

Лекция 10

Стандарт OpenMP: параллелизм задач

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Параллельные вычислительные технологии»

Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

#pragma omp sections

```
#pragma omp parallel  
{
```

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

```
    #pragma omp sections  
    {
```

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

```
        #pragma omp section  
        {  
            // Section 2  
        }
```

```
        #pragma omp section  
        {  
            // Section 3  
        }
```

```
    } // barrier
```

```
}
```

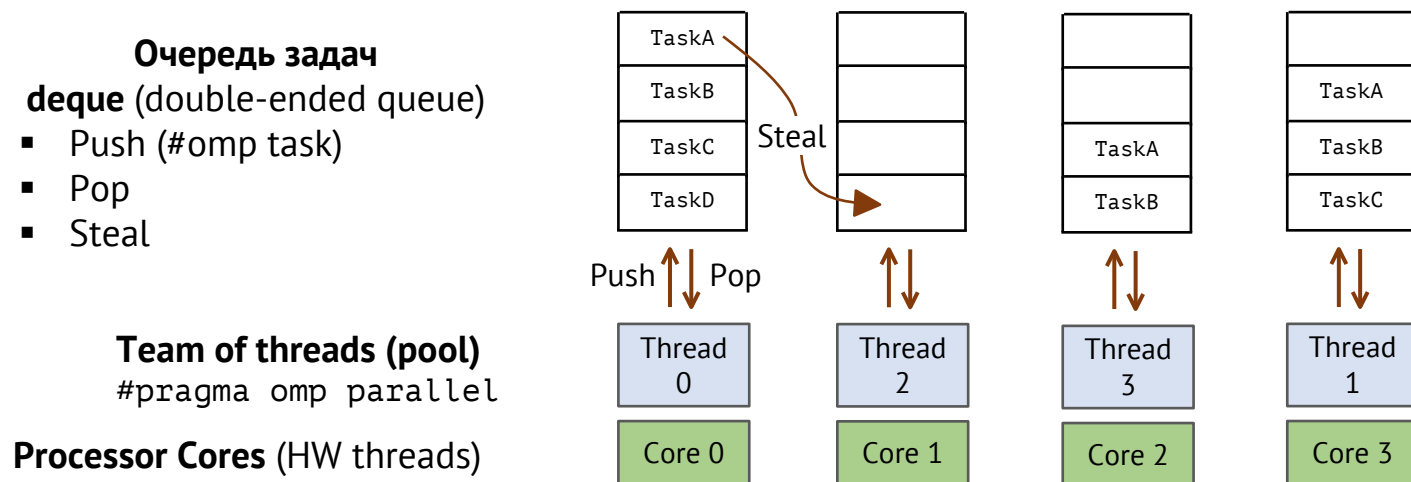
Код каждой секции выполняется одним потоком
(в контексте задачи)

$NSECTIONS > NTHREADS$

Не гарантируется, что все секции будут выполняться
разными потоками

Один поток может выполнить несколько секций

OpenMP Work Stealing – балансировка загрузки потоков



- На логических процессорах операционной системы (физических ядрах, аппаратных потоках) выполняется пул потоков (`num_threads()`, `OMP_NUM_THREADS`)
- Каждый поток имеет двустороннюю очередь (дек)
 - с нижнего конца поток добавляет новые задачи (`#pragma omp task, push`)
 - с нижнего конца поток извлекает задачи для выполнения (`pop`)
- Если у потока закончились задачи в очереди: генерирует псевдослучайный номер потока и извлекает с верхнего конца его очереди задачу (`steal`)

#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(), 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

```
$ ./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(), omp_get_num_threads());
            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;
}
```

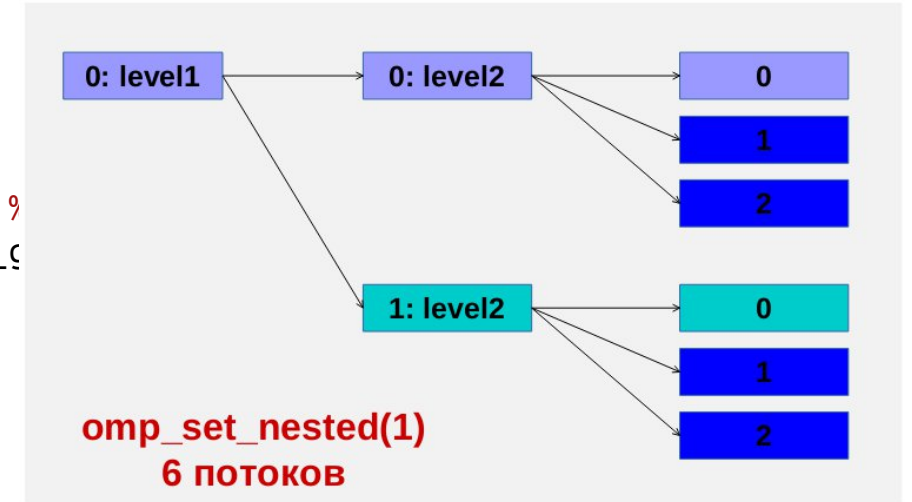
Вложенные параллельные регионы (nested parallelism)

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



```
$ ./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)
```

