Лекция 7
Стандарт OpenMP (продолжение)

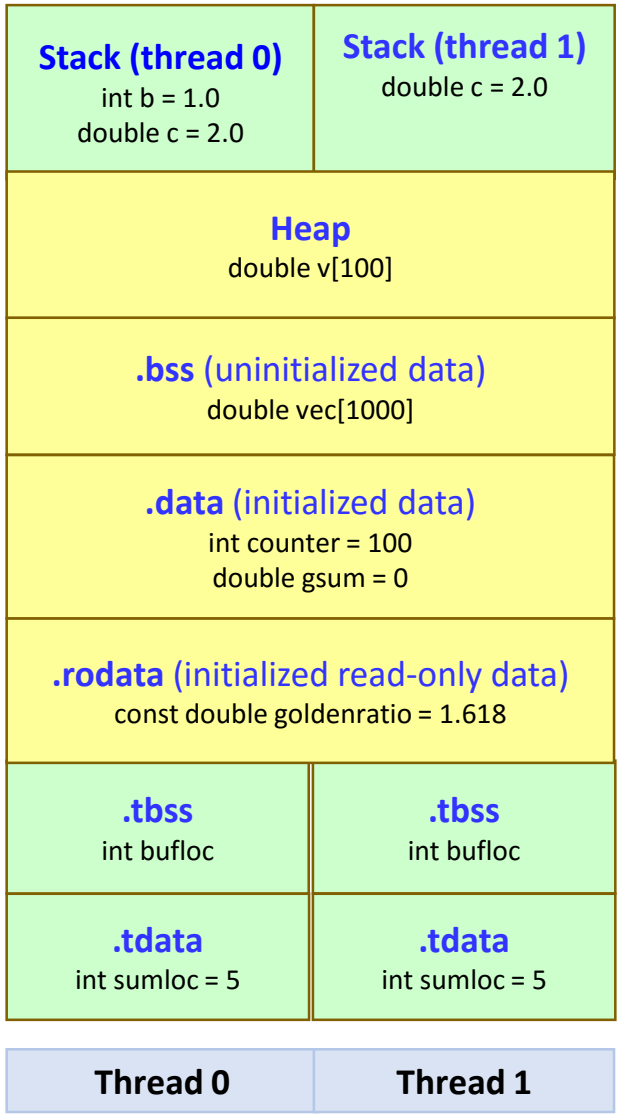
Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)
Осенний семестр, 2015

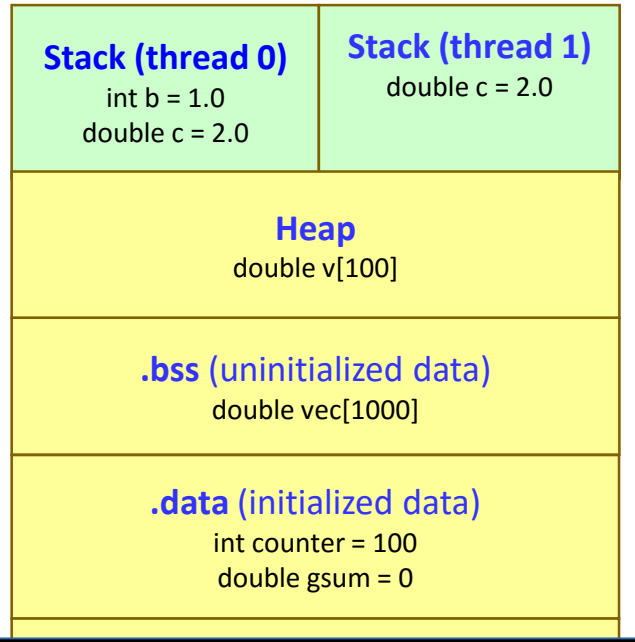
Видимость данных (C11 storage duration)

```c
const double goldenratio = 1.618;           /* Static (.rodata) */
double vec[1000];                           /* Static (.bss) */
int counter = 100;                          /* Static (.data) */

double fun(int a)
{
    double b = 1.0;                         /* Automatic (stack, register) */

    static double gsum = 0;                 /* Static (.data) */

    _Thread_local static double sumloc = 5; /* Thread (.tdata) */
    _Thread_local static double bufloc;     /* Thread (.tbbs) */

    double *v = malloc(sizeof(*v) * 100);   /* Allocated (Heap) */

    #pragma omp parallel num_threads(2)
    {
        double c = 2.0;                     /* Automatic (stack, register) */

        /* Shared: goldenratio, vec[], counter, b, gsum, v[] */
        /* Private: sumloc, bufloc, c */
    }

    free(v);
}
```



Видимость данных (C11 storage duration)

```c
const double goldenratio = 1.618;           /* Static (.rodata) */
double vec[1000];                           /* Static (.bss) */
int counter = 100;                          /* Static (.data) */

double fun(int a)
{
    double b = 1.0;                         /* Automatic (stack, register) */

    static double gsum = 0;                 /* Static (.data) */

    _Thread_local static double sumloc = 5; /* Thread (.tdata) */
    _Thread_local static double bufloc;     /* Thread (.tbbs) */

    double *v = malloc(sizeof(*v) * 100);   /* Allocated (Heap) */

    #pragma omp parallel num_threads(2)
    {
        double c = 2.0;

        /* Shared: goldenratio, vec[], cou
        /* Private: sumloc, bufloc, c */
    }

    free(v);
}
```




```
$ objdump --syms ./datasharing

./datasharing:     file format elf64-x86-64
SYMBOL TABLE:

0000000000601088 l     O .bss     0000000000000008    gsum.2231
0000000000000000 l       .tdata   0000000000000008    sumloc.2232
0000000000000008 l       .tbss    0000000000000008    bufloc.2233
00000000006010c0 g     O .bss     0000000000001f40    vec
000000000060104c g     O .data    0000000000000004    counter
00000000004008e0 g     O .rodata  0000000000000008    goldenratio
```

Атрибуты видимости данных

```
#pragma omp parallel shared(a, b, c) private(x, y, z) firstprivate(i, j, k)
{
    #pragma omp for lastprivate(v)
    for (int i = 0; i < 100; i++)
}
```

- **shared** (list) – указанные переменные сохраняют исходный класс памяти (auto, static, thread_local), все переменные кроме thread_local будут разделяемыми
- **private** (list) – для каждого потока создаются локальные копии указанных переменных (automatic storage duration)
- **firstprivate** (list) – для каждого потока создаются локальные копии переменных (automatic storage duration), они инициализируются значениями, которые имели соответствующие переменные до входа в параллельный регион
- **lastprivate** (list) – для каждого потока создаются локальные копии переменных (automatic storage duration), в переменные копируются значения последней итерации цикла, либо значения последней параллельной секции в коде (#pragma omp section)
- **#pragma omp threadprivate(list)** — делает указанные статические переменные локальными (TLS)

Атрибуты видимости данных

```
void fun()
{
    int a = 100;
    int b = 200;
    int c = 300;
    int d = 400;
    static int sum = 0;
    printf("Before parallel: a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);

    #pragma omp parallel private(a) firstprivate(b) num_threads(2)
    {
        int tid = omp_get_thread_num();
        printf("Thread %d: a = %d, b = %d, c = %d, d = %d\n", tid, a, b, c, d);
        a = 1;  b = 2;

        #pragma omp threadprivate(sum)
        sum++;

        #pragma omp for lastprivate(c)
        for (int i = 0; i < 100; i++)
            c = i;
        /* c=99 - has the value from last iteration */

    }
    // a = 100, b = 200, c = 99, d = 400, sum = 1
    printf("After parallel: a = %d, b = %d, c = %d, d = %d\n", a, b, c, d);
}
```

```
Before parallel: a = 100, b = 200, c = 300, d = 400
Thread 0: a = 0, b = 200, c = 300, d = 400
Thread 1: a = 0, b = 200, c = 300, d = 400
After parallel: a = 100, b = 200, c = 99, d = 400
```

Редукция (reduction, reduce)

```
int count_prime_numbers_omp(int a, int b)
{
    int nprimes = 0;

    /* Count '2' as a prime number */
    if (a <= 2) {
        nprimes = 1;
        a = 2;
    }

    /* Shift 'a' to odd number */
    if (a % 2 == 0)
        a++;

    #pragma omp parallel
    {
        #pragma omp for schedule(dynamic,100) reduction(+:nprimes)
        for (int i = a; i <= b; i += 2) {
            if (is_prime_number(i))
                nprimes++;
        }
    }
    return nprimes;
}
```

- В каждом потоке создает private-переменная nprimes
- После завершения параллельного региона к локальным
- копиям применяется операция «+»
- Результат редукции записывается в переменную nprimes
- Допустимые операции: +, -, *, &, |, ^, &&, ||

