
55	//fwrite(B, sizeof(double), L*L, f);		
56	return 0;		
57	}		

Selected 1 row(s): 17.800s

No filters are applied. Module: [All] Thread: [All] Process: [All]

Intel Vtune Amplifier XE 2018

The screenshot displays the Intel VTune Amplifier XE 2011 interface. The main window shows a table of function performance data for the analysis target 'jac.cpp'. The table columns are: Function / Thread / H/W Context, CPU Time, Instructions Retired, CPI Rate, Module, and Function (Full). The selected row is 'main\$omp\$parallel@29' with a CPU time of 48.270s and 108,400,000 instructions retired.

Function / Thread / H/W Context	CPU Time	Instructions Retired	CPI Rate	Module	Function (Full)
main\$omp\$parallel@29	48.270s	108,400,000,000	1.232	jac.exe	main\$omp\$parallel@29
Thread (0x1548)	12.131s	27,082,000,000	1.240	jac.exe	main\$omp\$parallel@29
Thread (0x19b0)	12.052s	27,082,000,000	1.233	jac.exe	main\$omp\$parallel@29
Thread (0x1ad8)	12.050s	27,106,000,000	1.233	jac.exe	main\$omp\$parallel@29
Thread (0x1e84)	12.037s	27,130,000,000	1.224	jac.exe	main\$omp\$parallel@29
_kmp_fork_call	2.307s	5,008,000,000	1.227	libiomp5md.dll	_kmp_fork_call
vcomp_for_static_simple_init	1.085s	2,382,000,000	1.199	libiomp5md.dll	vcomp_for_static_simple_init
_kmpc_omp_taskyield	0.638s	2,054,000,000	0.888	libiomp5md.dll	_kmpc_omp_taskyield
ZwYieldExecution	0.585s	742,000,000	2.070	ntkrnlpa.exe	ZwYieldExecution
KiFastSystemCallRet	0.329s	498,000,000	1.639	ntdll.dll	KiFastSystemCallRet
_kmp_invoke_microtask	0.290s	912,000,000	0.864	libiomp5md.dll	_kmp_invoke_microtask

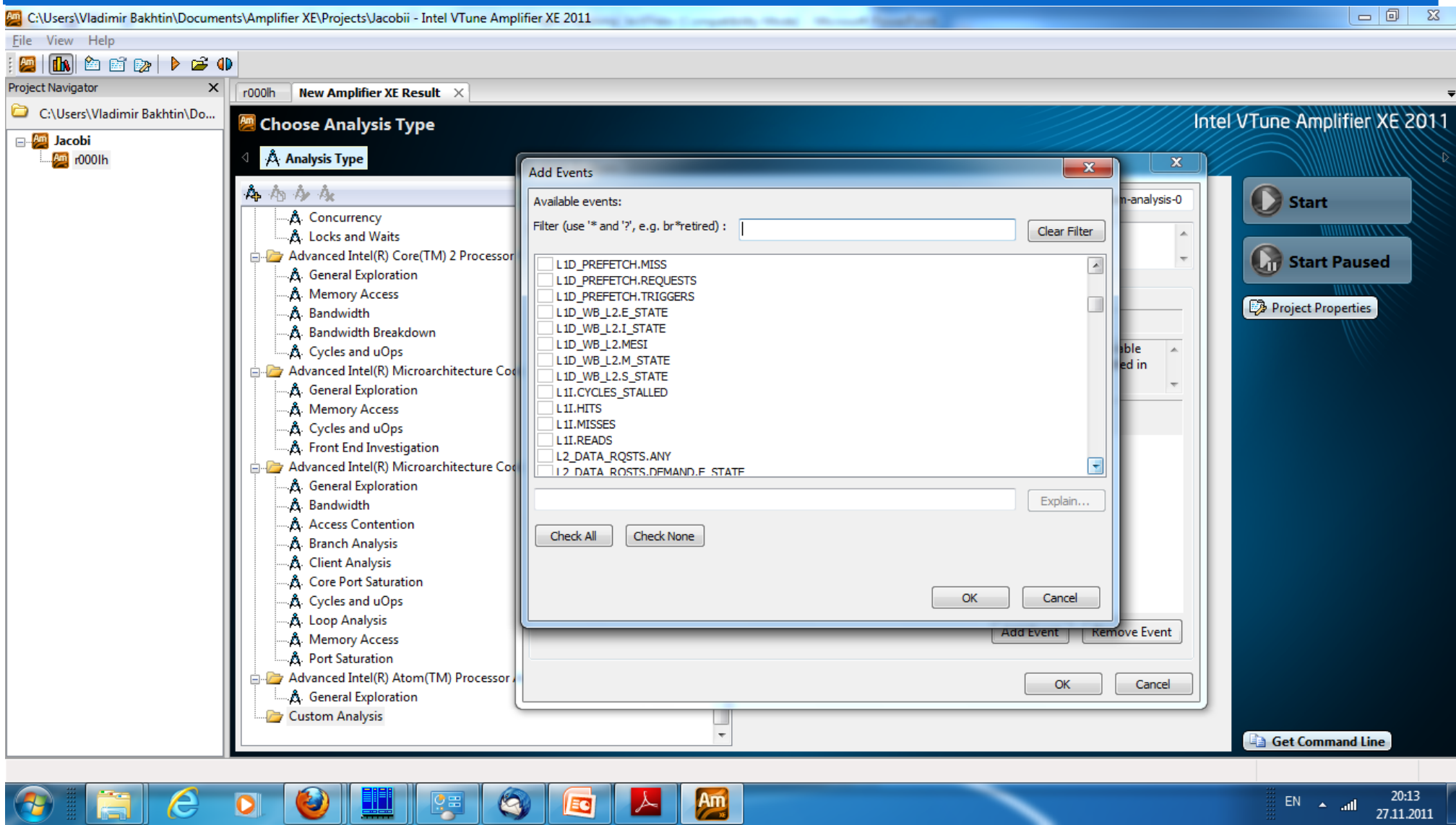
Below the table is a timeline view showing the execution of threads over time. The threads listed are Thread (0x19b0), Thread (0x1cfc), Thread (0x1ad8), Thread (0x1548), and Thread (0x1e84). The CPU Time view shows the overall system activity. The timeline is controlled by a play/pause button and a time scale from 1s to 16s.