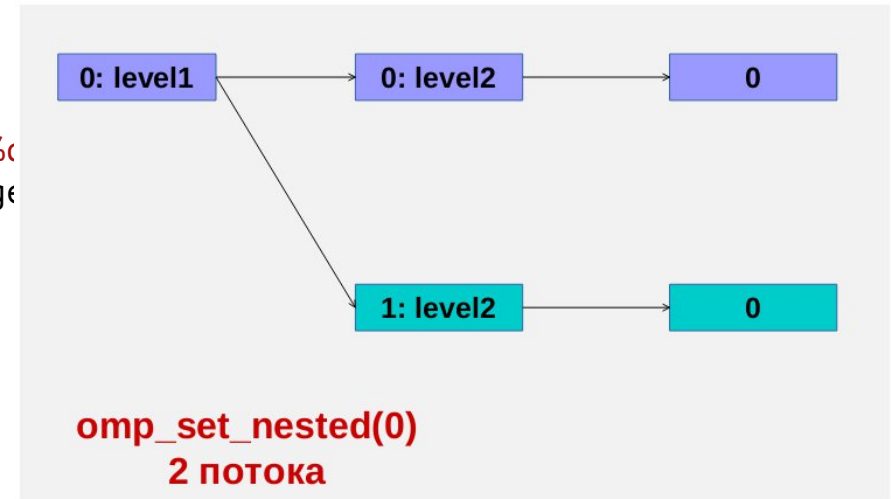

Вложенные параллельные регионы (nested parallelism)

```
void level2(int parent)
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp critical
        printf("L2: parent %d, thread %d / %d, level %d (nested regions %d)\n",
            parent, omp_get_thread_num(), omp_get_num_threads(), omp_get_active_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(0);
    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 / 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

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();
}
```

Максимальная глубина вложенности
параллельных
регионов равна 2

`omp_set_max_active_levels(2)`

В параллельном регионе активен 1 поток —
поток, который вызвал функцию level 3

Всего потоков $2 * 3 = 6$

Определение числа потоков в регионе

```
#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

ThreadsRequested = num_threads
либо OMP_NUM_THREADS

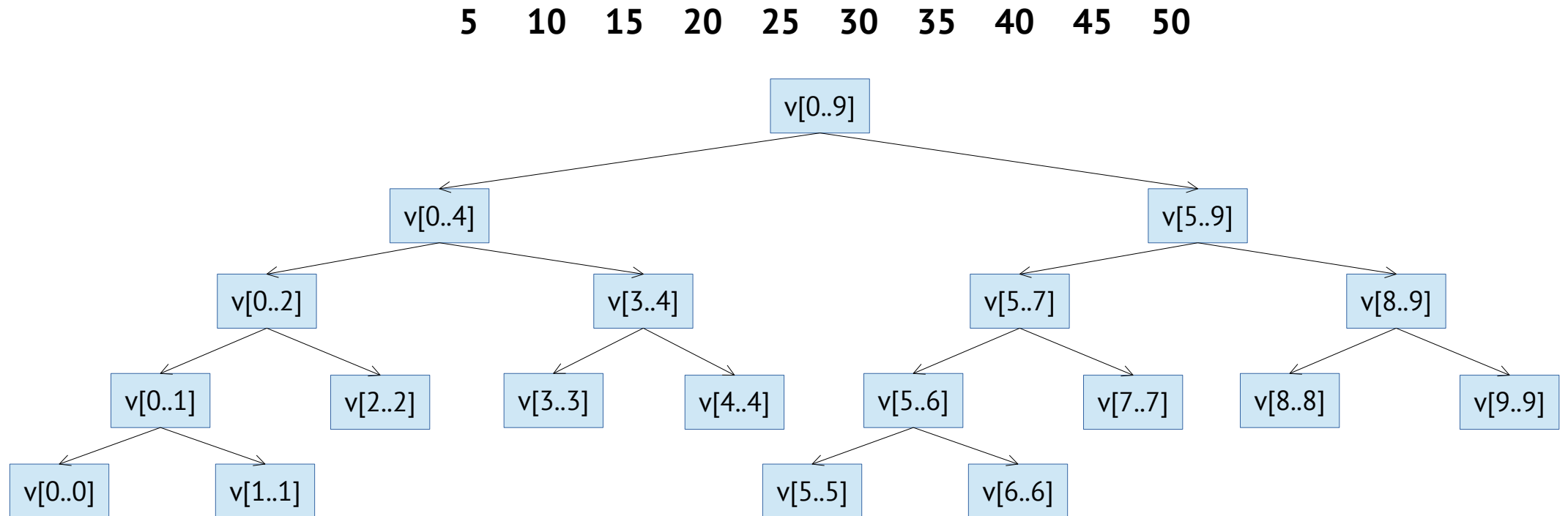
Алгоритм

```
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);
}
```



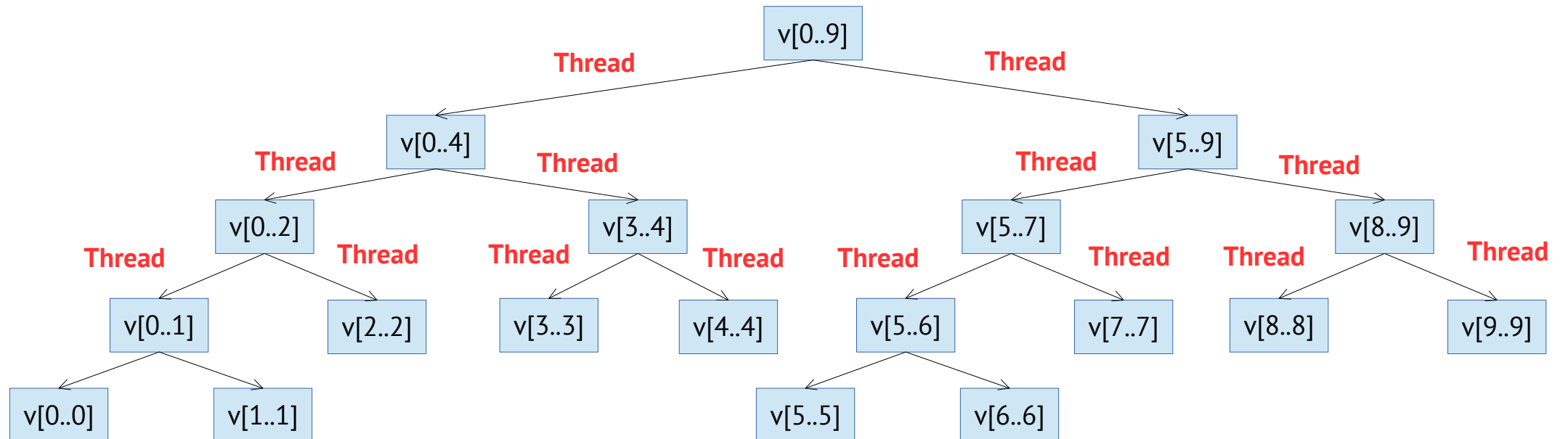
Параллельное рекурсивное суммирование