Начальные значения переменных редукции

Identifier	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 = *Least representable number in the reduction list item type*	omp_out = omp_in > omp_out ? omp_in : omp_out
min	omp_priv = *Largest representable number in the reduction list item type*	omp_out = omp_in < omp_out ? omp_in : omp_out

Умножение матрицы на вектор (DGEMV)

- Требуется вычислить произведение прямоугольной матрицы ***A*** размера $m \times n$ на вектор-столбец ***B*** размера $m \times 1$ (BLAS Level 2, DGEMV)

$$\boldsymbol{C_{m\times 1} = A_{m\times n} \cdot B_{n\times 1}}$$

$$C = \begin{pmatrix} c_1 \\ c_2 \\ \dots \\ c_m \end{pmatrix} \qquad A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \qquad B = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$$

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j\,, \qquad i = 1,2,\dots,m.$$

DGEMV: последовательная версия

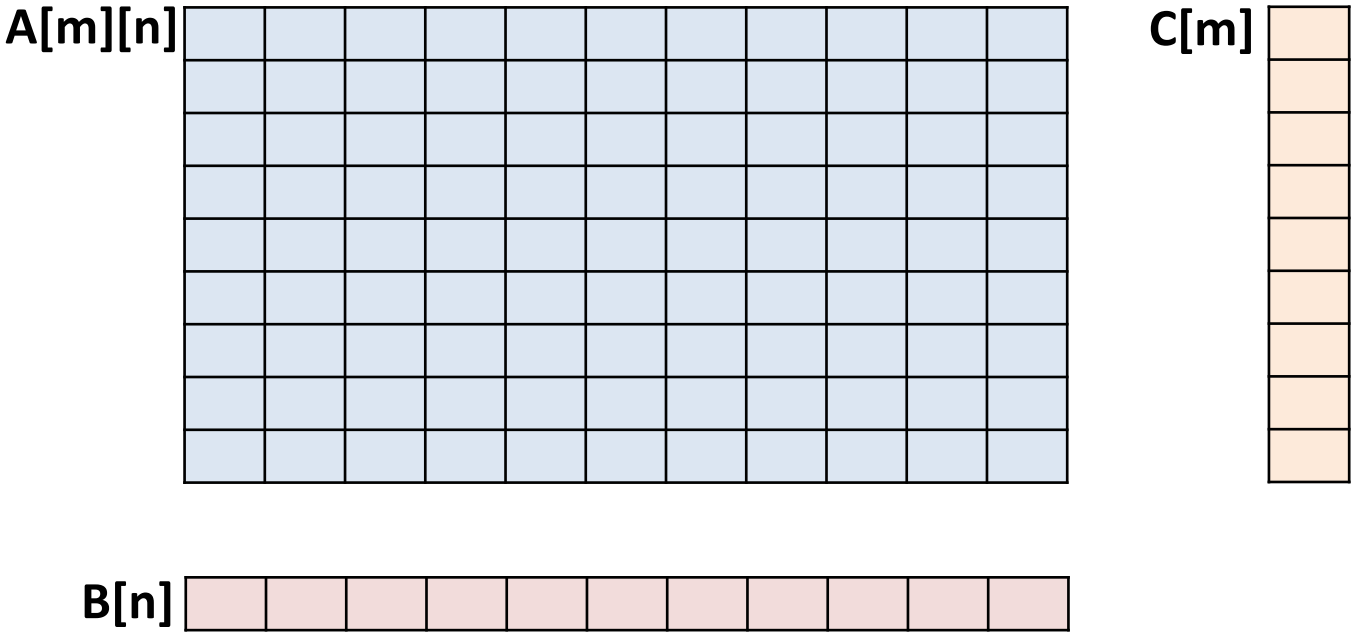
```
/*
 * matrix_vector_product: Compute matrix-vector product c[m] = a[m][n] * b[n]
 */
void matrix_vector_product(double *a, double *b, double *c, int m, int n)
{
    for (int i = 0; i < m; i++) {
        c[i] = 0.0;
        for (int j = 0; j < n; j++)
            c[i] += a[i * n + j] * b[j];
    }
}
```

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j , \qquad i = 1,2, \dots , m.$$

DGEMV: последовательная версия

```
void run_serial()
{
    double *a, *b, *c;

    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

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

    double t = wtime();
    matrix_vector_product(a, b, c, m, n);
    t = wtime() - t;

    printf("Elapsed time (serial): %.6f sec.\n", t);
    free(a);
    free(b);
    free(c);
}
```

DGEMV: параллельная версия

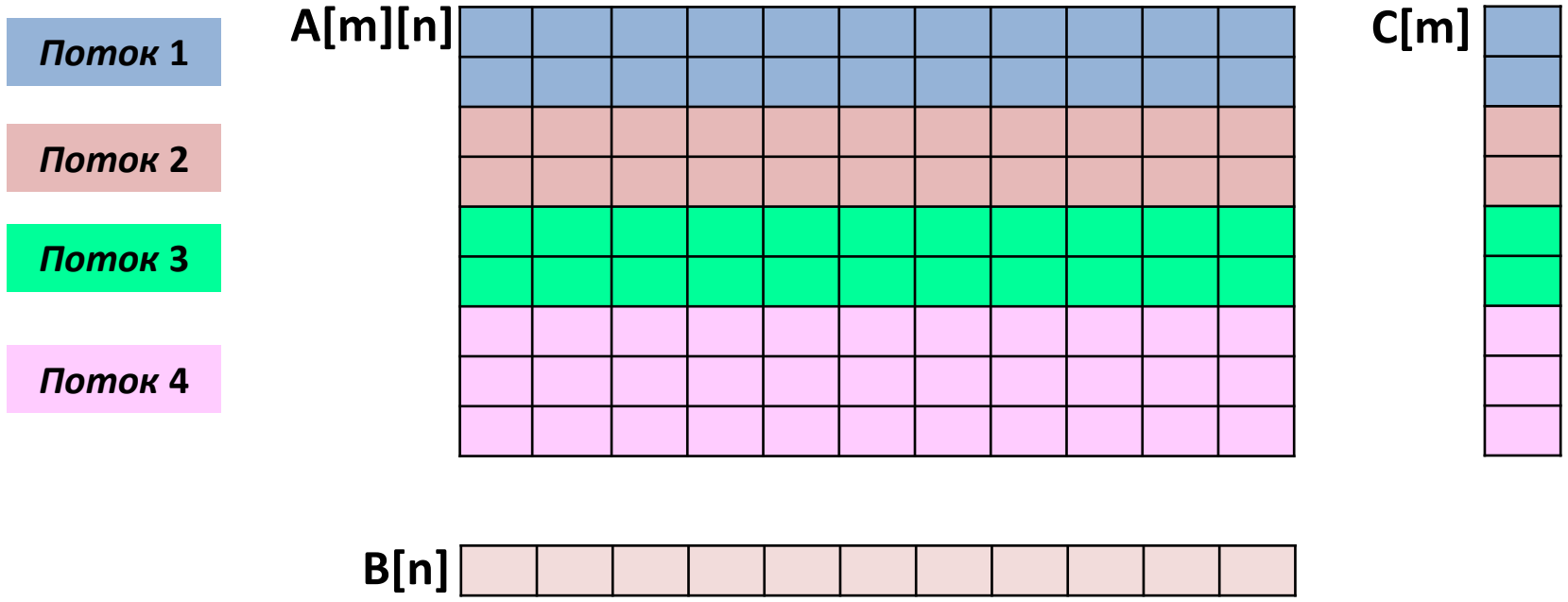



```
for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}
```

Требования к параллельному алгоритму

- Максимальная загрузка потоков вычислениями
- Минимум совместно используемых ячеек памяти – независимые области данных

DGEMV: параллельная версия



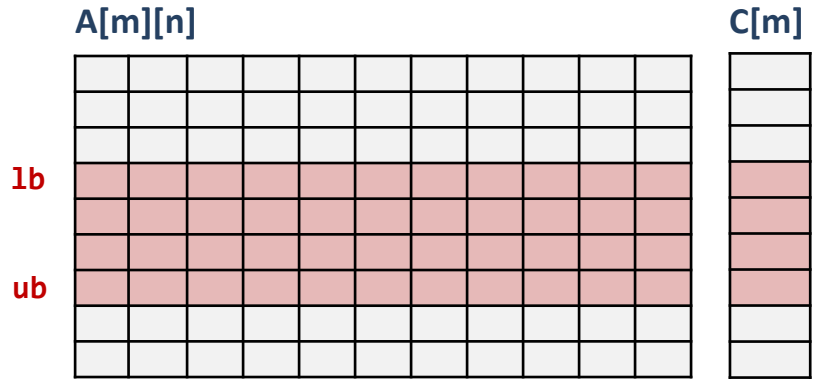

```
for (int i = 0; i < m; i++) {
    c[i] = 0.0;
    for (int j = 0; j < n; j++)
        c[i] += a[i * n + j] * b[j];
}
```

