Параллельные вычисления (часть 2)
Стандарт OpenMP

Михаил Георгиевич Курносов

Email: mkurnosov@gmail.com

WWW: http://www.mkurnosov.net

Курс «Параллельные и распределённые вычисления»

Школа анализа данных Яндекс (Новосибирск)

Весенний семестр, 2015

Популярный инструментарий параллельного программирования

- **Многопроцессорные системы с общей памятью (SMP, NUMA)**
  - ❑ OpenMP
- **Системы с распределенной памятью (вычислительные кластеры)**
  - ❑ MPI (Message Passing Interface)
- **Гибридные ВС**
  - ❑ MPI + OpenMP
  - ❑ MPI + CUDA/OpenCL/OpenACC/OpenMP 4.0

Инструментарий параллельного программирования

- **Многопроцессорные системы с общей памятью (SMP, NUMA)**
  - **Потоки ОС:** POSIX Threads, Window Threads
  - **Языки и библиотеки:** Cilk++ (Intel Cilk Plus), Intel TBB, .NET Task Parallel Library, Parallel Patterns Library, C++11 Threads, C11 Threads, Java Threads, Erlang Threads
  - **Task layers (легковесные задачи):** Qthread, MassiveThreads
- **Системы с распределенной памятью (вычислительные кластеры)**
  - MPI, Shmem, IBM X10, Cray Chapel, Unified Parallel C, Global Arrays
  - MapReduce, Google Cloud Dataflow, Microsoft Dryad, Spark

Стандарт OpenMP

- **OpenMP (Open Multi-Processing)** – стандарт, определяющий набор директив компилятора, библиотечных функций и переменных среды окружения для создания многопоточных программ
- Поддерживаются интерфейсы с языками C/C++ и Fortran
- Требуется поддержка со стороны компилятора
- Разрабатывается в рамках OpenMP Architecture Review Board с 1997 года
  - http://www.openmp.org
  - http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  - OpenMP 2.5 (2005), OpenMP 3.0 (2008), OpenMP 3.1 (2011), **OpenMP 4.0 (2013)**

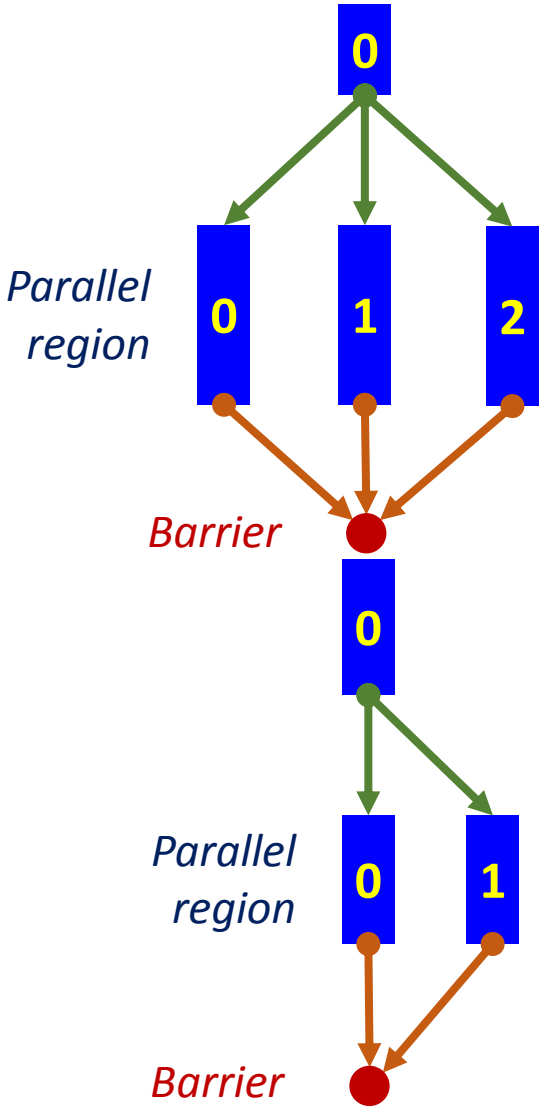
Поддержка компиляторами

Compiler	Information
GNU GCC	**Option: –fopenmp** gcc 4.2 – OpenMP 2.5, gcc 4.4 – OpenMP 3.0, gcc 4.7 – OpenMP 3.1 gcc 4.9 – OpenMP 4.0
Clang (LLVM)	OpenMP 3.1 clang + Intel OpenMP RTL http://clang-omp.github.io/
Intel C/C++, Fortran	OpenMP 4.0 Option: –Qopenmp, –openmp
Oracle Solaris Studio C/C++/Fortran	OpenMP 4.0 Option: –xopenmp
Microsoft Visual Studio C++	Option: /openmp OpenMP 2.0 only
Other compilers: IBM XL, PathScale, PGI, Absoft Pro, …	

http://openmp.org/wp/openmp-compilers/

Модель выполнения OpenMP-программы

- **Динамическое управление потоками в модели Fork-Join**:
  - ✓ **Fork** – порождение нового потока
  - ✓ **Join** – ожидание завершения потока (объединение потоков управления)
- OpenMP-программа – совокупность последовательных участков кода (serial code) и параллельных регионов (parallel region)
- Каждый поток имеет логический номер: 0, 1, 2, …
- Главный поток (master) имеет номер 0
- Параллельные регионы могут быть вложенными



Hello, OpenMP World!

```
#include <iostream>

int main(int argc, char *argv[])
{
    #pragma omp parallel     // fork
    {
        std::cout << "Hello, OpenMP World!" << std::endl;
    }                        // join (barrier)

    return 0;
}
```

Синтаксис директив OpenMP

- **Языки C/C++**

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
#pragma omp parallel
```

- **Fortran**

```
sentinel directive-name [clause[[,] clause]...]
!$omp parallel
```

Компиляция и запуск OpenMP-программ

```
$ g++ -Wall -fopenmp -o hello ./hello.cpp
$ ./hello
Hello, OpenMP World!
Hello, OpenMP World!
Hello, OpenMP World!
Hello, OpenMP World!

$ g++ -Wall -o hello ./hello.cpp
./hello.cpp:5:0: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]

$ ./hello
Hello, OpenMP World!
```

Условная компиляция (_OPENMP)

```
#include <omp.h>

int main()
{

    #pragma omp parallel
    {
#ifdef _OPENMP
        printf("Thread %d\n", omp_get_thread_num());
        if (_OPENMP >= 201107)
            printf("OpenMP 3.1 is supported\n");
#endif
    }

    return 0;
}
```

Hello, OpenMP World! (2)

```
#include <omp.h>

int main(int argc, char **argv)
{
    int nthreads, threadid;

    #pragma omp parallel num_threads(8)                 // Параллельный регион из 8 потоков
    {
        threadid = omp_get_thread_num();                // Data race ???
        printf("%d: Hello, OpenMP World\n", threadid);

        if (threadid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
    return 0;
}
```

Hello, OpenMP World! (2)

```
#include <omp.h>

int main(int argc, char **argv)
{
    int nthreads, threadid;

    #pragma omp parallel private(threadid) num_threads(8)
    {
        threadid = omp_get_thread_num();                    // threadid – локальная копия
        printf("%d: Hello, OpenMP World\n", threadid);

        if (threadid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
    return 0;
}
```

Атрибуты видимости данных

- **shared** (list) – указанные переменные являются разделяемыми (сохраняют исходный класс памяти: auto, static, thread_local)
- **private** (list) – создает локальную переменную того же типа (automatic storage duration)
- **firstprivate** (list) – создает локальную переменную того же типа и инициализирует ее исходных значением (C++: copy assignment)
- **lastprivate** (list) – значение локальной копии копируется в исходную переменную (C++: copy assignment operator)
- **#pragma omp threadprivate** (list) – создает для каждого потока копии указанных статических переменных (static storage duration)

Количество потоков

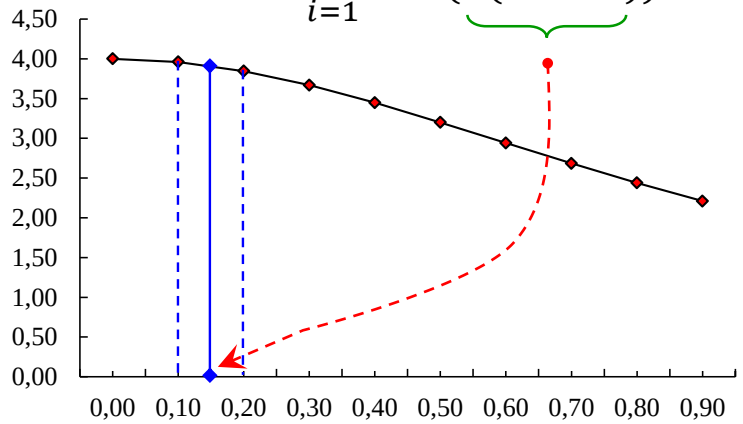
- По умолчанию обычно равно количеству логических процессоров в системе
- Может быть задано явно
  - Переменная окружения OMP_NUM_THREADS
  - Функция omp_set_num_threads()
  - Директива #pragma omp parallel ... num_threads(N)
- Может изменяться в ходе выполнения программы

Вычисление числа π