```
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);
}
```

"Наивное решение"
порождение потока на каждый
вызов sum()

5 10 15 20 25 30 35 40 45 50

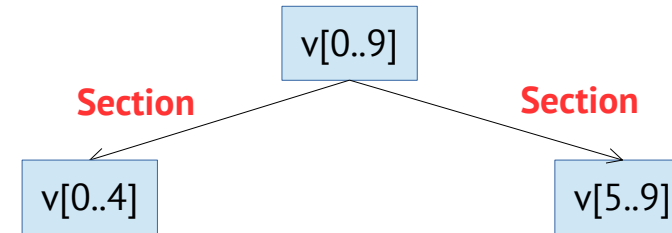


Выполнение рекурсивного вызова в отдельном потоке (задаче)

```
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);
            }
        }
    } // barrier
    return sum_left + sum_right;
}

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



Выполнение рекурсивного вызова в отдельном потоке (задаче)

```
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;
}
```

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

На хватает ресурсов для поддержания пула потоков

```
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
```


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**

Привязать 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);

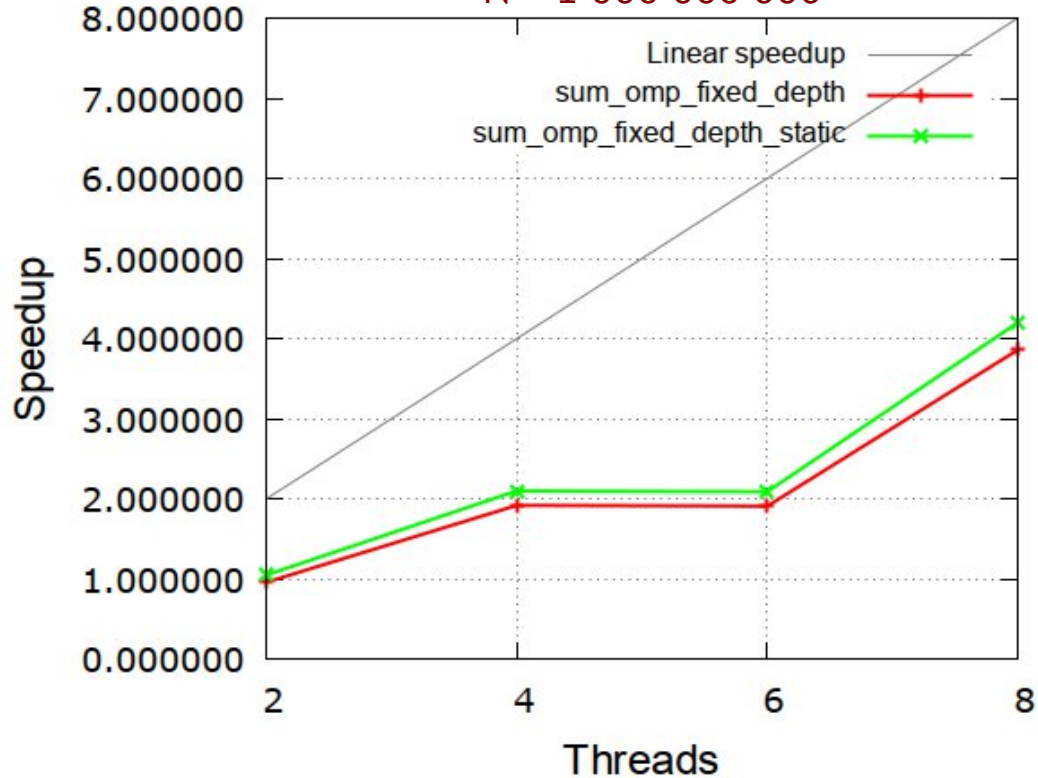
    #pragma omp parallel num_threads(2)
    {
        int tid = omp_get_thread_num();
        if (tid == 0) {
            sum_left = sum_omp_fixed_depth_static(v, low, mid);
        } else if (tid == 1) {
            sum_right = sum_omp_fixed_depth_static(v, mid + 1, high);
        }
    }
    return sum_left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