Распараллеливание внешнего цикла

- Каждом потоку выделяется часть строк матрицы *A*

DGEMV: параллельная версия

```
/* matrix_vector_product_omp: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            c[i] = 0.0;
            for (int j = 0; j < n; j++)
                c[i] += a[i * n + j] * b[j];
        }
    }
}
```



DGEMV: параллельная версия

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

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

    double t = wtime();
    matrix_vector_product_omp(a, b, c, m, n);
    t = wtime() - t;

    printf("Elapsed time (parallel): %.6f sec.\n", t);
    free(a);
    free(b);
    free(c);
}
```

DGEMV: параллельная версия

```
int main(int argc, char **argv)
{
    printf("Matrix-vector product (c[m] = a[m, n] * b[n]; m = %d, n = %d)\n", m, n);
    printf("Memory used: %" PRIu64 " MiB\n", ((m * n + m + n) * sizeof(double)) >> 20);

    run_serial();
    run_parallel();

    return 0;
}
```

Анализ эффективности OpenMP-версии

- **Введем обозначения:**
  - $\boldsymbol{T(n)}$ – время выполнения последовательной программы (serial program) при заданном размере *n* входных данных
  - $\boldsymbol{T_p(n,p)}$ – время выполнения параллельной программы (parallel program) на *p* процессорах при заданном размере *n* входных данных

- **Коэффициент $S_p(n)$ ускорения** параллельной программ (Speedup):

$$\boldsymbol{S_p(n) = \frac{T(n)}{T_p(n)}}$$

Во сколько раз параллельная программа выполняется на *p* процессорах быстрее последовательной программы при обработке одних и тех же данных размера *n*

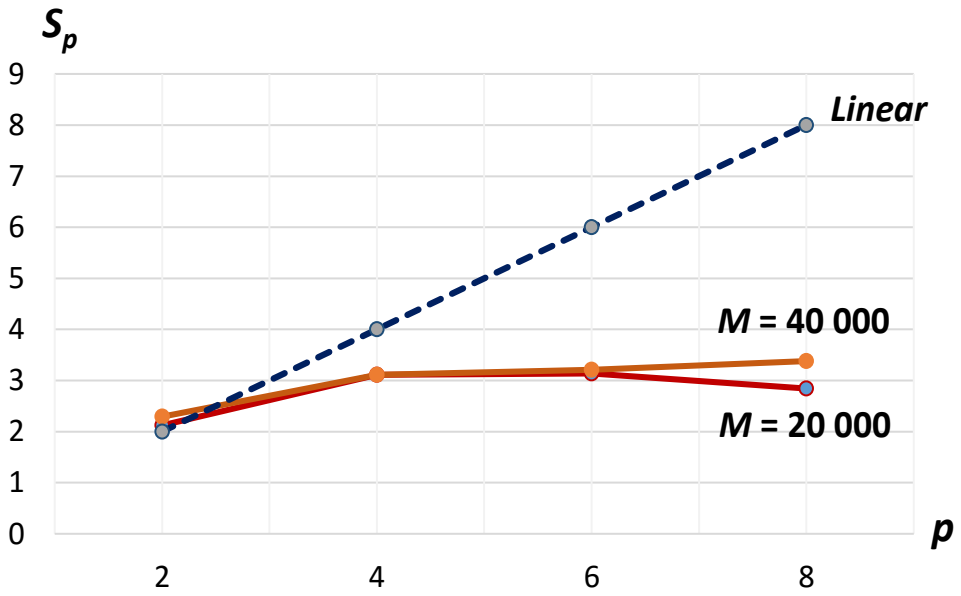
- Как правило

$$S_p(n) \leq p$$

- **Цель распараллеливания –**
  достичь линейного ускорения на наибольшем числе процессоров: $S_p(n) \geq c \cdot p, \quad \text{при } p \to \infty \text{ и } c > 0$

Анализ эффективности OpenMP-версии

M = N	Количество потоков								
	2			4		6		8	
	T_1	T_2	S_2	T_4	S_4	T_6	S_6	T_8	S_8
20 000 (~ 3 GiB)	0.73	0.34	**2.12**	0.24	**3.11**	0.23	**3.14**	0.25	**2.84**
40 000 (~ 12 GiB)	2.98	1.30	**2.29**	0.95	**3.11**	0.91	**3.21**	0.87	**3.38**
49 000 (~ 18 GiB)								1.23	**3.69**



Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR**
- **8 ядер** – два Intel Quad Xeon E5620 (2.4 GHz)
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz
- CentOS 6.5 x86_64, GCC 4.4.7
- Ключи компиляции: -std=c99 -Wall -O2 -fopenmp

Низкая масштабируемость!

Причины ?

DGEMV: конкуренция за доступ к памяти

```
/* matrix_vector_product_omp: Compute matrix-vector product c[m] = a[m][n] * b[n] */
void matrix_vector_product_omp(double *a, double *b, double *c, int m, int n)
{
    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            c[i] = 0.0; // Store – запись в память
            for (int j = 0; j < n; j++)
                // Memory ops: Load c[i], Load a[i][j], Load b[j], Store c[i]
                c[i] += a[i * n + j] * b[j];
        }
    }
}
```

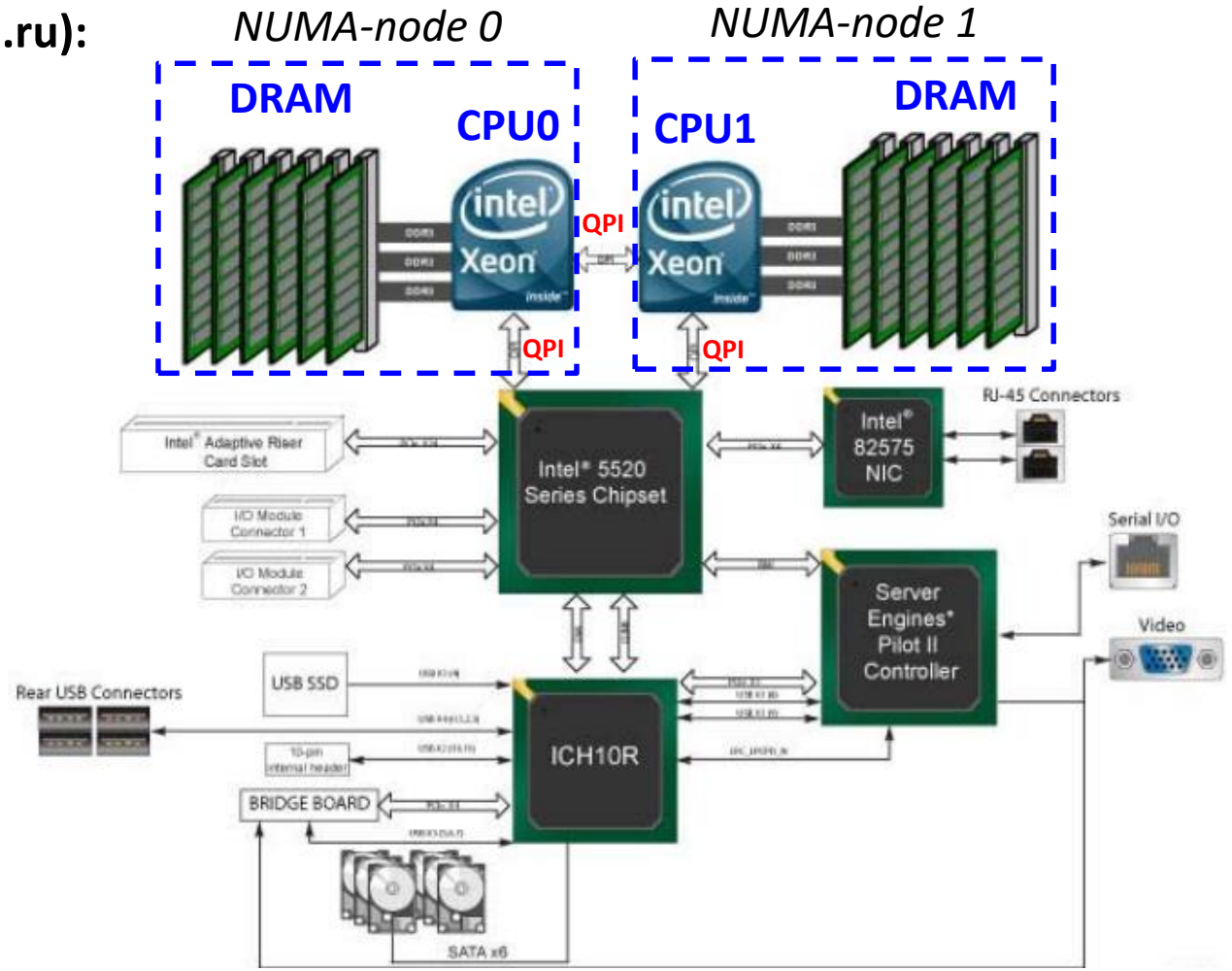
- DGEMV – *data intensive application*
- Конкуренция за доступ к контролеру памяти
- ALU ядер загружены незначительно

Конфигурация узла кластера Oak

Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR** (NUMA-система)
- **Процессоры связаны шиной QPI Link: 5.86 GT/s**
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 12224 MB
node 0 free: 11443 MB
node 1 cpus: 1 3 5 7
node 1 size: 12288 MB
node 1 free: 11837 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```



http://download.intel.com/support/motherboards/server/s5520ur/sb/e44031012_s5520ur_s5520urt_tps_r1_9.pdf

Конфигурация узла кластера Oak

Вычислительный узел кластера Oak (oak.cpct.sibsutis.ru):

- **System board: Intel 5520UR** (NUMA-система)
- **Процессоры связаны шиной QPI Link: 5.86 GT/s**
- **24 GiB RAM** – 6 x 4GB DDR3 1067 MHz