```
int main(int argc, char **argv)
{
    double PI25DT = 3.14159265358979323846264643;
    double pi, x, step, sum;
    int i, nsteps;

    nsteps = (argc > 1) ? atoi(argv[1]) : 1000000;
    step = 1.0 / (double)nsteps;

    sum = 0.0;
    for (i = 1; i <= nsteps; i++) {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }
    pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \qquad h = \frac{1}{n}$$

$$\pi \approx h \sum_{i=1}^{n} \frac{4}{1+(h(i-0.5))^2}$$



Вычисление числа π – версия 1 (SPMD)

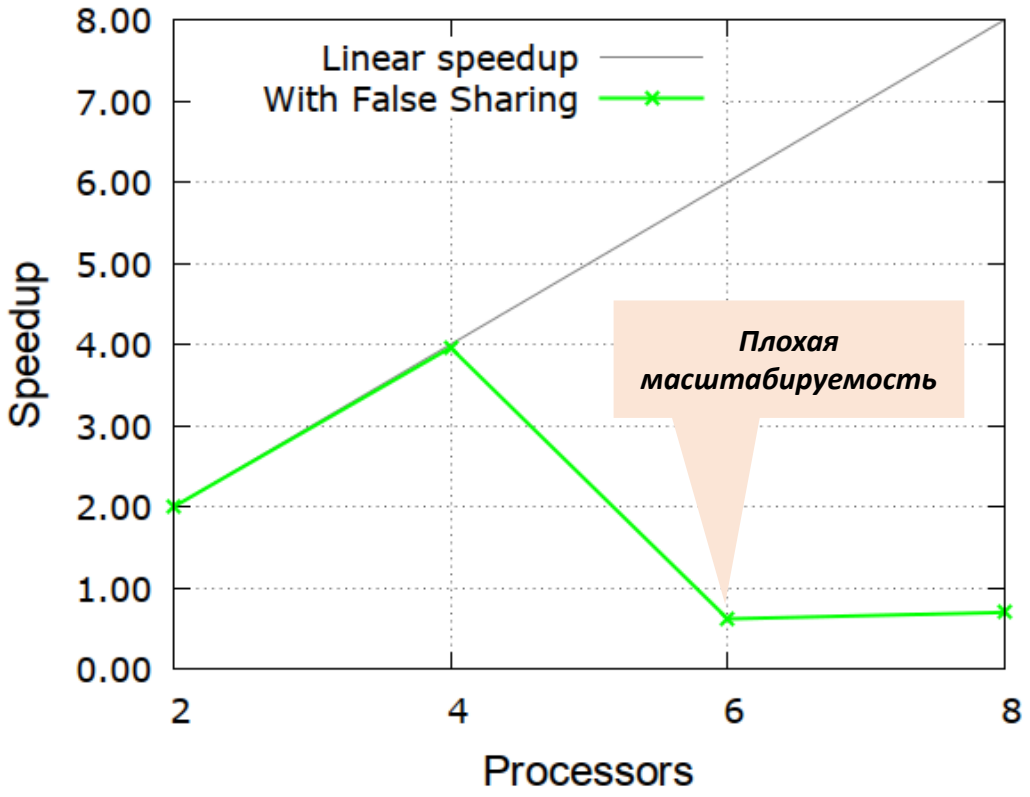
```
int main(int argc, char **argv) {
    double t = omp_get_wtime();
    double PI25DT = 3.14159265358979323846264643;
    int nsteps = (argc > 1) ? atoi(argv[1]) : 1000000;
    double step = 1.0 / (double)nsteps;

    int nthreads = omp_get_max_threads();
    double sumloc[nthreads];                    // Thread local storage – хранилище потока
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        sumloc[tid] = 0.0;
        for (int i = tid + 1; i <= nsteps; i += nthreads) {       // Циклическое распределение итераций
            double x = (i - 0.5) * step;
            sumloc[tid] += 4.0 / (1.0 + x * x);
        }
    }
    double sum = 0.0;
    for (int i = 0; i < nthreads; i++)
        sum += sumloc[i];
    double pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    printf("Elapsed time = %.6f sec.\n", omp_get_wtime() - t);
    return 0;
}
```

Вычисление числа π – версия 2 (for, nowait, critical)

```
int main(int argc, char **argv) {
    // ...
    int nthreads = omp_get_max_threads();
    double sumloc[nthreads];
    double sum = 0.0;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        sumloc[tid] = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid] += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid];
    }
    double pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    // ...
}
```

Вычисление числа π – версия 2 (for, nowait, critical)

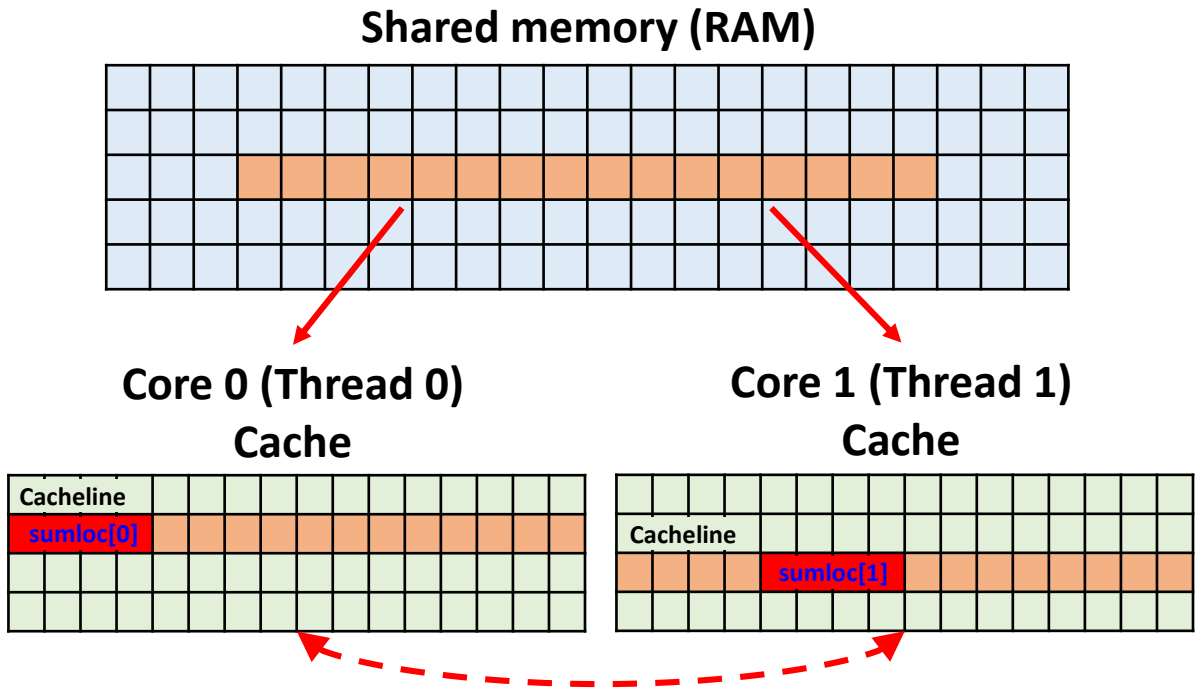
```
int main(int argc, char **argv) {
    // ...
    int nthreads = omp_get_max_threads();
    double sumloc[nthreads];
    double sum = 0.0;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        sumloc[tid] = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid] += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid];
    }
    double pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    // ...
}
```



Вычислительный узел Intel S5000VSA:
2 x Intel Quad Xeon E5420, RAM 8 GB (4 x 2GB PC-5300)

Ложное разделение данных (false sharing)

```
int main(int argc, char **argv) {
    // ...
    int nthreads = omp_get_max_threads();
    double sumloc[nthreads];
    double sum = 0.0;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        sumloc[tid] = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid] += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid];
    }
    double pi = step * sum;

    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    // ...
}
```



MESI (Intel MESIF)
cache coherency protocol
Две строки в кешах постоянно обновляются актуальными данными из памяти – кеширование “отключается”