At the bottom, there are filter controls: 'No filters are applied.', 'Module: [All]', 'Thread: [All]', and 'Process: [All]'.

Intel Vtune Amplifier XE 2018

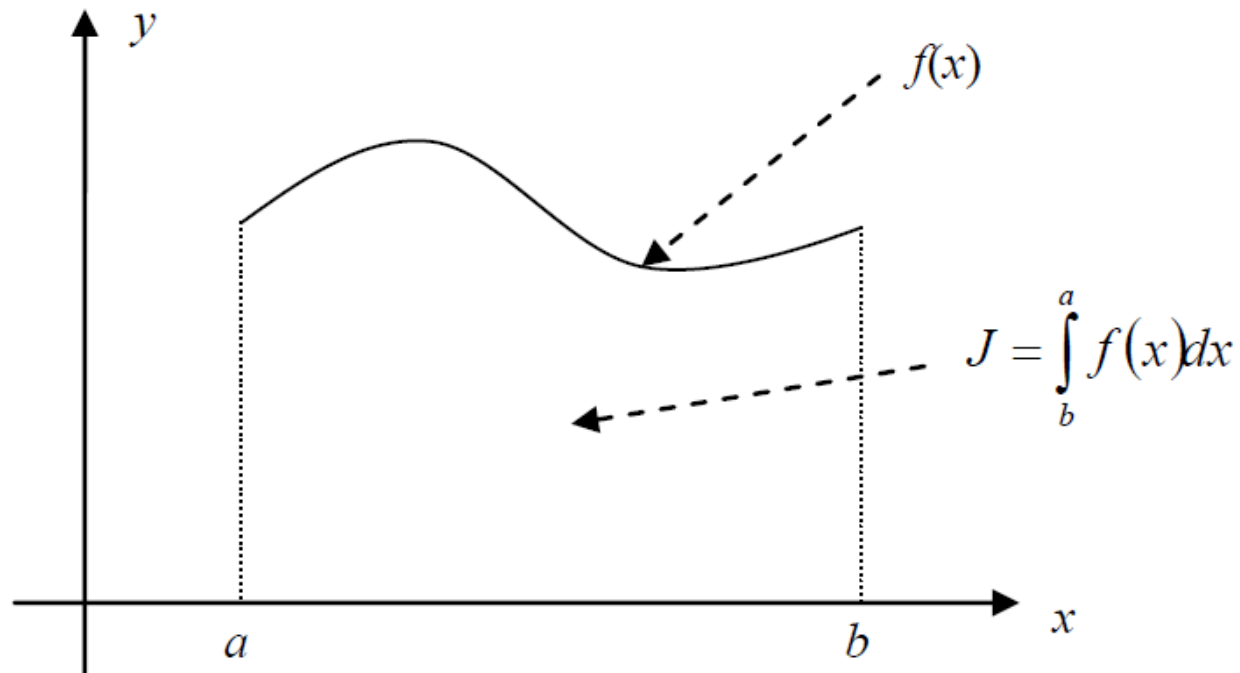
The screenshot displays the Intel VTune Amplifier XE 2011 software interface. The main window is titled "Choose Analysis Type" and shows a tree view of analysis types. The "Hotspots" analysis type is selected. The right pane provides details for the "Hotspots" analysis, including a warning that "Highly accurate CPU time collection is disabled for this analysis" and a list of configuration options such as "CPU sampling interval, ms: 10" and "Detect context switches: Yes". The interface also features a "Start" button and a "Start Paused" button on the right side.