```
$ numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 12224 MB
node 0 free: 11443 MB
node 1 cpus: 1 3 5 7
node 1 size: 12288 MB
node 1 free: 11837 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

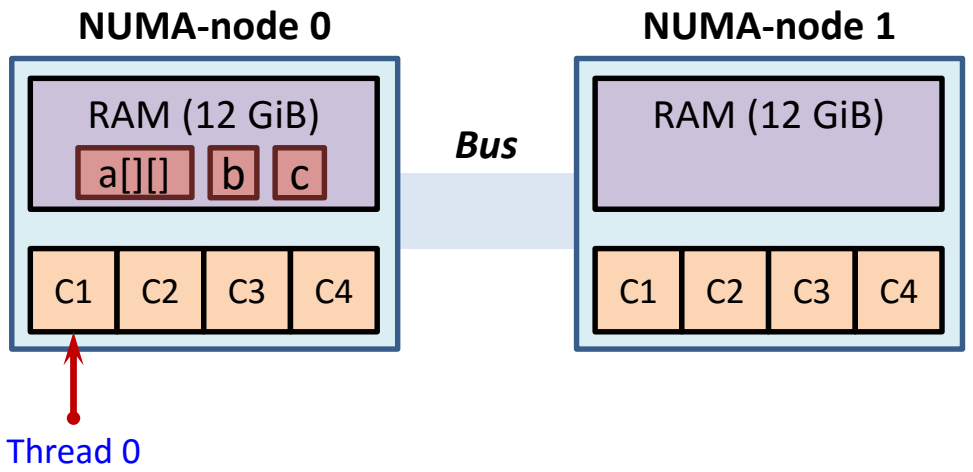
А на каком NUMA-узле (узлах) размещена матрица A и векторы B, C?

http://download.intel.com/support/motherboards/server/s5520ur/sb/e44031012_s5520ur_s5520urt_tps_r1_9.pdf

Выделение памяти потокам в GNU/Linux

- Страница памяти выделяется с NUMA-узла того потока, который первый к ней обратился (*first-touch policy*)
- Данные желательно инициализировать теми потоками, которые будут с ними работать

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

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

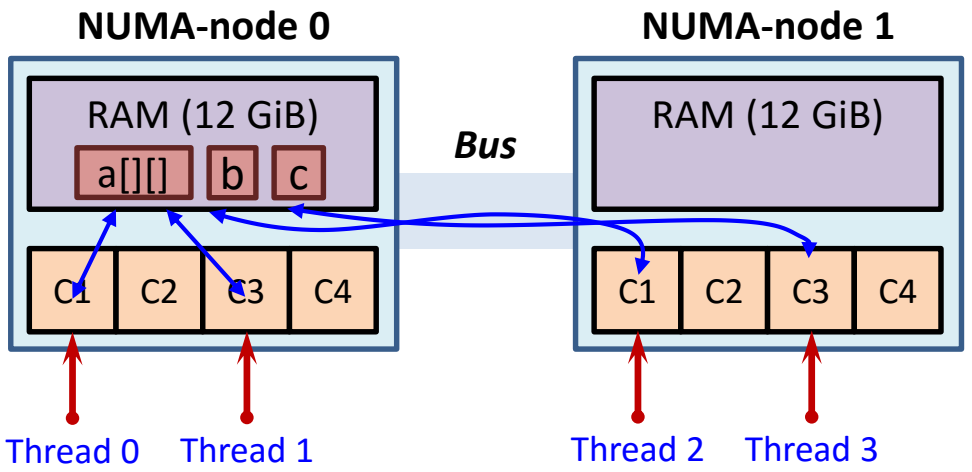


- Поток 0 запрашивает выделение памяти под массивы
- Пока хватает памяти, ядро выделяет страницы с NUMA-узла 0, затем с NUMA-узла 1

Выделение памяти потокам в GNU/Linux

- Страница памяти выделяется с NUMA-узла того потока, который первый к ней обратился (*first-touch policy*)
- Данные желательно инициализировать теми потоками, которые будут с ними работать

```
void run_parallel()
{
    double *a, *b, *c;

    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

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



- Обращение к массивам из потоков NUMA-узла 1 будет идти через межпроцессорную шину в память узла 0

Параллельная инициализация массивов

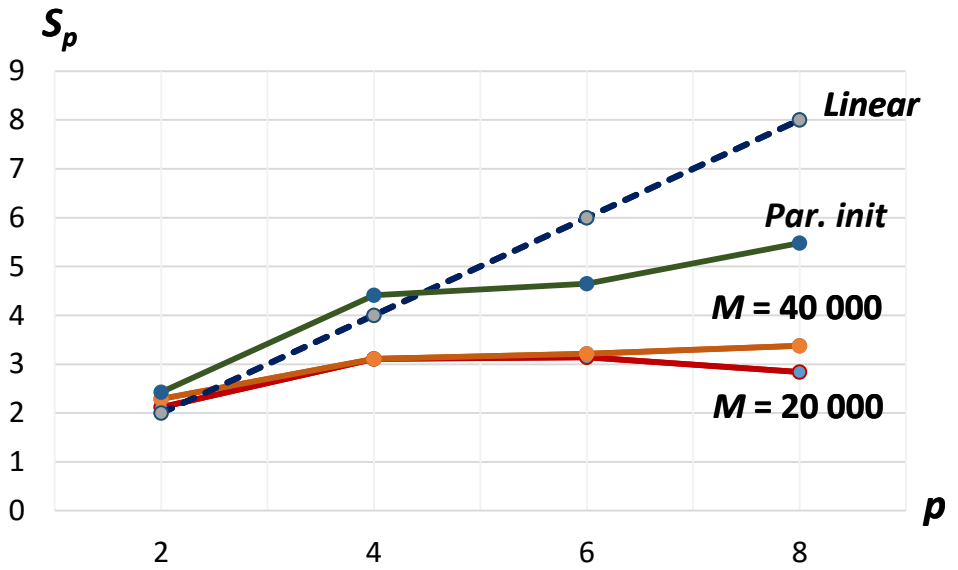
```
void run_parallel()
{
    double *a, *b, *c;
    // Allocate memory for 2-d array a[m, n]
    a = xmalloc(sizeof(*a) * m * n);
    b = xmalloc(sizeof(*b) * n);
    c = xmalloc(sizeof(*c) * m);

    #pragma omp parallel
    {
        int nthreads = omp_get_num_threads();
        int threadid = omp_get_thread_num();
        int items_per_thread = m / nthreads;
        int lb = threadid * items_per_thread;
        int ub = (threadid == nthreads - 1) ? (m - 1) : (lb + items_per_thread - 1);

        for (int i = lb; i <= ub; i++) {
            for (int j = 0; j < n; j++)
                a[i * n + j] = i + j;
            c[i] = 0.0;
        }
    }
    for (int j = 0; j < n; j++)
        b[j] = j;

    /* ... */
}
```

Анализ эффективности OpenMP-версии (2)

M = N	Количество потоков								
	2			4		6		8	
	T_1	T_2	S_2	T_4	S_4	T_6	S_6	T_8	S_8
20 000 (~ 3 GiB)	0.73	0.34	**2.12**	0.24	**3.11**	0.23	**3.14**	0.25	**2.84**
40 000 (~ 12 GiB)	2.98	1.30	**2.29**	0.95	**3.11**	0.91	**3.21**	0.87	**3.38**
49 000 (~ 18 GiB)								1.23	3.69

Parallel initialization									
40 000 (~ 12 GiB)	2.98	1.22	**2.43**	0.67	**4.41**	0.65	**4.65**	0.54	**5.48**
49 000 (~ 18 GiB)								0.83	**5.41**

Суперлинейное ускорение (super-linear speedup): $S_p(n) > p$



Улучшили масштабируемость

Дальнейшие оптимизации:

- Эффективный доступ к кеш-памяти
- Векторизация кода (SSE/AVX)
- …

Барьерная синхронизация