1. Ограничили глубину рекурсивных вызовов

2. Привязали «секции» к разным потокам

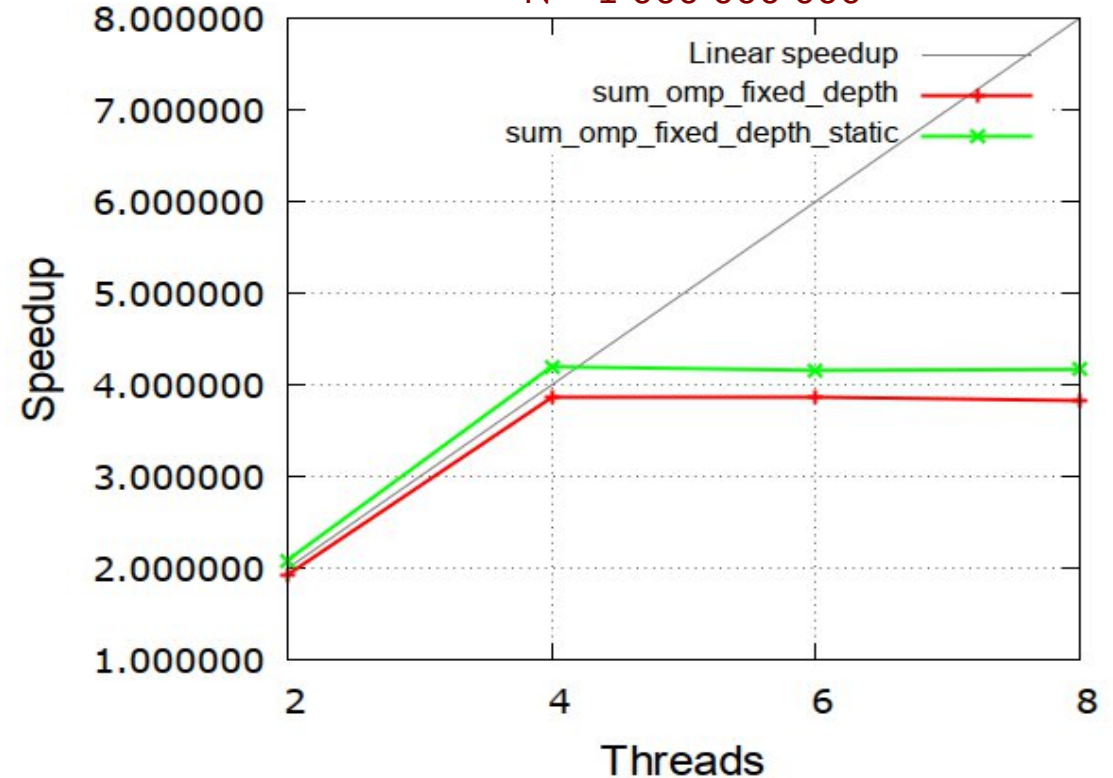
Анализ эффективности (2 x Intel Xeon Quad, NUMA)

N = 1 000 000 000



`omp_set_max_active_levels(log2(nthreads))`
levels: 1, 2, 2, 3

N = 1 000 000 000

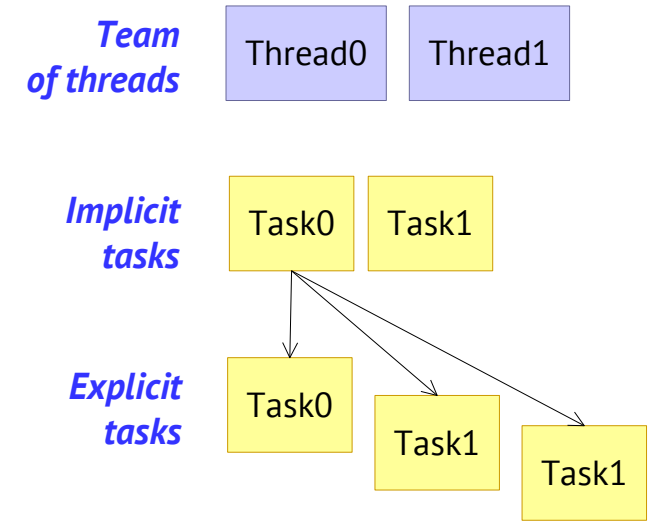


`omp_set_max_active_levels(log2(nthreads) + 1)`
levels: 2, 3, 3, 4

Параллелизм задач (task parallelism, \geq OpenMP 3.0)

```
void fun()
{
    int a, b;
    #pragma omp parallel num_threads(2) shared(a) private(b)
    {
        #pragma omp single nowait
        {
            for (int i = 0; i < 3; i++) {
                #pragma omp task default(firstprivate)
                {
                    int c;
                    // A - shared, B - firstprivate, C - private
                }
            }
        }
    }
}

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



Параллелизм задач

Параллельная обработка динамических структур данных (связные списки, деревья, ...)

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (; list != NULL; list = list->next) {
            #pragma omp task firstprivate(list)
            {
                process_node(list);
            }
        }
    }
}
```

```
void postorder(node *p)
{
    if (p->left) {
        #pragma omp task
        postorder(p->left);
    }

    if (p->right) {
        #pragma omp task
        postorder(p->right);
    }

    #pragma omp taskwait
    process(p->data);
}
```


Параллельное рекурсивное суммирование (v1)

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

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

    #pragma omp task shared(sum_left)
    sum_left = sum_omp_tasks(v, low, mid);

    #pragma omp task shared(sum_right)
    sum_right = sum_omp_tasks(v, mid + 1, high);