❑ **Avoiding and Identifying False Sharing Among Threads** // http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Вычисление числа π – версия 3 (избавляемся от false sharing)

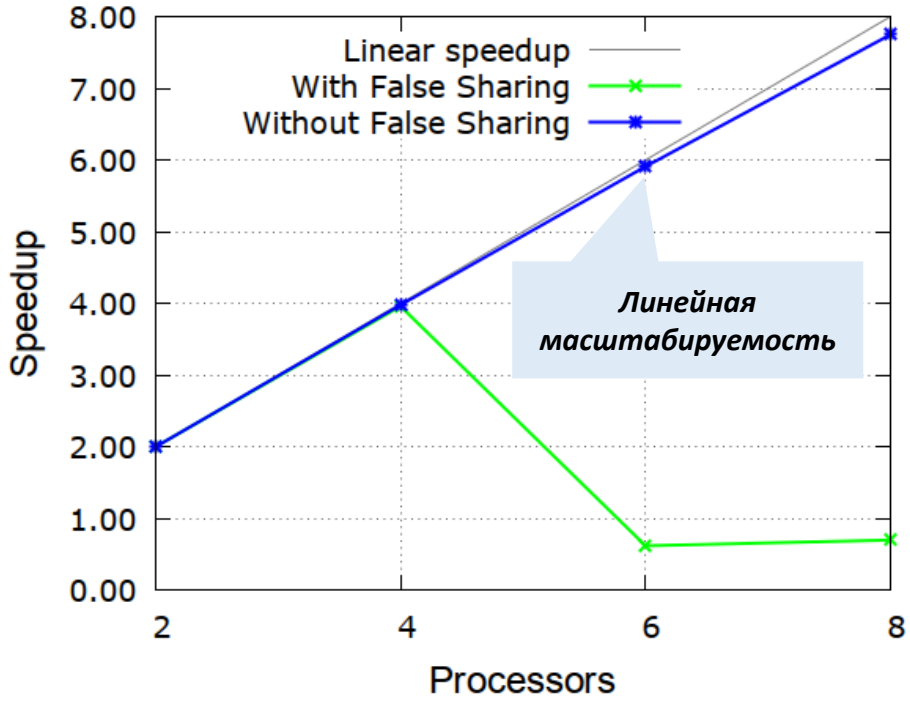
```
struct threadparams {
    double sum;
    double padding[7]; // Padding for cacheline size (64 bytes)
};

int main(int argc, char **argv) {
    // ...
    threadparams sumloc[nthreads] __attribute__ ((aligned(64)));
    // double sumloc[nthreads * 8];
    double sum = 0.0;
    #pragma omp parallel num_threads(nthreads) {
        int tid = omp_get_thread_num();
        sumloc[tid].sum = 0.0;
        #pragma omp for nowait
        for (int i = 1; i <= nsteps; i++) {
            double x = (i - 0.5) * step;
            sumloc[tid].sum += 4.0 / (1.0 + x * x);
        }
        #pragma omp critical
        sum += sumloc[tid].sum;
    }
    // ...
}
```

Вычисление числа π – версия 3.1 (избавляемся от false sharing)

```
// ...
double sum = 0.0;
#pragma omp parallel num_threads(nthreads)
{
    double sumloc = 0.0;        // Избавились от массива в памяти
    #pragma omp for nowait
    for (int i = 1; i <= nsteps; i++) {
        double x = (i - 0.5) * step;
        sumloc += 4.0 / (1.0 + x * x);
    }
    #pragma omp critical
    sum += sumloc;
}
double pi = step * sum;
// ...
```

Вычисление числа π – версия 3.1 (избавляемся от false sharing)

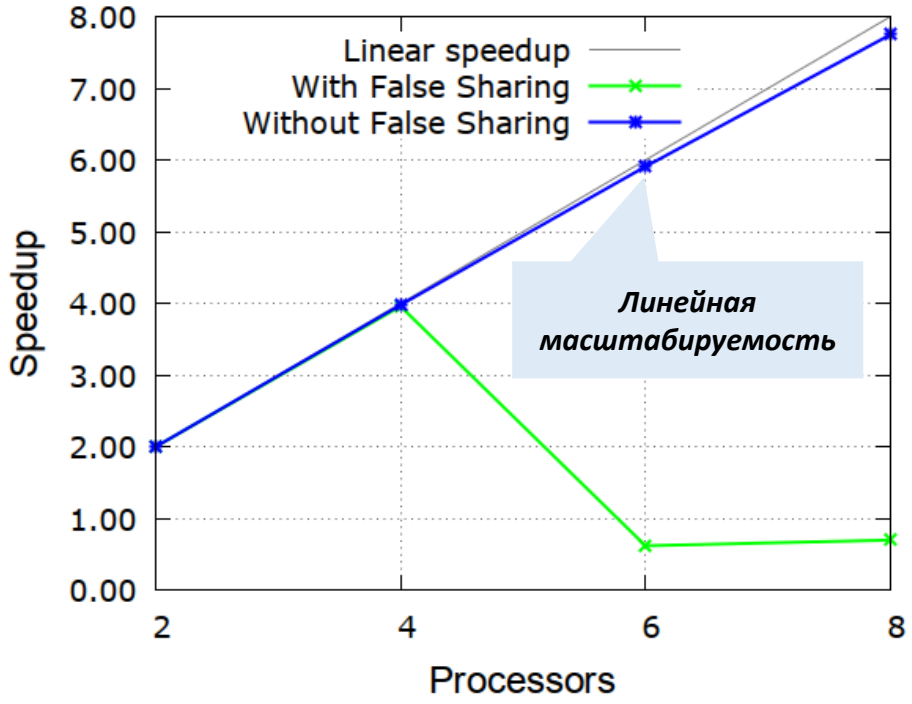
```
// ...
double sum = 0.0;
#pragma omp parallel num_threads(nthreads)
{
    double sumloc = 0.0;
    #pragma omp for nowait
    for (int i = 1; i <= nsteps; i++) {
        double x = (i - 0.5) * step;
        sumloc += 4.0 / (1.0 + x * x);
    }
    #pragma omp critical
    sum += sumloc;
}
double pi = step * sum;
// ...
```



Вычислительный узел Intel S5000VSA:
2 x Intel Quad Xeon E5420, RAM 8 GB (4 x 2GB PC-5300)

Вычисление числа π – версия 3.2 (#pragma omp atomic)

```
// ...
double sum = 0.0;
#pragma omp parallel num_threads(nthreads)
{
    double sumloc = 0.0;
    #pragma omp for nowait
    for (int i = 1; i <= nsteps; i++) {
        double x = (i - 0.5) * step;
        sumloc += 4.0 / (1.0 + x * x);
    }
    #pragma omp atomic
    sum += sumloc;
}
double pi = step * sum;
// ...
```



Вычислительный узел Intel S5000VSA:
2 x Intel Quad Xeon E5420, RAM 8 GB (4 x 2GB PC-5300)

Атомарные операции (#pragma omp atomic)

```
#pragma omp atomic
x = x binop expr;
```

- Операции: x++, x = x + y, x = x - y, x = x * y, x = x / y, ...

- **Атомарная операция** (atomic operation) — это инструкция процессора, в процессе выполнения которой операнд в памяти блокируются для других потоков

- **Intel 64 locked atomic operation** (BTS, XADD, CMPXCHG, ADD, …)
  - ❑ Integer operand – one atomic operation
  - ❑ Floating point operand – loop with CAS

Атомарные операции: float & int

```
int counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += i;
}
```

```
lock addl %ecx, (%rsi)
```

```
double counter = 0;
#pragma omp parallel for
for (int i = 0; i < 1000; i++) {
    #pragma omp atomic
    counter += static_cast<double>(i);
}
```

```
.L8:  movq          %rax, %rdx
.L4:  movq          %rdx, 8(%rsp)
      movq          %rdx, %rax
      movsd 8(%rsp), %xmm1
      addsd %xmm0, %xmm1
      movq          %xmm1, %rsi
      lock cmpxchgq %rsi, (%rcx)
      cmpq          %rax, %rdx
      jne .L8
```

Loop!