```
void fun()
{
    #pragma omp parallel
    {
        // Parallel code
        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            x[i] = f(i);

        // Serial code
        #pragma omp single
        do_stuff();

        #pragma omp barrier
        // Ждем готовности x[0:n-1]

        // Parallel code
        #pragma omp for nowait
        for (int i = 0; i < n; i++)
            y[i] = x[i] + 2.0 * f(i);

        // Serial code
        #pragma omp master
        do_stuff_last();
    }
}
```

`#pragma omp barrier`

Потоки ждут пока все не достигнут этого места в программе

Нормализация яркости изображения

```
const uint64_t width = 32 * 1024;  const uint64_t height = 32 * 1024;

void hist_serial(uint8_t *pixels, int height, int width)
{
    uint64_t npixels = height * width;
    int *h = xmalloc(sizeof(*h) * 256);
    for (int i = 0; i < 256; i++)
        h[i] = 0;

    for (int i = 0; i < npixels; i++)
        h[pixels[i]]++;

    int mini, maxi;
    for (mini = 0; mini < 256 && h[mini] == 0; mini++);
    for (maxi = 255; maxi >= 0 && h[maxi] == 0; maxi--);

    int q = 255 / (maxi - mini);
    for (int i = 0; i < npixels; i++)
        pixels[i] = (pixels[i] - mini) * q;
    free(h);
}

int main(int argc, char *argv[])
{
    uint64_t npixels = width * height;
    pixels1 = xmalloc(sizeof(*pixels1) * npixels);
    hist_serial(pixels1, height, width);
    // ...
}
```

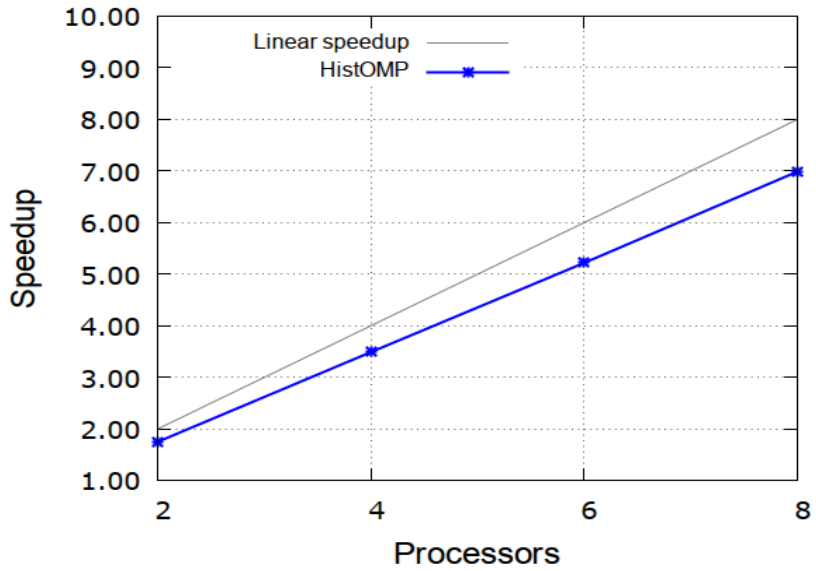
h[i] — количество точек цвета i в изображении (гистограмма)

$$\mathrm{LUT}[i] = 255 \cdot \frac{i - I_{\min}}{I_{\max} - I_{\min}}.$$

Нормализация яркости изображения (OpenMP) v1

```
void hist_omp(uint8_t *pixels, int height, int width) {
    uint64_t npixels = height * width;
    int *h = xmalloc(sizeof(*h) * 256);
    for (int i = 0; i < 256; i++) h[i] = 0;

    #pragma omp parallel
    {
        int *hloc = xmalloc(sizeof(*hloc) * 256);
        for (int i = 0; i < 256; i++)
            hloc[i] = 0;

        #pragma omp for nowait
        for (int i = 0; i < npixels; i++)
            hloc[pixels[i]]++;
        #pragma omp critical
        {
            for (int i = 0; i < 256; i++)
                h[i] += hloc[i];
        }
        free(hloc);

        #pragma omp barrier
        int mini, maxi;
        for (mini = 0; mini < 256 && h[mini] == 0; mini++);
        for (maxi = 255; maxi >=0 && h[maxi] == 0; maxi--);

        int q = 255 / (maxi - mini);
        #pragma omp for
        for (int i = 0; i < npixels; i++)
            pixels[i] = (pixels[i] - mini) * q;
    }
    free(h);
}
```

Локальная таблица hloc[]
в каждом потоке



$$\mathrm{LUT}[i] = 255 \cdot \frac{i - I_{\min}}{I_{\max} - I_{\min}}.$$

Нормализация яркости изображения (OpenMP) v2

```
void hist_omp_v2(uint8_t *pixels, int height, int width) {
    uint64_t npixels = height * width;
    int *h = xmalloc(sizeof(*h) * 256);
    for (int i = 0; i < 256; i++) h[i] = 0;
    int mini = 256, maxi = -1;

    #pragma omp parallel
    {
        int *hloc = xmalloc(sizeof(*hloc) * 256);
        for (int i = 0; i < 256; i++) hloc[i] = 0;

        #pragma omp for nowait
        for (int i = 0; i < npixels; i++)
            hloc[pixels[i]]++;

        int mini_loc, maxi_loc;
        for (mini_loc = 0; mini_loc < 256 && hloc[mini_loc] == 0; mini_loc++);
        for (maxi_loc = 255; maxi_loc >= 0 && hloc[maxi_loc] == 0; maxi_loc--);

        #pragma omp critical
        {
            if (mini > mini_loc) mini = mini_loc;
            if (maxi < maxi_loc) maxi = maxi_loc;
        }

        int q = 255 / (maxi - mini);
        #pragma omp for
        for (int i = 0; i < npixels; i++)
            pixels[i] = (pixels[i] - mini) * q;
        free(hloc);
    }
    free(h);
}
```

Локальная таблица hloc[] в каждом потоке

Синхронизация при поиске глобальных I_{min} и I_{max}

$$\mathrm{LUT}[i] = 255 \cdot \frac{i - I_{\min}}{I_{\max} - I_{\min}}.$$

#pragma omp sections

```
#pragma omp parallel
{
        #pragma omp sections
        {
                #pragma omp section
                {
                        // Section 1
                }

                #pragma omp section
                {
                        // Section 2
                }

                #pragma omp section
                {
                        // Section 3
                }

        } // barrier
}
```

Порождает **пул потоков** (team of threads)
и **набор задач** (set of tasks)

Код каждой секции выполняется одним потоком
(в контексте задачи)

NSECTIONS > NTHREADS
Не гарантируется, что все секции будут выполняться разными потоками
Один поток может выполнить несколько секций

#pragma omp sections

```
#pragma omp parallel num_threads(3)
{
    #pragma omp sections
    {
        // Section directive is optional for the first structured block
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 0: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 1: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 2: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 3: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
    }
}
```

3 потока, 4 секции

#pragma omp sections

```
#pragma omp parallel num_threads(3)
{
    #pragma omp sections
    {
        // Section directive is optional for the first structured block
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 0: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 1: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 2: thread %d / %d\n", omp_get_thread_num(),
        }
        #pragma omp section
        {
            sleep_rand_ns(100000, 200000);
            printf("Section 3: thread %d / %d\n", omp_get_thread_num(),
        }
    }
}
```

3 потока, 4 секции

```
$ ./sections
Section 1: thread 1 / 3
Section 0: thread 0 / 3
Section 2: thread 2 / 3
Section 3: thread 1 / 3
```

Ручное распределение задач по потокам