Intel Vtune Amplifier XE 2018



Пример. Вычисление определенного интеграла

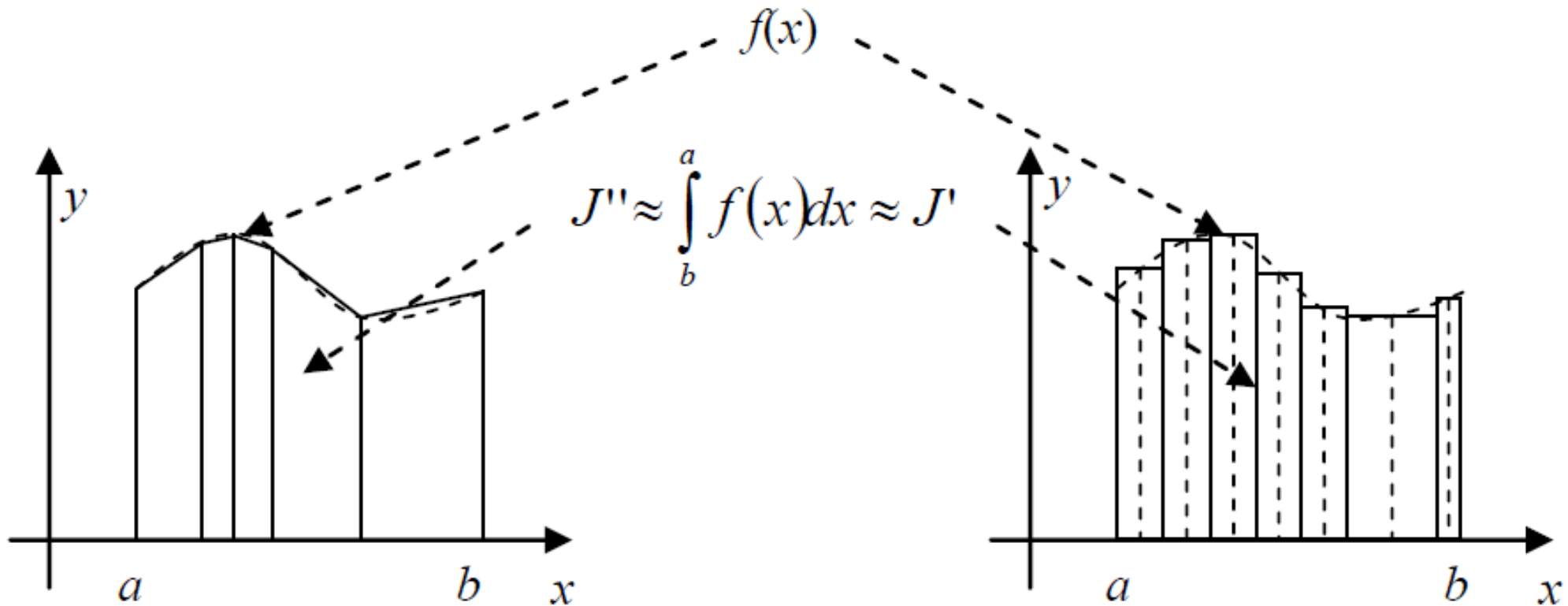
Пусть на отрезке $[a, b]$ задана непрерывная неотрицательная функция. В этом случае значение определенного интеграла от $f(x)$ на отрезке $[a, b]$ совпадает с площадью фигуры, ограниченной графиком функции, осью Ox и прямыми $x = a$, $x = b$.