Цикл выполняется пока ячейка успешно не обновится

Вычисление числа π – версия 4 (for + reduction)

```
int main(int argc, char **argv)
{
    double t = omp_get_wtime();
    double PI25DT = 3.14159265358979323846264643;
    int nsteps = (argc > 1) ? atoi(argv[1]) : 1000000000;
    double step = 1.0 / (double)nsteps;

    int nthreads = omp_get_max_threads();
    double sum = 0.0;

    #pragma omp parallel for reduction (+:sum) num_threads(nthreads)
    for (int i = 1; i <= nsteps; i++) {
        double x = (i - 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    double pi = step * sum;
    t = omp_get_wtime() - t;
    printf("PI is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    printf("(nsteps = %d, step = %f)\n", nsteps, step);
    printf("Elapsed time = %.6f sec.\n", t);
    return 0;
}
```

Допустимые операции директивы reductioin

Op	Initializer	Combiner
+	omp_priv = 0	omp_out += omp_in
*	omp_priv = 1	omp_out *= omp_in
-	omp_priv = 0	omp_out += omp_in
&	omp_priv = ~0	omp_out &= omp_in
|	omp_priv = 0	omp_out |= omp_in
^	omp_priv = 0	omp_out ^= omp_in
&&	omp_priv = 1	omp_out = omp_in && omp_out
||	omp_priv = 0	omp_out = omp_in || omp_out
max	omp_priv = *Min*	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = *Max*	omp_out = omp_in < omp_out ? omp_in : omp_out

- В OpenMP 4.0 допустимо создание своих операций редукции

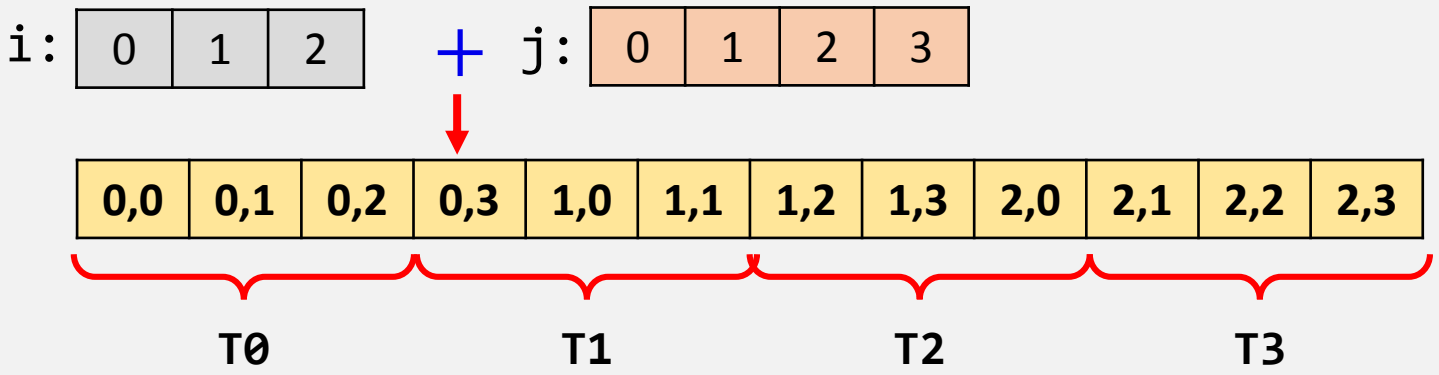
Объединение вложенных циклов

```
// N < количество потоков; как эффективно загрузить потоки?
for (j = 0; j < N; j++) {
    for (i = 0; i < M; i++) {
        A[i][j] = work(i, j);
    }
}
```

```
// Объединяем циклы
#pragma omp parallel for private(j, i)
for (ij = 0; ij < N * M; ij++) {
    j = ij / M;
    i = ij % M;
    A[i][j] = work(i, j);
}
```

Объединение пространств итераций циклов

```
#define N 3
#define M 4

#pragma omp parallel
{
    #pragma omp for collapse(2)
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++)
            printf("Thread %d i = %d\n", omp_get_thread_num(), i);
    }
}
```



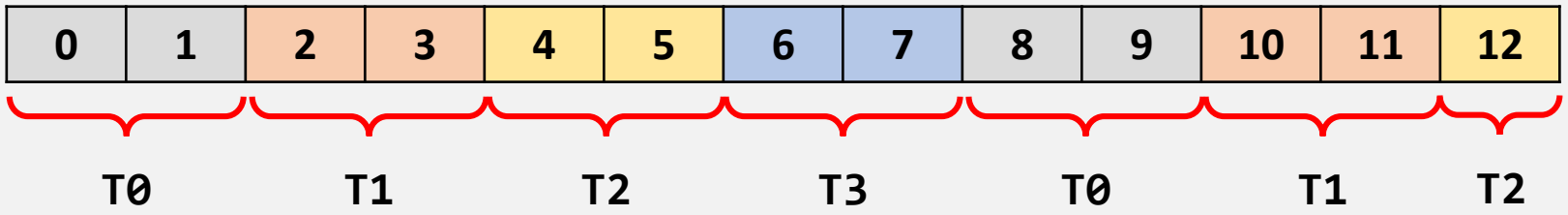
Директива **collapse(n)** объединяет пространства итераций *n* циклов

Распределение итераций цикла for между потоками

```
#pragma omp for schedule(static, 1)
```

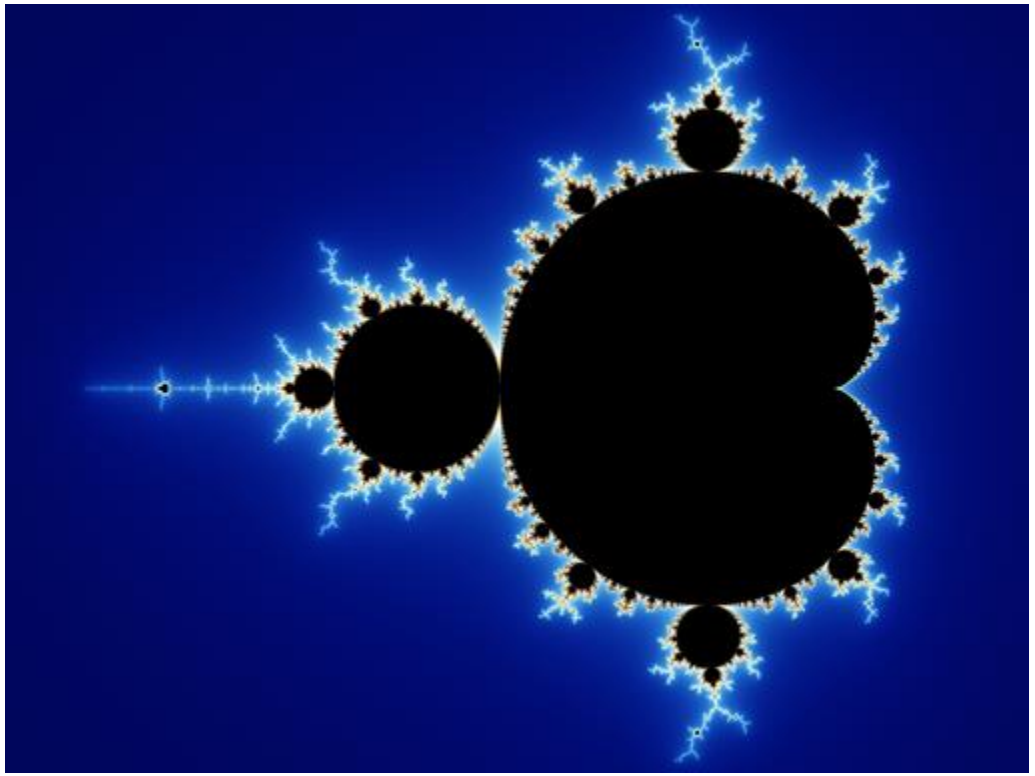
- Атрибут schedule(type, chunk)
  - **static** – статическое циклическое распределение блоками по chunk итераций (по принципу round-robin, детерминированное)
  - **dynamic** – динамическое распределение блоками по chunk-итераций (по принципу master-worker)
  - **guided** – динамическое распределение с уменьшающимися порциями
  - **runtime** – тип распределения берется из переменной среды окружения OMP_SCHEDULE (export OMP_SCHEDULE="static,1")

#pragma omp for schedule(static, 2)