```
#pragma omp parallel num_threads(3)
{
    int tid = omp_get_thread_num();

    switch (tid) {

        case 0:
            sleep_rand_ns(100000, 200000);
            printf("Section 0: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
            break;

        case 1:
            sleep_rand_ns(100000, 200000);
            printf("Section 1: thread %d / %d\n", omp_get_thread_num(), omp_get_num_threads());
            break;

        case 2:
            sleep_rand_ns(100000, 200000);
            printf("Section 3: thread %d / %d\n", omp_get_thread_num(),
            break;

        default:
            fprintf(stderr, "Error: TID > 2\n");
    }
}
```

3 потока, 3 блока

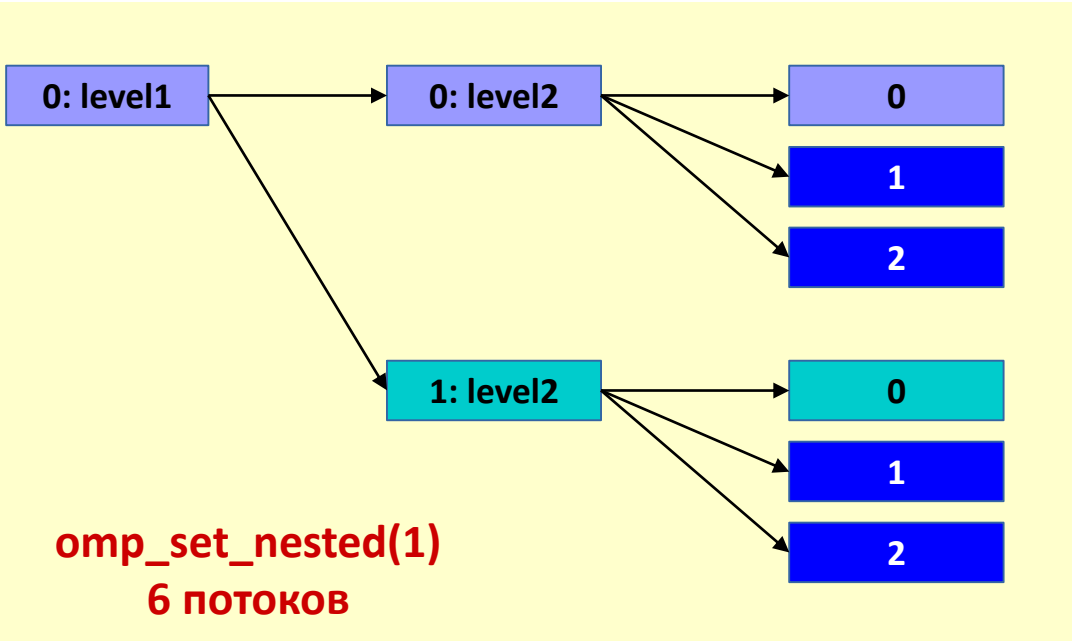
```
$ ./sections
Section 3: thread 2 / 3
Section 1: thread 1 / 3
Section 0: thread 0 / 3
```

Вложенные параллельные регионы (nested parallelism)

```
void level2(int parent)
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp critical
        printf("L2: parent %d, thread %d / %d, level %d (nested regions %d)\n",
               parent, omp_get_thread_num(), omp_get_num_threads(), omp_get_active_level(), omp_get_level());
    }
}

void level1()
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp critical
        printf("L1: thread %d / %d, level %d (nested regions %d)\n",
               omp_get_thread_num(), omp_get_num_threads(), omp_get_active_level(), omp_get_level());

        level2(omp_get_thread_num());
    }
}

int main(int argc, char **argv)
{
    omp_set_nested(1);
    level1();
    return 0;
}
```

Вложенные параллельные регионы (nested parallelism)

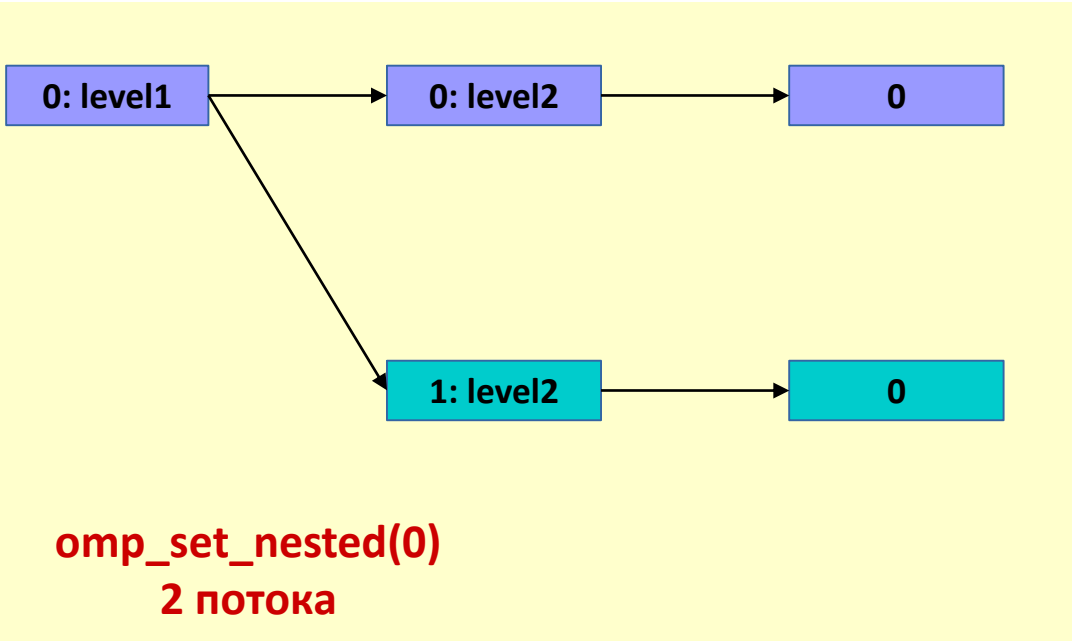
```
void level2(int parent)
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp critical
        printf("L2: parent %d, thread %d / %d, level %d (nested
               parent, omp_get_thread_num(), omp_get_num_thread
    }
}

void level1()
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp critical
        printf("L1: thread %d / %d, level %d (nested regions %d)\n",
               omp_get_thread_num(), omp_get_num_threads(), omp_get_active_level(), omp_get_level());

        level2(omp_get_thread_num());
    }
}

int main(int argc, char **argv)
{
    omp_set_nested(1);
    level1();
    return 0;
}
```




```
$ ./nested
L1: thread 0 / 2, level 1 (nested regions 1)
L1: thread 1 / 2, level 1 (nested regions 1)
L2: parent 0, thread 0 / 3, level 2 (nested regions 2)
L2: parent 0, thread 1 / 3, level 2 (nested regions 2)
L2: parent 0, thread 2 / 3, level 2 (nested regions 2)
L2: parent 1, thread 0 / 3, level 2 (nested regions 2)
L2: parent 1, thread 1 / 3, level 2 (nested regions 2)
L2: parent 1, thread 2 / 3, level 2 (nested regions 2)
```

Вложенные параллельные регионы (nested parallelism)

```
void level2(int parent)
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp critical
        printf("L2: parent %d, thread %d / %d, level %d (nested
               parent, omp_get_thread_num(), omp_get_num_thread
    }
}

void level1()
{
    #pragma omp parallel num_threads(2)
    {
        #pragma omp critical
        printf("L1: thread %d / %d, level %d (nested regions %d)\n",
               omp_get_thread_num(), omp_get_num_threads(), omp_get_active_level(), omp_get_level());

        level2(omp_get_thread_num());
    }
}

int main(int argc, char **argv)
{
    omp_set_nested(0);
    level1();
    return 0;
}
```



omp_set_nested(0)
2 потока

```
$ ./nested
L1: thread 0 / 2, level 1 (nested regions 1)
L1: thread 1 / 2, level 1 (nested regions 1)
L2: parent 0, thread 0 / 1, level 1 (nested regions 2)
L2: parent 1, thread 0 / 1, level 1 (nested regions 2)
```

Ограничение глубины вложенного параллелизма

```
void level3(int parent) {
    #pragma omp parallel num_threads(2)
    {
        // omp_get_active_level() == 2, omp_get_level() == 3
    }
}

void level2(int parent) {
    #pragma omp parallel num_threads(3)
    {
        // omp_get_active_level() == 2
        level3(omp_get_thread_num());
    }
}

void level1() {
    #pragma omp parallel num_threads(2)
    {
        // omp_get_active_level() == 1
        level2(omp_get_thread_num());
    }
}

int main(int argc, char **argv) {
    omp_set_nested(1);
    omp_set_max_active_levels(2);
    level1();
}
```

При создании параллельного региона runtime-система проверяет глубину вложенности параллельных регионов

omp_set_max_active_levels(*N*)

Если глубина превышена,
то параллельный регион будет содержать один поток

Определение числа потоков