    #pragma omp taskwait
    return sum_left + sum_right;
}

double sum_omp(double *v, int low, int high)
{
    double s = 0;
    #pragma omp parallel
    {
        #pragma omp single nowait
        s = sum_omp_tasks(v, low, high);
    }
    return s;
}
```

#omp task

Отдельная задача для каждого рекурсивного вызова

#omp taskwait

Ожидание завершения дочерних задач

Пул из N потоков + N задач (implicit tasks)

Параллельное рекурсивное суммирование (v1)

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

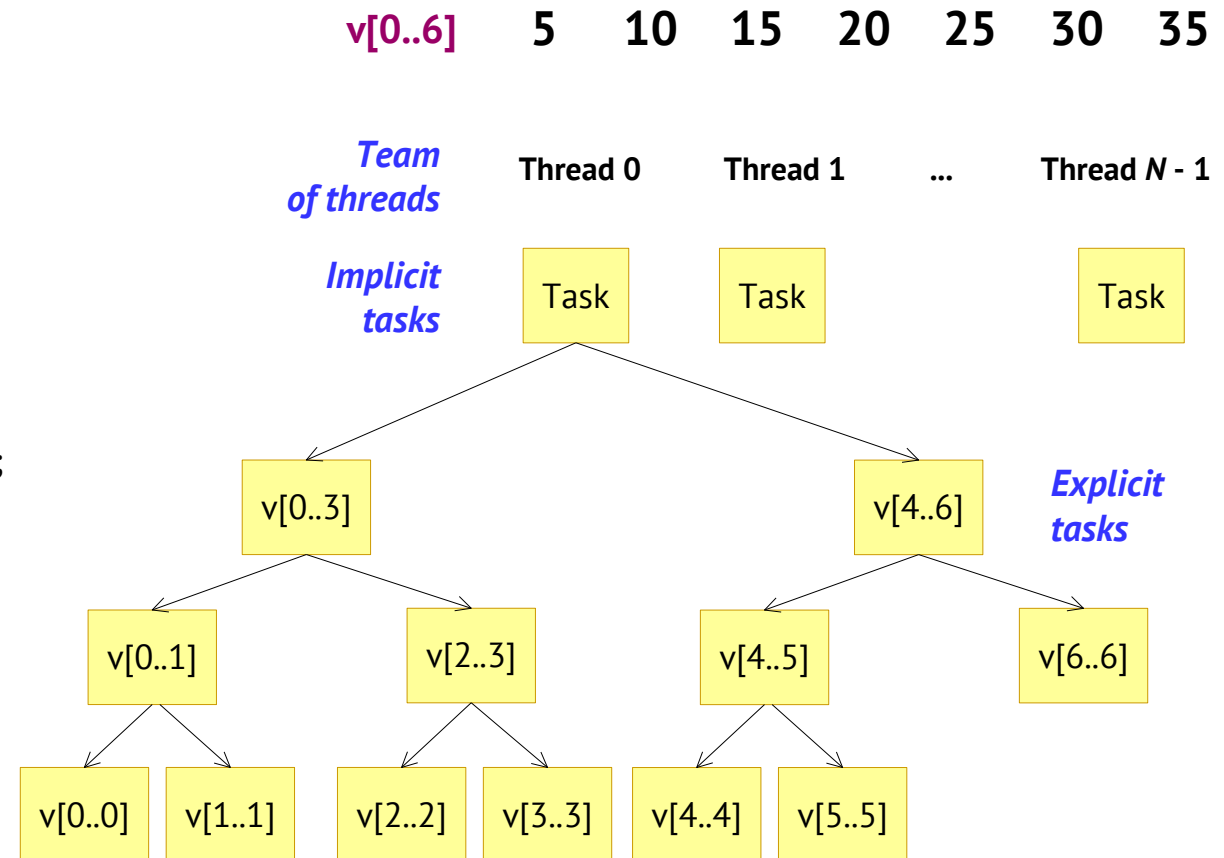
    double sum_left, sum_right;
    int mid = (low + high) / 2;

    #pragma omp task shared(sum_left)
    sum_left = sum_omp_tasks(v, low, mid);

    #pragma omp task shared(sum_right)
    sum_right = sum_omp_tasks(v, mid + 1, high);

    #pragma omp taskwait
    return sum_left + sum_right;
}

double sum_omp(double *v, int low, int high)
{
    double s = 0;
    #pragma omp parallel
    {
        #pragma omp single nowait
        s = sum_omp_tasks(v, low, high);
    }
    return s;
}
```



- Создается большое количество задач
- Значительные накладные расходы

Параллельное рекурсивное суммирование (v2)

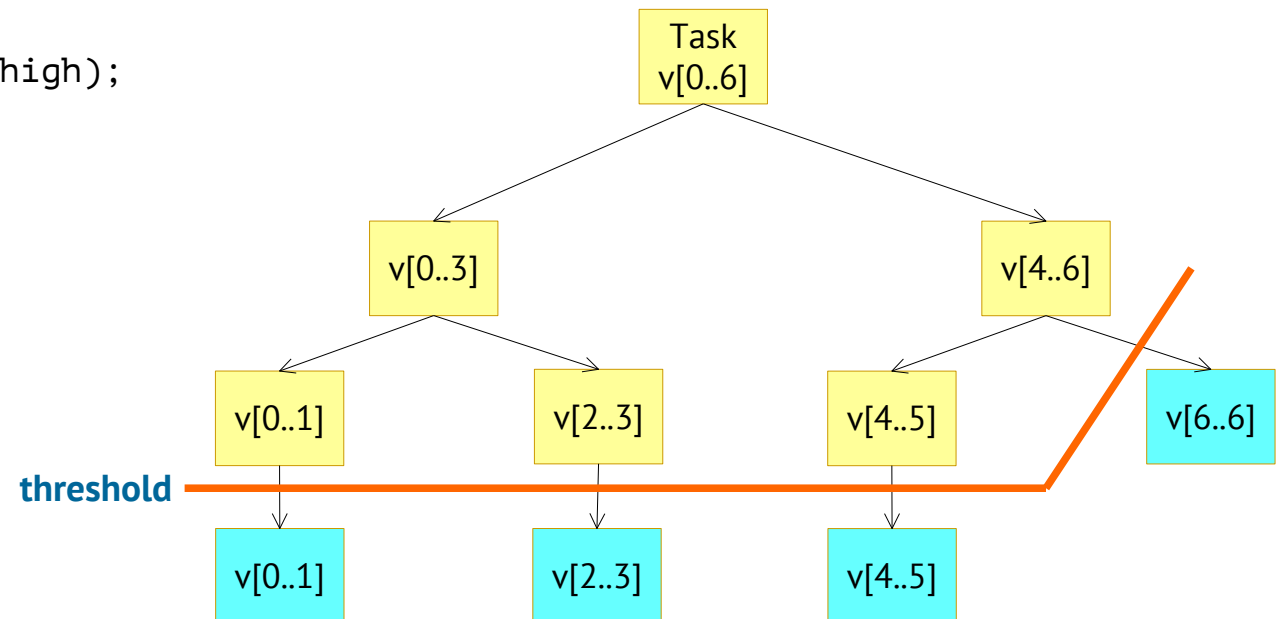
```
double sum_omp_tasks_threshold(double *v, int low, int high)
{
    if (low == high)
        return v[low];
    if (high - low < SUM_OMP_ARRAY_MIN_SIZE)
        return sum(v, low, high);

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

    #pragma omp task shared(sum_left)
    sum_left = sum_omp_tasks_threshold(v, low, mid);
    #pragma omp task shared(sum_right)
    sum_right = sum_omp_tasks_threshold(v, mid + 1, high);
    #pragma omp taskwait
    return sum_left + sum_right;
}
```

```
double sum_omp(double *v, int low, int high)
{
    double s = 0;
    #pragma omp parallel
    {
        #pragma omp single nowait
        s = sum_omp_tasks_threshold(v, low, high);
    }
    return s;
}
```