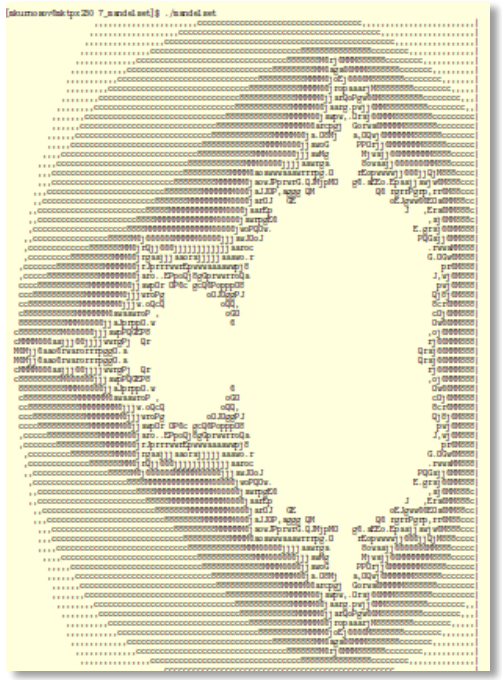
Пример: Множество Мандельброта

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$



Множество Мандельброта – версия 0

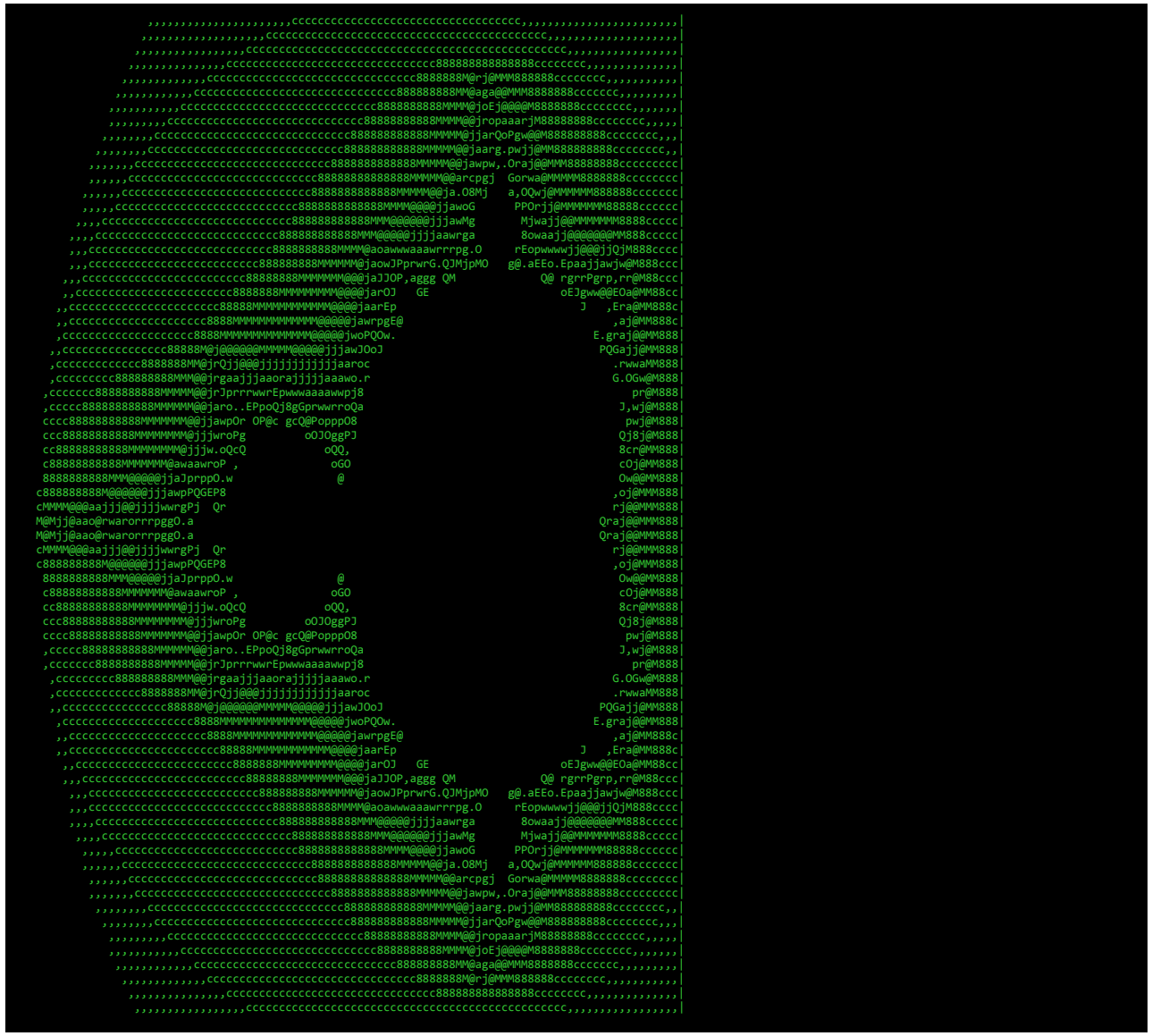
```
int MandelbrotCalculate(complex c, int maxiter) {
    complex z = c;
    int n = 0;
    for (; n < maxiter; n++) {
        if (std::abs(z) >= 2.0) break;
        z = z * z + c;
    }
    return n;
}

int main() {
    // ...
    for (int pix = 0; pix < npixels; ++pix) {
        const int x = pix % width, y = pix / width;
        complex c = begin + complex(x * span.real() / (width + 1.0),
                                    y * span.imag() / (height + 1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter) n = 0;
        pixels[pix] = n;
    }
    // ...
}
```



Множество Мандельброта – версия 1 (static)

```
int main() {
    // ...
    #pragma omp parallel for
    for (int pix = 0; pix < npixels; ++pix) {
        const int x = pix % width, y = pix / width;
        complex c = begin + complex(x * span.real() / (width + 1.0),
                                    y * span.imag() / (height + 1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter) n = 0;
        pixels[pix] = n;
    }
    // ...
}
```

Множество Мандельброта – версия 1 (static)



```
$ ./mandelset

Time: 17.453450 sec.
Thread 0 time: 0.900164 sec.
Thread 1 time: 15.925677 sec.
Thread 2 time: 17.450009 sec.
Thread 3 time: 2.208047 sec.
```

Load imbalance!

Множество Мандельброта – версия 2 (dynamic)

```
int main() {
    // ...
    #pragma omp parallel for schedule(dynamic, 16)
    for (int pix = 0; pix < npixels; ++pix) {
        const int x = pix % width, y = pix / width;
        complex c = begin + complex(x * span.real() / (width + 1.0),
                                    y * span.imag() / (height + 1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter) n = 0;
        pixels[pix] = n;
    }
    // ...
}
```

- **Почему (dynamic, 16)?**
- Каждый поток будет иметь 16 элементов массива pixels (64 байта)

```
Time: 11.857098 sec.
Thread 0 time: 11.854174 sec.
Thread 1 time: 11.842355 sec.
Thread 2 time: 11.842731 sec.
Thread 3 time: 11.846665 sec.
```

Множество Мандельброта – версия 1.1 (static, 1)

```
int main() {
    // ...
    #pragma omp parallel for schedule(static, 1)
    for (int pix = 0; pix < npixels; ++pix) {
        const int x = pix % width, y = pix / width;
        complex c = begin + complex(x * span.real() / (width + 1.0),
                                    y * span.imag() / (height + 1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter) n = 0;
        pixels[pix] = n;
    }
    // ...
}
```

```
Time: 12.110358 sec.
Thread 0 time: 11.473495 sec.
Thread 1 time: 12.106071 sec.
Thread 2 time: 12.090148 sec.
Thread 3 time: 11.732312 sec.
```

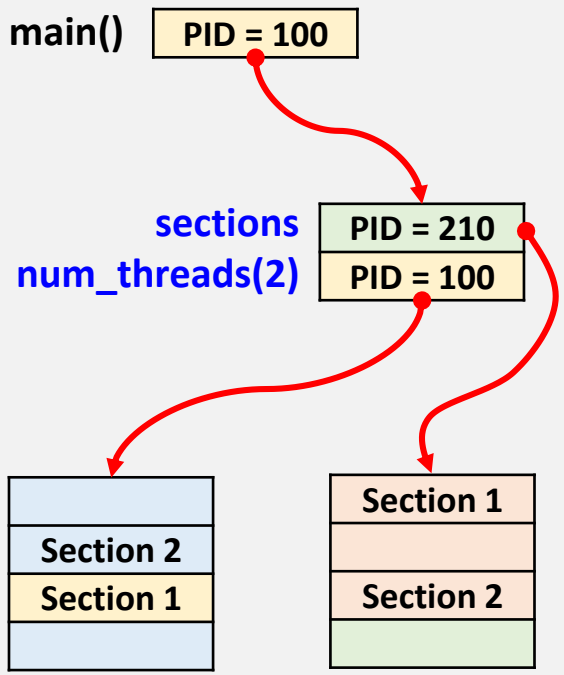
Множество Мандельброта – версия 3 (ordered)

```
int main() {
    // ...
    #pragma omp parallel for ordered schedule(dynamic) nowait
    for (int pix = 0; pix < npixels; ++pix) {
        const int x = pix % width, y = pix / width;
        complex c = begin + complex(x * span.real() / (width + 1.0),
                                    y * span.imag() / (height + 1.0));
        int n = MandelbrotCalculate(c, maxiter);
        if (n == maxiter) n = 0;
        #pragma omp ordered  // Ждем пока выполнятся итерации < pix
        {
            char c = ' ';
            if (n > 0) c = charset[n % (sizeof(charset) - 1)];
            std::putchar(c);
            if (x + 1 == width)
                std::puts("|");
        }
    }
    // ...
}
```

```
Time: 12.967938 sec.
Thread 0 time: 12.964203 sec.
Thread 1 time: 12.955229 sec.
Thread 2 time: 12.964203 sec.
Thread 3 time: 12.950563 sec.
```