```
#pragma omp parallel num_threads(n)
// code
```

- **OMP_THREAD_LIMIT** — максимальное число потоков в программе
- **OMP_NESTED** — разрешает/запрещает вложенный параллелизм
- **OMP_DYNAMIC** — разрешает/запрещает динамическое управление числом потоков в параллельном регионе
- **ActiveParRegions** — число активных вложенных параллельных регионов
- **ThreadsBusy** — число уже выполняющихся потоков
- **ThreadRequested** = num_threads либо OMP_NUM_THREADS

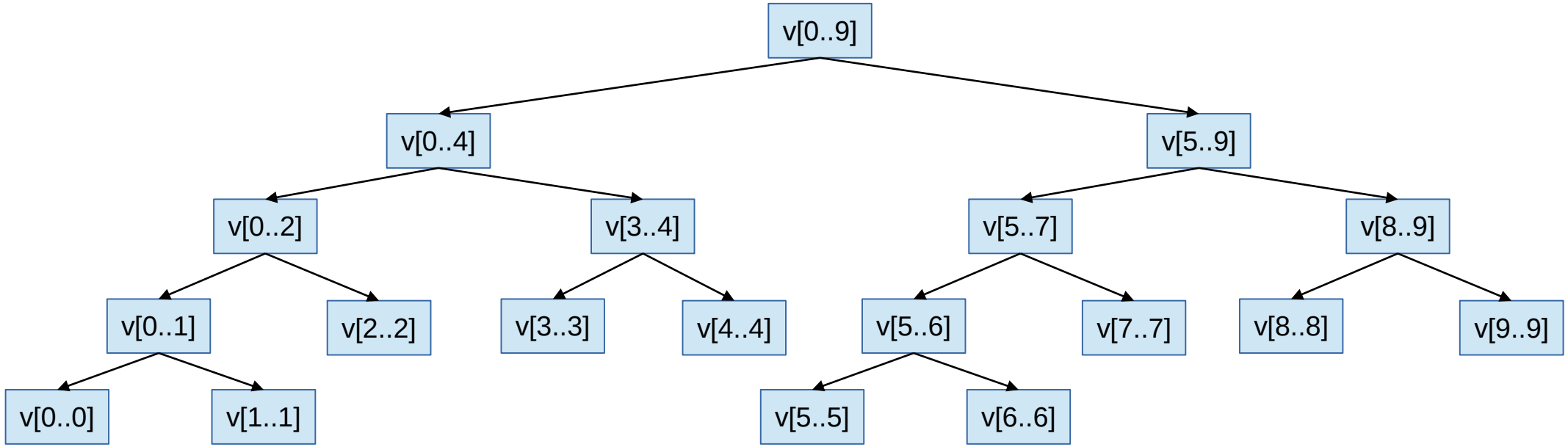
Алгоритм

```
ThreadsAvailable = OMP_THREAD_LIMIT – ThreadsBusy + 1
if ActiveParRegions >= 1 and OMP_NESTED = false then
    nthreads = 1
else if ActiveParRegions == OMP_MAX_ACTIVE_LEVELS then
    nthreads = 1
else if OMP_DYNAMIC and ThreadsRequested <= ThreadsAvailable then
    nthreads = [1 : ThreadsRequested] // выбирается runtime-системой
else if OMP_DYNAMIC and ThreadsRequested > ThreadsAvailable then
    nthreads = [1 : ThreadsAvailable] // выбирается runtime-системой
else if OMP_DYNAMIC = false and ThreadsRequested <= ThreadsAvailable then
    nthreads = ThreadsRequested
else if OMP_DYNAMIC = false and ThreadsRequested > ThreadsAvailable then
    // число потоков определяется реализацией
end if
```

Рекурсивное суммирование

```
double sum(double *v, int low, int high)
{
    if (low == high)
        return v[low];

    int mid = (low + high) / 2;
    return sum(v, low, mid) + sum(v, mid + 1, high);
}
```

5	10	15	20	25	30	35	40	45	50

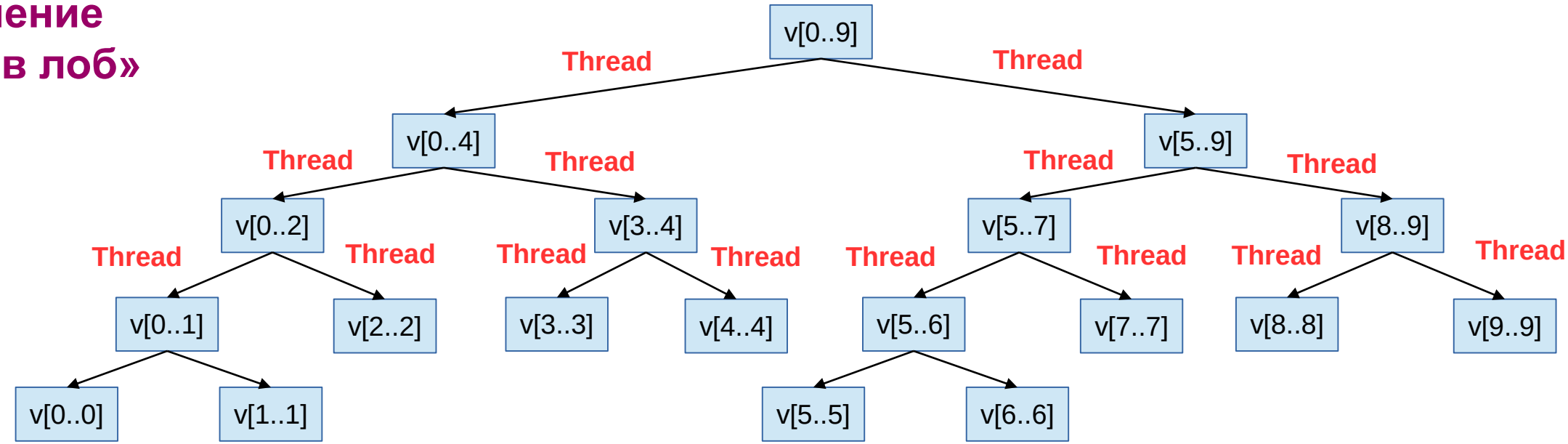


Параллельное рекурсивное суммирование

```
double sum(double *v, int low, int high)
{
    if (low == high)
        return v[low];

    int mid = (low + high) / 2;
    return sum(v, low, mid) + sum(v, mid + 1, high);
}
```

5	10	15	20	25	30	35	40	45	50

Решение «в лоб»



Решение «в лоб»

```
double sum_omp(double *v, int low, int high){
    if (low == high) return v[low];

    double sum_left, sum_right;
    #pragma omp parallel num_threads(2)
    {
        int mid = (low + high) / 2;
        #pragma omp sections
        {
            #pragma omp section
            {
                sum_left = sum_omp(v, low, mid);
            }

            #pragma omp section
            {
                sum_right = sum_omp(v, mid + 1, high);
            }
        }
    }
    return sum_left + sum_right;
}

double run_parallel()
{
    omp_set_nested(1);
    double res = sum_omp(v, 0, N - 1);
}
```

Решение «в лоб»

```
double sum_omp(double *v, int low, int high){
    if (low == high) return v[low];

    double sum_left, sum_right;
    #pragma omp parallel num_threads(2)
    {
        int mid = (low + high) / 2;
        #pragma omp sections
        {
            #pragma omp section
            {
                sum_left = sum_omp(v, low, mid);
            }

            #pragma omp section
            {
                sum_right = sum_omp(v, mid + 1, high);
            }
        }
    }
    return sum_left + sum_right;
}

double run_parallel()
{
    omp_set_nested(1);
    double res = sum_omp(v, 0, N - 1);
}
```

```
# N = 100000
$ ./sum
libgomp: Thread creation failed: Resource temporarily unavailable
```

Глубина вложенных параллельных регионов не ограничена (создается очень много потоков)

На хватает ресурсов для поддержания пула потоков

Ограничение глубины вложенного параллелизма

```
double sum_omp(double *v, int low, int high) {
    if (low == high) return v[low];

    double sum_left, sum_right;
    #pragma omp parallel num_threads(2)
    {
        int mid = (low + high) / 2;
        #pragma omp sections
        {
            #pragma omp section
            {
                sum_left = sum_omp(v, low, mid);
            }

            #pragma omp section
            {
                sum_right = sum_omp(v, mid + 1, high);
            }
        }
    }
    return sum_left + sum_right;
}

double run_parallel() {
    omp_set_nested(1);
    omp_set_max_active_levels(ilog2(4)); // 2 уровня
    double res = sum_omp(v, 0, N - 1);
}
```

Привяжем глубину вложенных параллельных регионов к числу доступных процессорных ядер

2 потока (процессора) — глубина 1
4 потока — глубина 2
8 потоков — глубина 3
...
n потоков — глубина $\log_2(n)$

Ограничение глубины вложенного параллелизма

```
double sum_omp(double *v, int low, int high) {
    if (low == high) return v[low];

    double sum_left, sum_right;
    #pragma omp parallel num_threads(2)
    {
        int mid = (low + high) / 2;
        #pragma omp sections
        {
            #pragma omp section
            {
                sum_left = sum_omp(v, low, mid);
            }

            #pragma omp section
            {
                sum_right = sum_omp(v, mid + 1, high);
            }
        }
    }
    return sum_left + sum_right;
}

double run_parallel() {
    omp_set_nested(1);
    omp_set_max_active_levels(ilog2(4)); //
    double res = sum_omp(v, 0, N - 1);
}
```

Привяжем глубину вложенных параллельных регионов к числу доступных процессорных ядер

2 потока (процессора) — глубина 1
4 потока — глубина 2
8 потоков — глубина 3
...
n потоков — глубина $\log_2(n)$