Переключение на
последовательную версию
при достижении предельного
размера подмассива



Параллельное рекурсивное суммирование (v3)

```
double sum_omp_tasks_maxthreads(double *v, int low, int high, int nthreads)
{
    if (low == high)
        return v[low];
    if (nthreads <= 1)
        return sum(v, low, high);

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

    #pragma omp task shared(sum_left)
    sum_left = sum_omp_tasks_maxthreads(v, low, mid, nthreads / 2);
    #pragma omp task shared(sum_right)
    sum_right = sum_omp_tasks_maxthreads(v, mid + 1, high, nthreads - nthreads / 2);

    #pragma omp taskwait
    return sum_left + sum_right;
}

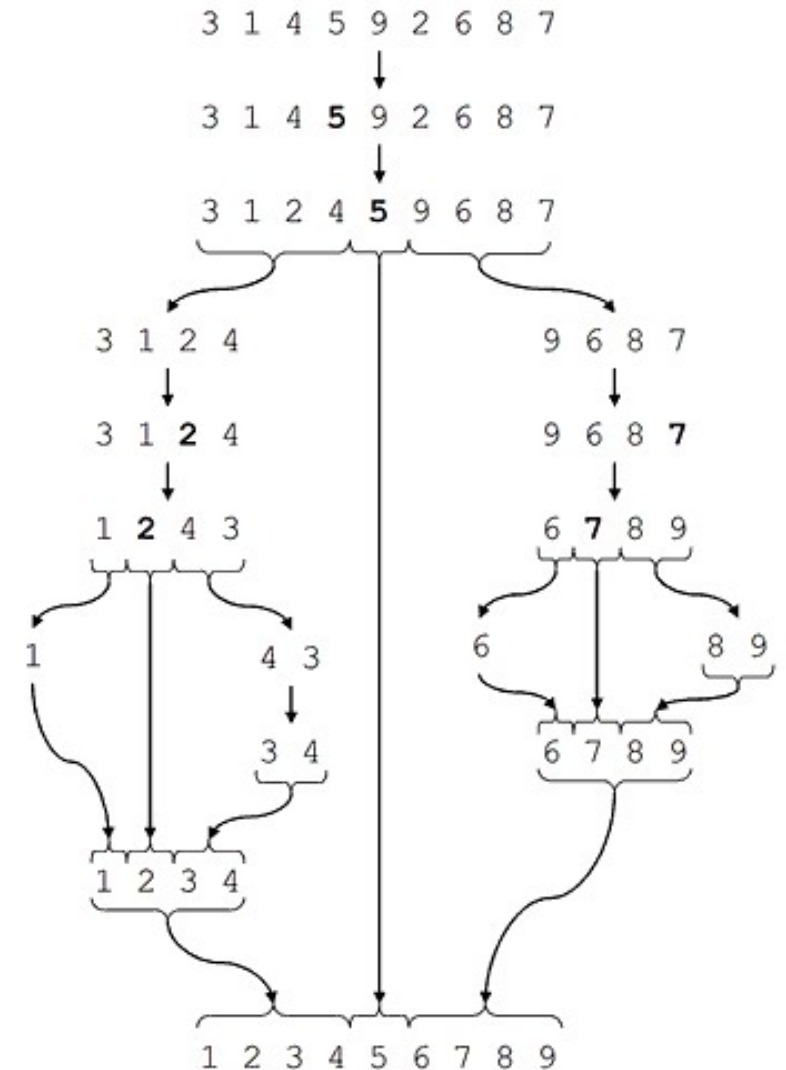
double sum_omp(double *v, int low, int high)
{
    double s = 0;
    #pragma omp parallel
    {
        #pragma omp single nowait
        s = sum_omp_tasks_maxthreads(v, low, high, omp_get_num_procs());
    }
    return s;
}
```

**Переключение на
последовательную версию
при достижении предельного
числа запущенных задач**

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 (v1): 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 (v2): 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 (v3): 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 (v5): 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 (v6): untied tasks