Вложенные параллельные регионы (Nested parallelism)

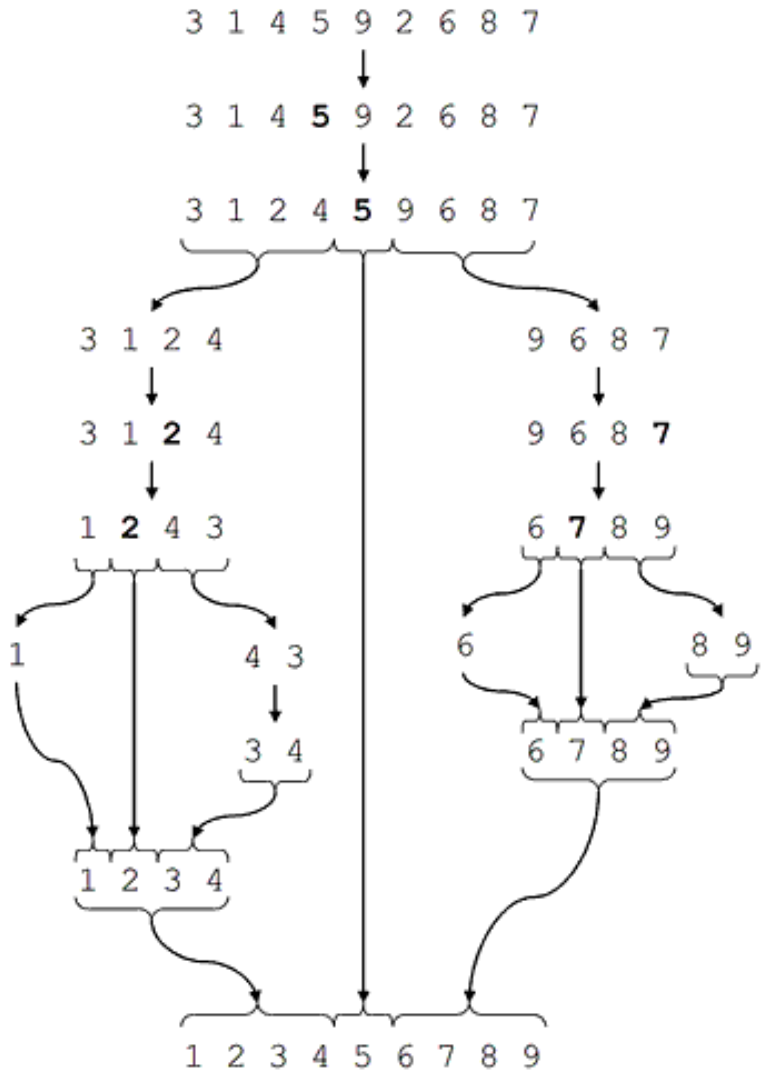
```
void level2() {
    #pragma omp parallel sections
    {
        #pragma omp section
        {  printf("L2 1 Thread PID %u\n", (unsigned int)pthread_self());  }
        #pragma omp section
        {  printf("L2 2 Thread PID %u\n", (unsigned int)pthread_self());  }
    }
}
void level1() {
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {   printf("L1 1 Thread PID %u\n", (unsigned int)pthread_self());
            level2();  }
        #pragma omp section
        {  printf("L1 2 Thread PID %u\n", (unsigned int)pthread_self());
            level2();  }
    }
}

int main() {  omp_set_dynamic(0); omp_set_nested(1); level1();  }
```



Секции выполняются свободными потоками параллельного региона

Быстрая сортировка (QuickSort)

```
void partition(int *v, int& i, int& j, int low, int high) {
    i = low;
    j = high;
    int pivot = v[(low + high) / 2];
    do {
        while (v[i] < pivot) i++;
        while (v[j] > pivot) j--;
        if (i <= j) {
            std::swap(v[i], v[j]);
            i++;
            j--;
        }
    } while (i <= j);
}

void quicksort(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);
    if (low < j)
        quicksort(v, low, j);
    if (i < high)
        quicksort(v, i, high);
}
```



Быстрая сортировка (QuickSort) – версия 1 (nested sections)

```
omp_set_nested(1); // Enable nested parallel regions
...

void quicksort_nested(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            if (low < j) quicksort_nested(v, low, j);
        }
        #pragma omp section
        {
            if (i < high) quicksort_nested(v, i, high);
        }
    }
}
```

- Неограниченная глубина вложенных параллельных регионов
- Отдельные потоки создаются даже для сортировки коротких отрезков [low, high]

Быстрая сортировка (QuickSort) – версия 2 (max_active_levels)

```
omp_set_nested(1); // Enable nested parallel regions
omp_set_max_active_levels(4); // Maximum allowed number of nested, active parallel regions
...
void quicksort_nested(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            if (low < j) quicksort_nested(v, low, j);
        }
        #pragma omp section
        {
            if (i < high) quicksort_nested(v, i, high);
        }
    }
}
```

Быстрая сортировка (QuickSort) – версия 3 (пороговое значение)

```
omp_set_nested(1); // Enable nested parallel regions
omp_set_max_active_levels(4); // Maximum allowed number of nested, active parallel regions
...
void quicksort_nested(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    if (high - low < threshold || (j - low < threshold || high - i < threshold)) {
        if (low < j) // Sequential execution
            quicksort_nested(v, low, j);
        if (i < high)
            quicksort_nested(v, i, high);
    } else {
        #pragma omp parallel sections num_threads(2)
        {
            #pragma omp section
            {  quicksort_nested(v, low, j);  }

            #pragma omp section
            {  quicksort_nested(v, i, high);  }
        }
    }
}
```

- Короткие интервалы сортируем последовательным алгоритмом
- Сокращение накладных расходов на создание потоков

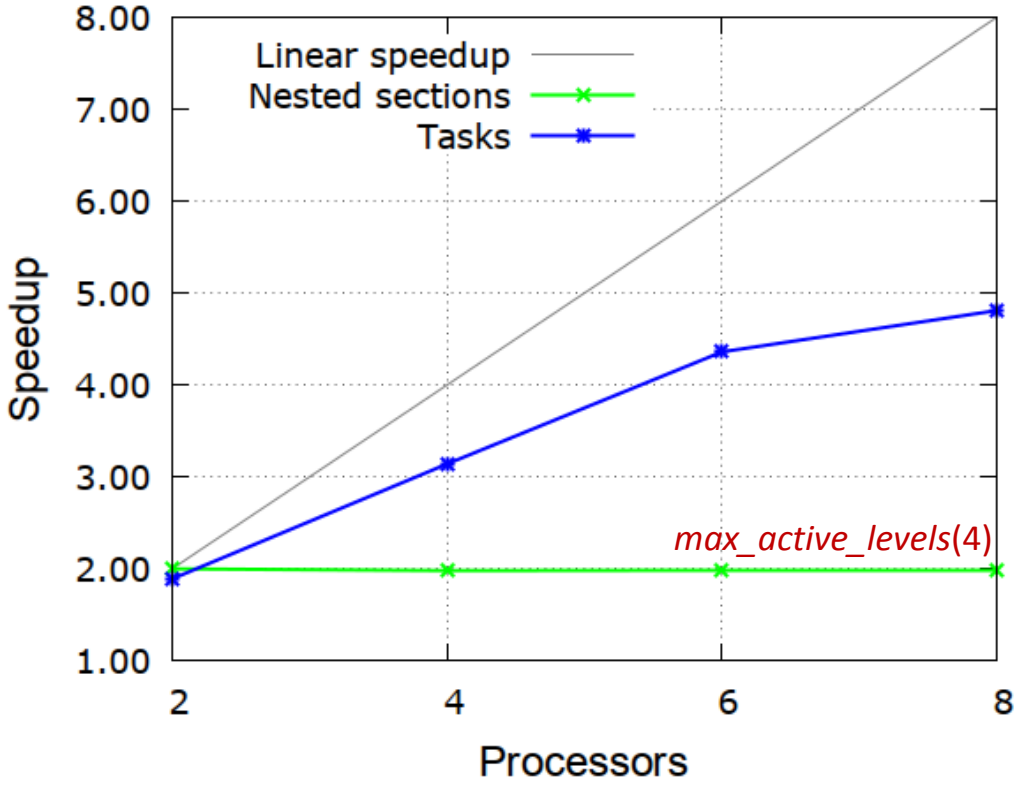
Быстрая сортировка (QuickSort) – версия 4 (tasks)