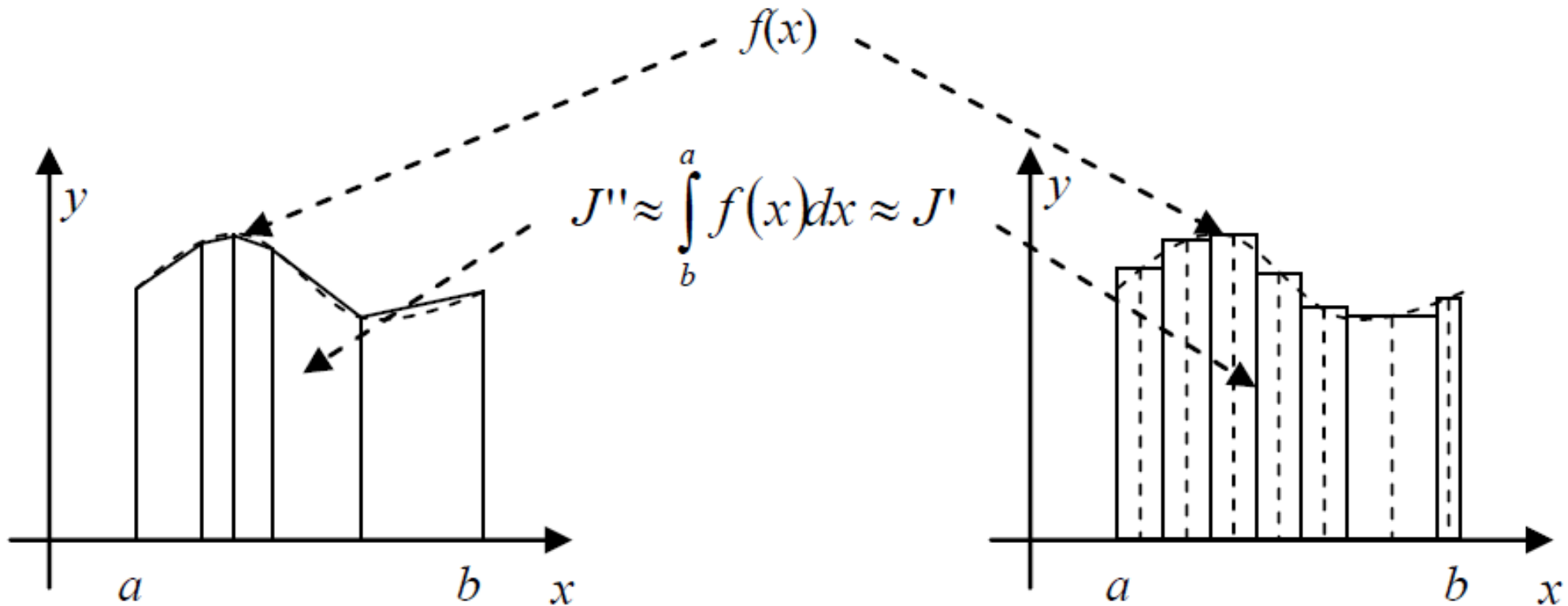
Спасибо Е.А. Козинову и А.В. Сысоеву:

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

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

где N -количество отрезков интегрирования по оси x ,
а $h=(b-a)/N$

$$\left\{ \begin{array}{l} J = \int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x, y) dx dy \approx h_1 h_2 \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f(x_i, y_j) \\ x_i = a_1 + ih_1 + \frac{h_1}{2}, y_j = a_2 + jh_2 + \frac{h_2}{2} \end{array} \right. ,$$

где M -количество отрезков интегрирования по оси y ,
 $h_1=(b_1-a_1)/N$,
 $h_2=(b_2-a_2)/M$

Численное вычисление интеграла методом параллелепипедов

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

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 программа (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 * 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: 0.681427

Численное вычисление интеграла методом параллелепипедов. Параллельная 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;
    // 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 * 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: 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;
            }
        }
    }
}
}
```

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

...

*res = sum, Москва, 2021 г.

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