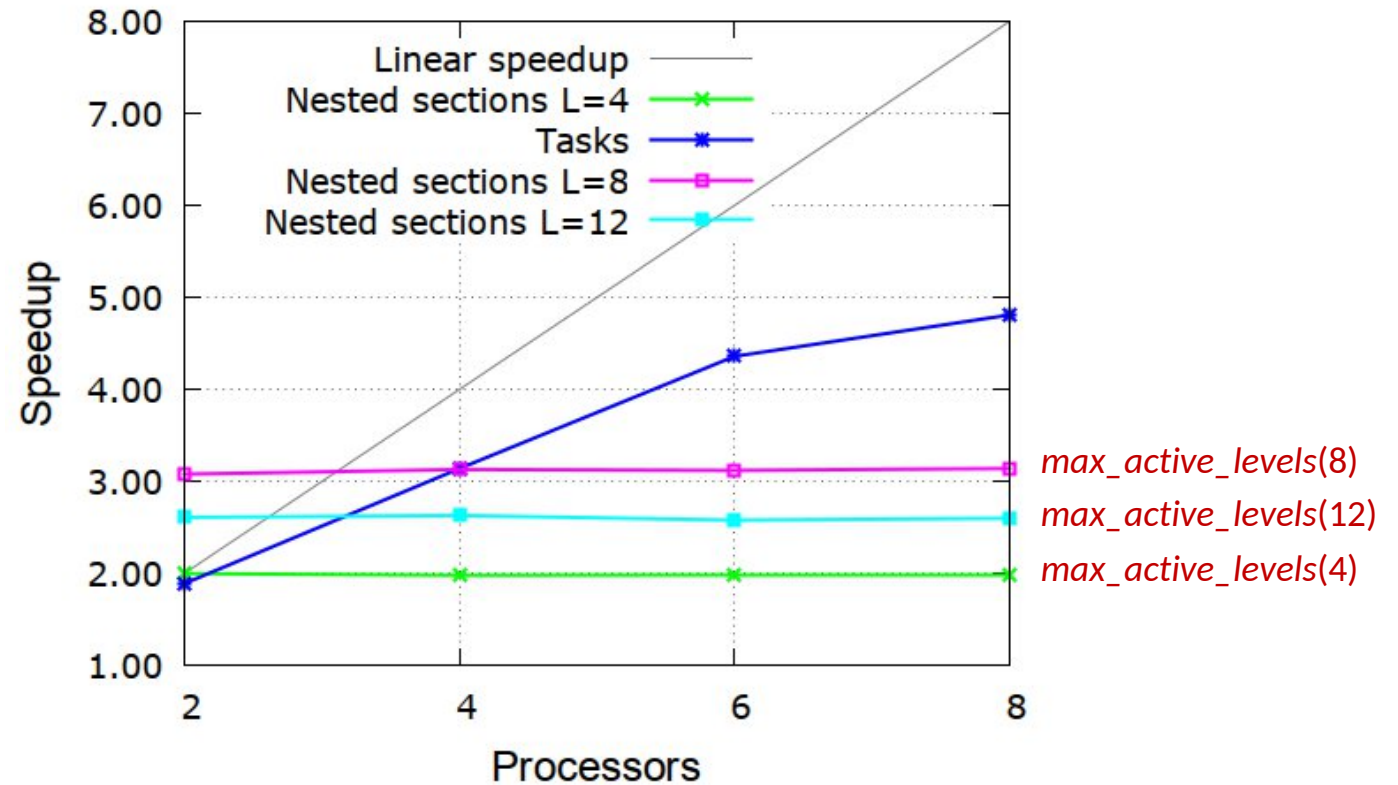
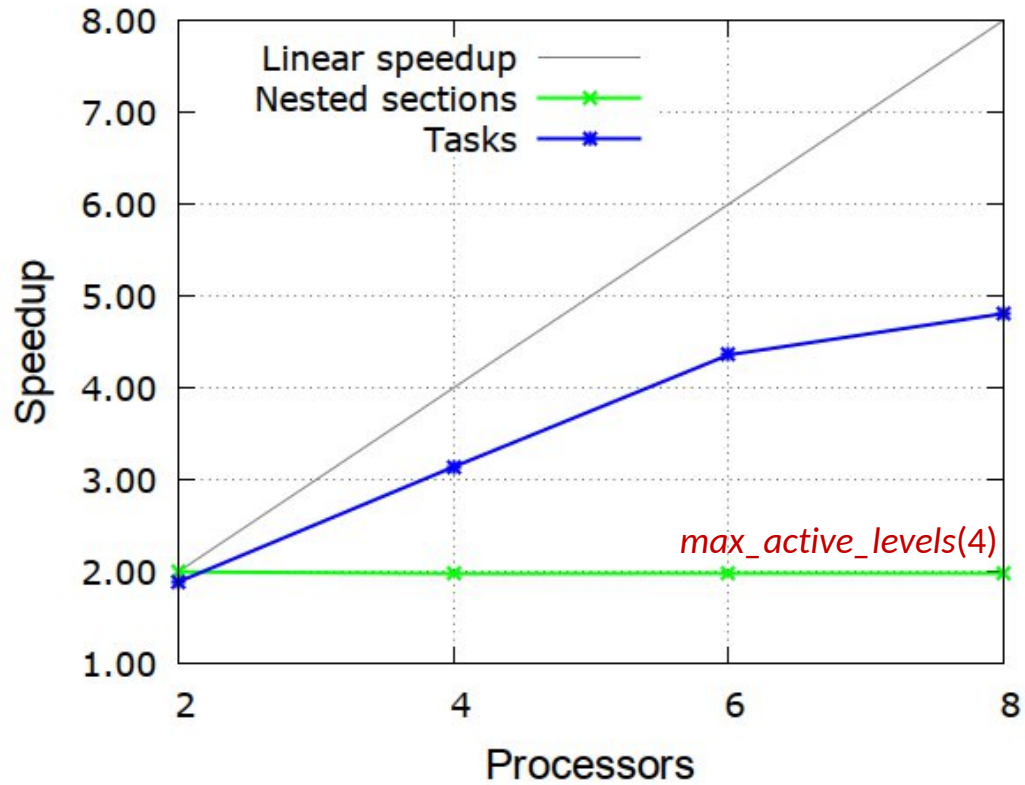
```
#pragma omp parallel
{
    #pragma omp single
    quicksort_tasks(array, 0, size - 1);
}
...

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

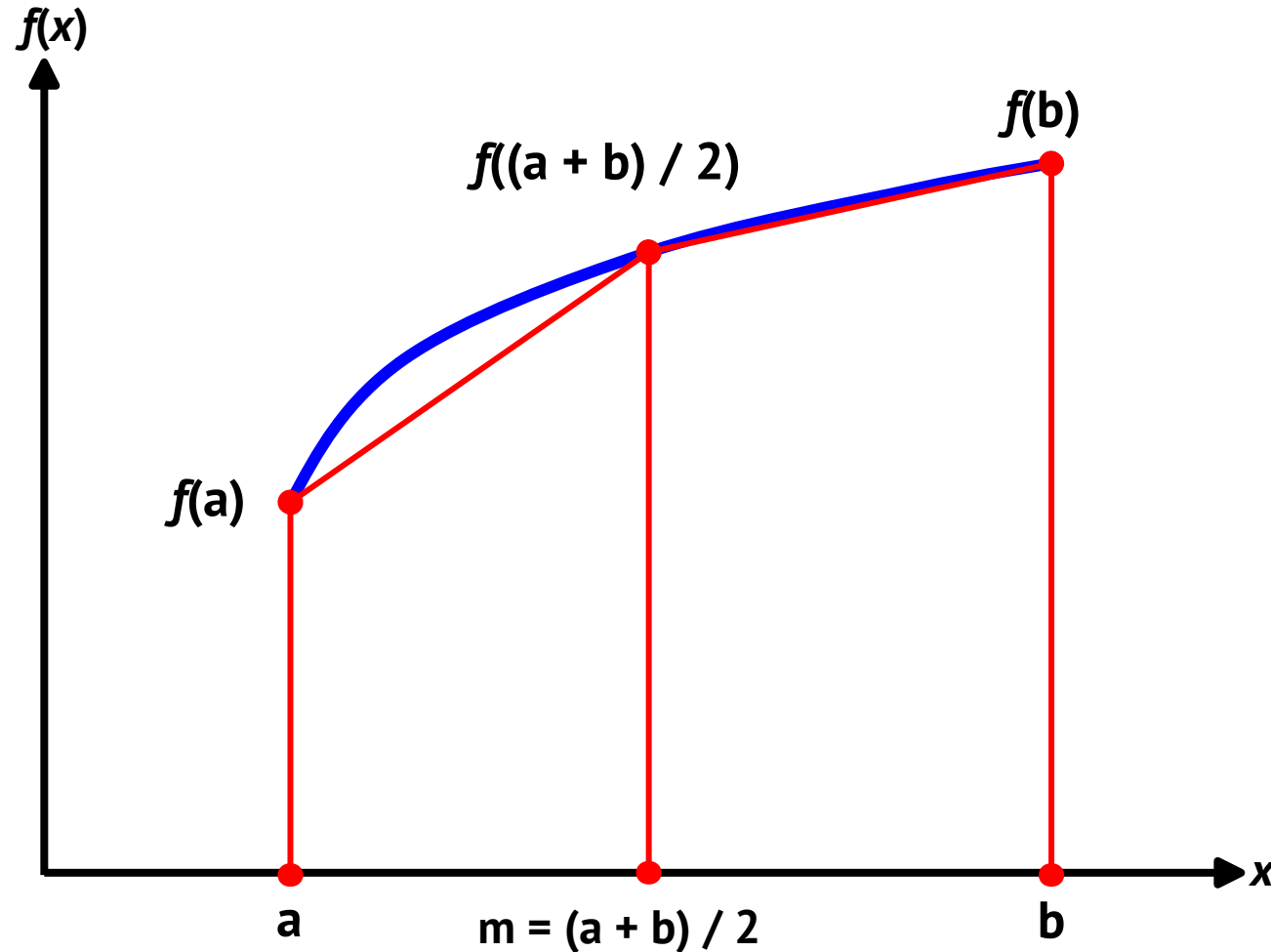
    if (high - low < threshold || (j - low < threshold || high - i < threshold)) {
        if (low < j)
            quicksort_tasks(v, low, j);
        if (i < high)
            quicksort_tasks(v, i, high);
    } else {
        #pragma omp task untied // Открепить задачу от потока (задачу может выполнять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

Быстрая сортировка (QuickSort)

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



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 (#omp 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 (#omp 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;
}
```