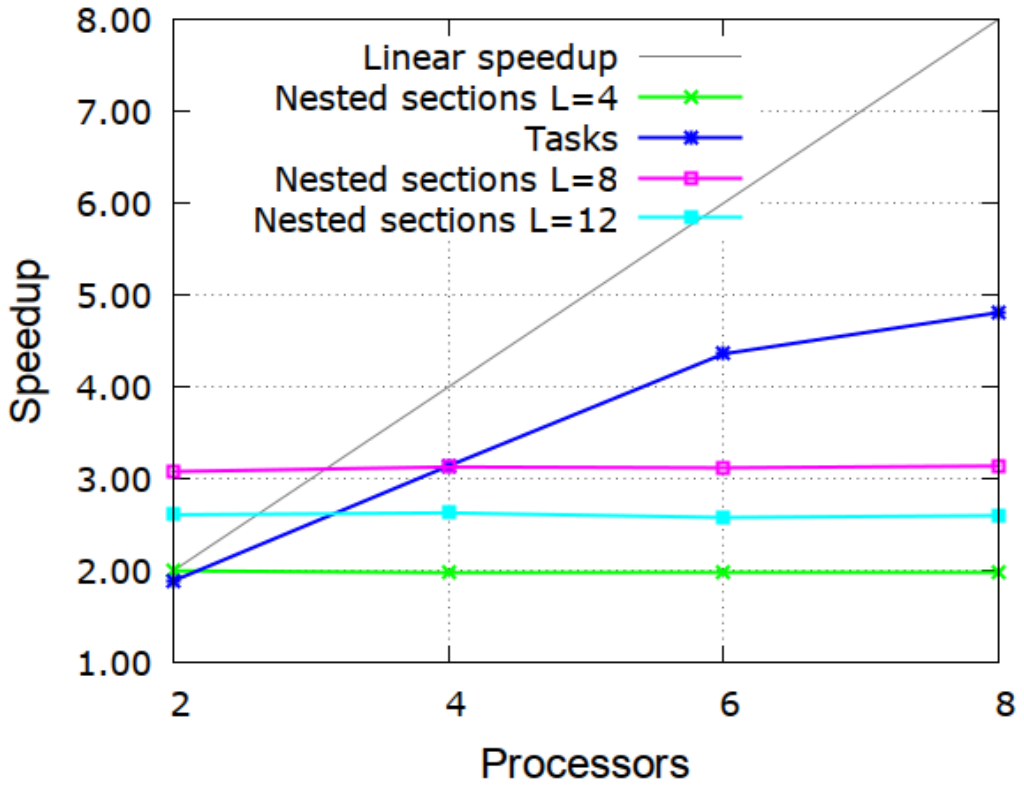
```
#pragma omp parallel
{
    #pragma omp single
    quicksort_tasks(array, 0, size - 1);
}
...

void quicksort_tasks(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    if (high - low < threshold || (j - low < threshold || high - i < threshold)) {
        if (low < j)
            quicksort_tasks(v, low, j);
        if(i < high)
            quicksort_tasks(v, i, high);
    } else {
        #pragma omp task
        {  quicksort_tasks(v, low, j);  }
        quicksort_tasks(v, i, high);
    }
}
```

Быстрая сортировка (QuickSort) – версия 4 (tasks)

```
#pragma omp parallel
{
    #pragma omp single
    quicksort_tasks(array, 0, size - 1);
}
...

void quicksort_tasks(int *v, int low, int high) {
    int i, j;
    partition(v, i, j, low, high);

    if (high - low < threshold || (j - low < threshold || high - i < threshold)) {
        if (low < j)
            quicksort_tasks(v, low, j);
        if(i < high)
            quicksort_tasks(v, i, high);
    } else {
        #pragma omp task untied  // Открепить задачу от потока (задачу может выполнять любой поток)
        {  quicksort_tasks(v, low, j);  }
        quicksort_tasks(v, i, high);
    }
}
```

Быстрая сортировка (QuickSort)





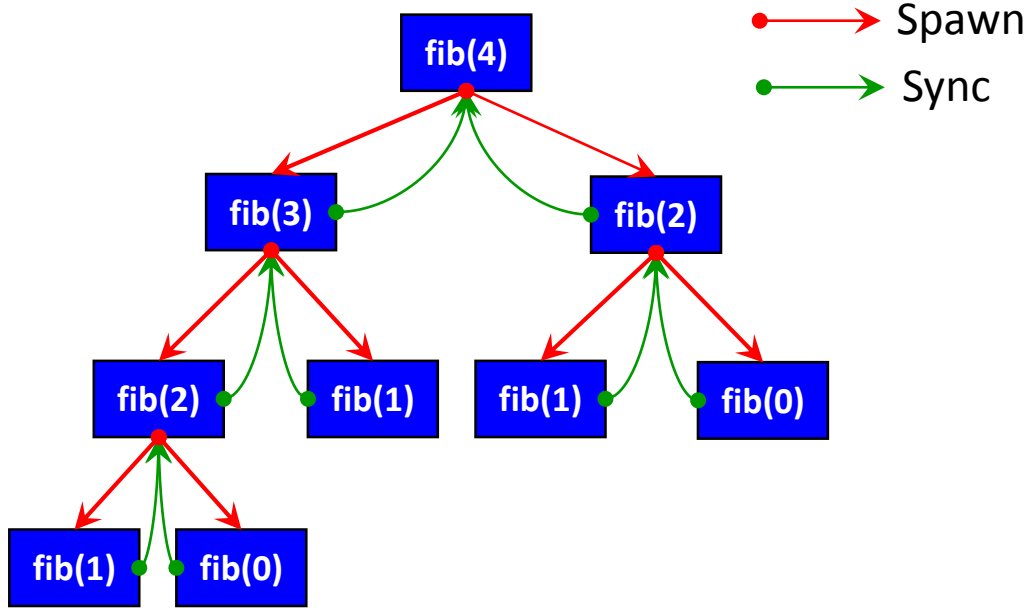
max_active_levels(8)
max_active_levels(12)
max_active_levels(4)

Вычислительный узел Intel S5000VSA:
2 x Intel Quad Xeon E5420, RAM 8 GB (4 x 2GB PC-5300)

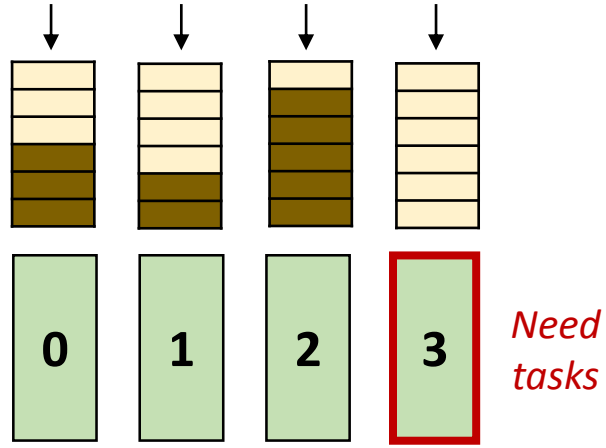
Параллелизм задач (OpenMP 3.1)

```
int fib(int n)
{
    int x, y;

    if (n < 2)
        return n;
#pragma omp task shared(x, n)
    x = fib(n - 1);
#pragma omp task shared(y, n)
    y = fib(n - 2);
#pragma omp taskwait
    return x + y;
}

#pragma omp parallel
#pragma omp single
    val = fib(n);
```