```
Recursive summation N = 100000000
Result (serial): 5000000050000000.0000; error 0.000000000000
Parallel version: max_threads = 8, max_levels = 3
Result (parallel): 5000000050000000.0000; error 0.000000000000
Execution time (serial): 0.798292
Execution time (parallel): 20.302973
Speedup: 0.04
```

Сокращение активаций параллельных регионов

```
double sum_omp_fixed_depth(double *v, int low, int high)
{
    if (low == high)
        return v[low];

    double sum_left, sum_right;
    int mid = (low + high) / 2;

    if (omp_get_active_level() >= omp_get_max_active_levels())
        return sum_omp_fixed_depth(v, low, mid) + sum_omp_fixed_depth(v, mid + 1, high);

    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections
        {
            #pragma omp section
            sum_left = sum_omp_fixed_depth(v, low, mid);

            #pragma omp section
            sum_right = sum_omp_fixed_depth(v, mid + 1, high);
        }
    }
    return sum_left + sum_right;
}
```

Ручная проверка глубины
При достижении предельной глубины избегаем активации параллельного региона

Сокращение активаций параллельных регионов

```
double sum_omp_fixed_depth(double *v, int low, int high)
{
    if (low == high)
        return v[low];

    double sum_left, sum_right;
    int mid = (low + high) / 2;

    if (omp_get_active_level() >= omp_get_max_active_levels())
        return sum_omp_fixed_depth(v, low, mid) + sum_omp_fixed_depth(v, mid + 1, high);

    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections
        {
            #pragma omp section
            sum_left = sum_omp_fixed_depth(v, low, mid);

            #pragma omp section
            sum_right = sum_omp_fixed_depth(v, mid + 1, high);
        }
    }
    return sum_left + sum_right;
}
```

Секции могут выполняться одним и тем же потоком

Привяжем секции к разным потокам

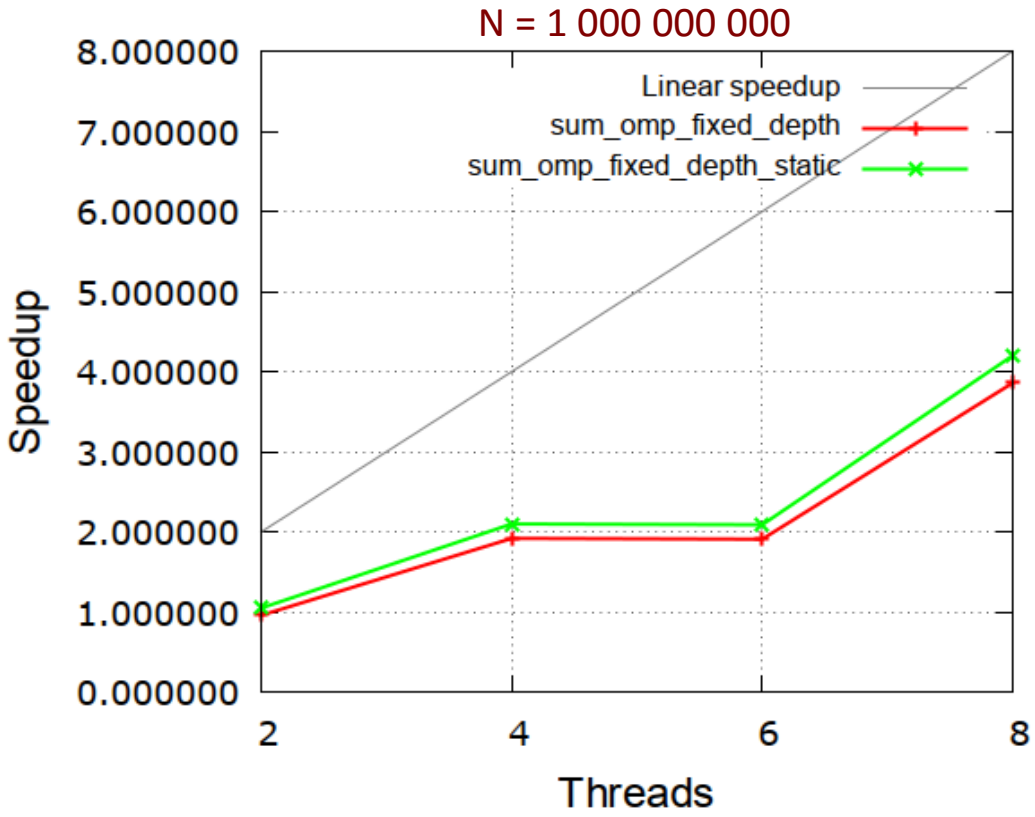
Рекурсивные вызовы в разных потоках

```
double sum_omp_fixed_depth_static(double *v, int low, int high)
{
    if (low == high)
        return v[low];

    double sum_left, sum_right;
    int mid = (low + high) / 2;

    if (omp_get_active_level() >= omp_get_max_active_levels())
        return sum_omp_fixed_depth_static(v, low, mid) +
               sum_omp_fixed_depth_static(v, mid + 1, high);

    #pragma omp parallel num_threads(2)
    {
        int tid = omp_get_thread_num();
        if (tid == 0) {
            sum_left = sum_omp_fixed_depth_static(v, low, mid);
        } else if (tid == 1) {
            sum_right = sum_omp_fixed_depth_static(v, mid + 1, high);
        }
    }
    return sum_left + sum_right;
}
```

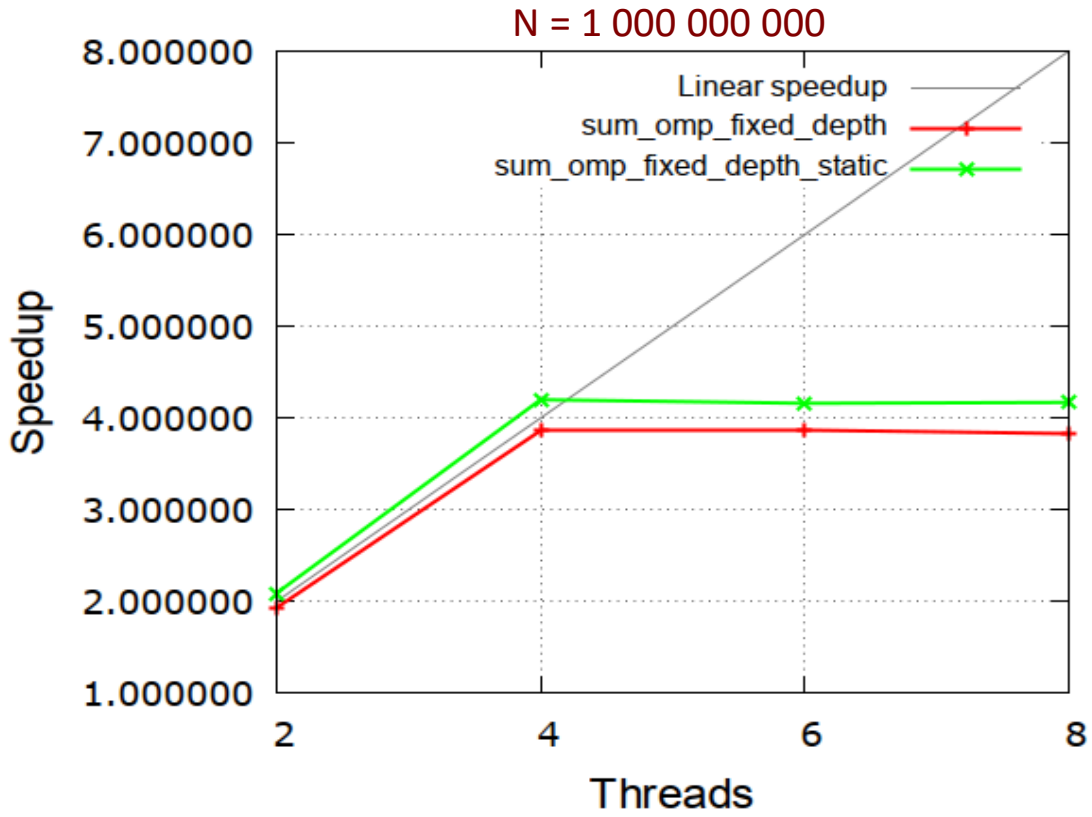
1. Ограничили глубину рекурсивных вызовов

2. Привязали «секции» к разным потокам

Анализ эффективности (кластер Oak)



omp_set_max_active_levels(log2(nthreads))



omp_set_max_active_levels(log2(nthreads) + 1)