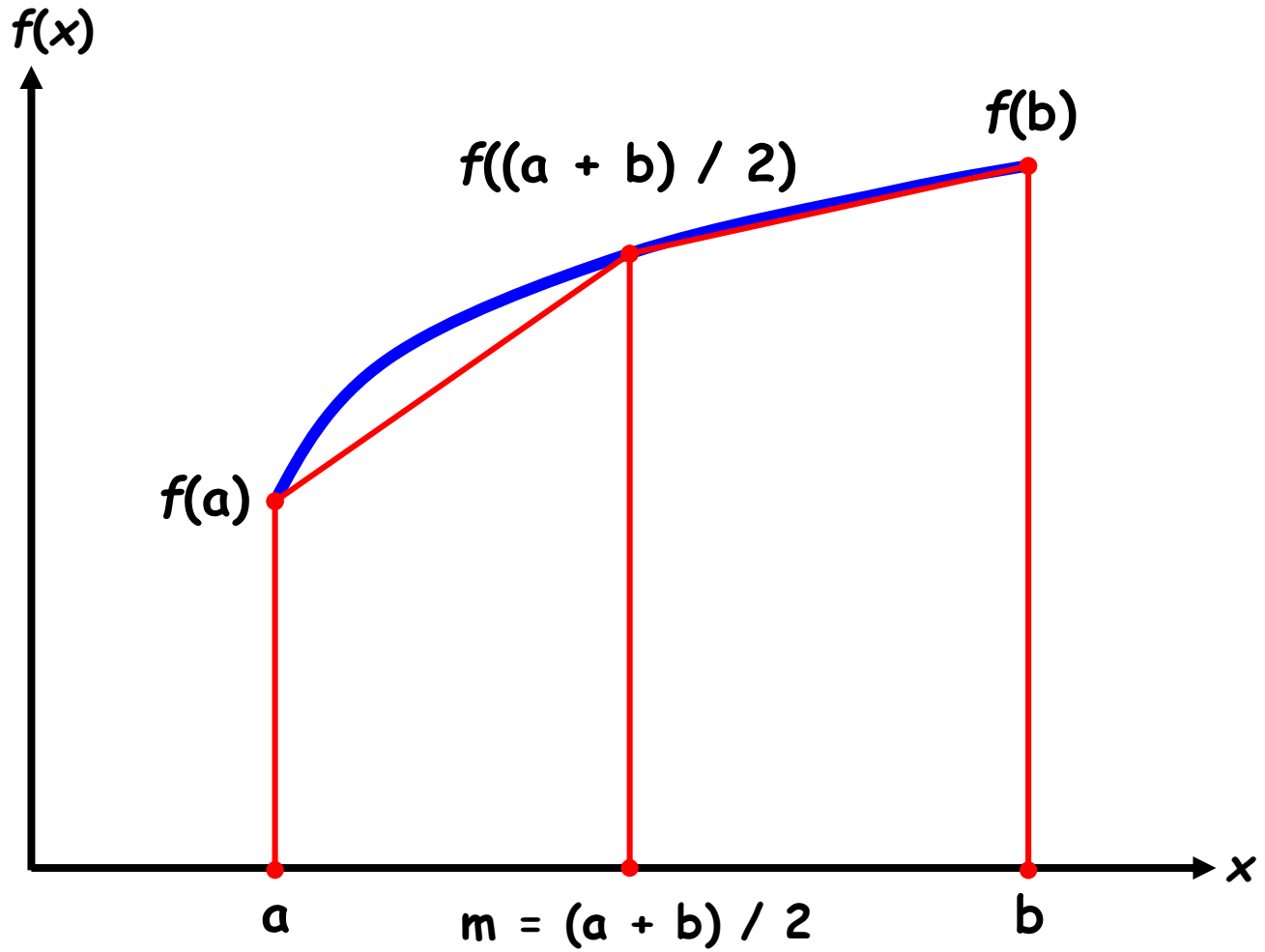
Легковесные задачи (tasks)



Очереди (деки) задач + пул потоков операционной системы (*Work stealing*)



Вычисление определенного интеграла методом трапеций



- Площадь левой трапеции:

$$S_L = (f(a) + f(m)) * (m - a) / 2$$

- Площадь правой трапеции:

$$S_R = (f(m) + f(b)) * (b - m) / 2$$

$$S = S_L + S_R$$

Метод трапеций – вычисление числа π

```
long double f(double x) { return 4.0 / (1.0 + x * x); }

long double quad(double left, double right, long double f_left, long double f_right,
                 long double lr_area)
{
    double mid = (left + right) / 2;
    long double f_mid = f(mid);
    long double l_area = (f_left + f_mid) * (mid - left) / 2;
    long double r_area = (f_mid + f_right) * (right - mid) / 2;
    if (fabs((l_area + r_area) - lr_area) > eps) {
        l_area = quad(left, mid, f_left, f_mid, l_area);
        r_area = quad(mid, right, f_mid, f_right, r_area);
    }
    return (l_area + r_area);
}

int main(int argc, char *argv[]) {
    double start = omp_get_wtime();
    long double pi = quad(0.0, 1.0, f(0), f(1), (f(0) + f(1)) / 2);
}
```

Метод трапеций – вычисление числа π (OpenMP tasks)

```
long double quad_tasks(double left, double right, long double f_left, long double f_right,
                       long double lr_area)
{
    double mid = (left + right) / 2;
    long double f_mid = f(mid);
    long double l_area = (f_left + f_mid) * (mid - left) / 2;
    long double r_area = (f_mid + f_right) * (right - mid) / 2;
    if (fabs((l_area + r_area) - lr_area) > eps) {
        if (right - left < threshold) {
            l_area = quad_tasks(left, mid, f_left, f_mid, l_area);
            r_area = quad_tasks(mid, right, f_mid, f_right, r_area);
        } else {
            #pragma omp task shared(l_area)
            {
                l_area = quad_tasks(left, mid, f_left, f_mid, l_area);
            }
            r_area = quad_tasks(mid, right, f_mid, f_right, r_area);
            #pragma omp taskwait
        }
    }
    return (l_area + r_area);
}
```

Метод трапеций – вычисление числа π (OpenMP tasks)

```
int main(int argc, char *argv[])
{

    start = omp_get_wtime();
    #pragma omp parallel
    #pragma omp single
    {
        pi = quad_tasks(0.0, 1.0, f(0), f(1), (f(0) + f(1)) / 2);
        printf("PI is approximately %.16Lf, Error is %.16f\n", pi, fabs(pi - pi_real));
    }
    printf("Parallel version: %.6f sec.\n", omp_get_wtime() - start);

    return 0;
}
```

Накладные расходы OpenMP (overhead)

Constructs	Cost (in microseconds)	Scalability
parallel	1.5	Linear
Barrier	1.0	Linear or O(log(n))
schedule(static)	1.0	Linear
schedule(guided)	6.0	Depends on contention
schedule(dynamic)	50	Depends on contention
ordered	0.5	Depends on contention
Single	1.0	Depends on contention
Reduction	2.5	Linear or O(log(n))
Atomic	0.5	Depends on data-type and hardware
Critical	0.5	Depends on contention
Lock/Unlock	0.5	Depends on contention

Shameem Akhter and Jason Roberts. Using OpenMP for programming parallel threads in multicore applications: Part 2
// http://www.embedded.com/design/mcus-processors-and-socs/4007155/Using-OpenMP-for-programming-parallel-threads-in-multicore-applications-Part-2

Эхтер Ш., Робертс Дж. **Многоядерное программирование**. – СПб.: Питер, 2010. – 316 с.

Что осталось за кадром

- **#pragma omp atomic**
- **#pragma omp barrier**
- **#pragma omp flush**
- **#pragma omp parallel if**
- **#pragma omp for ordered**
- **omp_lock_set(), omp_lock_unset()**

- **OpenMP 4.0**
- **#pragma omp target device(acc0) in(B,C)**
- **#pragma omp parallel for simd**
- **#pragma omp parallel proc_bind(master | close | spread)**
- **#pragma omp declare reduction**

OpenM 4.0 – программирование ускорителей

```
sum = 0;
#pragma omp target device(acc0) in(B, C)
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < N; i++)
    sum += B[i] * C[i]
```

- **omp_set_default_device()**
- **omp_get_default_device()**
- **omp_get_num_devices()**

OpenM 4.0 – SIMD-директивы (векторизация)

```
void minex(float *a, float *b, float *c, float *d)
{
    #pragma omp parallel for simd
    for (i = 0; i < N; i++)
        d[i] = min(distsq(a[i], b[i]), c[i]);
}
```

- SIMD-конструкции для векторизации циклов
  (наборы SIMD-инструкций: SSE, AVX2, AVX-512, AltiVec, NEON SIMD, …)

OpenM 4.0 – Привязка к процессорам (Thread affinity)

- **Thread affinity** – привязка потоков к процессорным ядрам (логическим процессорам операционной системы)

- `#pragma omp parallel proc_bind(master | close | spread)`
- `omp_proc_bind_t omp_get_proc_bind(void)`
- Переменная среды OMP_PLACES

Домашнее чтение

- **32 подводных камня OpenMP при программировании на Си++ //** http://www.viva64.com/ru/a/0054/
- **Intel Threading Tools and OpenMP //** http://www.viva64.com/ru/r/0047/
- **How to Get Good Performance by Using OpenMP //** http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf/4_Performance.pdf