OpenM 4.0 – программирование GPU и 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 потоков к процессорам (thread affinity)

Выбор ядер для выполнения потоков

- `OMP_PLACES="cores | threads | sockets"`
- Places – абстракция "точек" привязки потоков (области, группы процессоров) (можно выбрать какие процессоры будут доступны программе)
- Пример: 8 областей (область – 4 логических процессора):
`OMP_PLACES=' {0:4} , {4:4} , {8:4} , {12:4} , {16:4} , {20:4} , {24:4} , {28:4} '`

Привязка потоков к областям (логическим процессорам)

- `OMP_PROC_BIND=false` – планировщик операционной системы может перемещать потоки между процессорами (OMP places)
- `OMP_PROC_BIND=true` – потоки привязываются к областям
- **#pragma omp parallel proc_bind**(master | close | spread) – задает схему распределения потоков по областям (процессорам)

OpenM 4.0 – привязка потоков к процессорам (thread affinity)

```
# {processor-id:list-length}
# 8 places
OMP_PLACES='{0:4},{4:4},{8:4},{12:4},{16:4},{20:4},{24:4},{28:4}'
OMP_NUM_THREADS=4
```

OMP_PROC_BIND=master

```
# Every thread in the team is assigned to the place on which the master executes
# Four threads are assigned to place 0.
```

OMP_PROC_BIND=close

```
# The thread is assigned to a place that is close to the place of the parent thread.
# The thread assignment is as follows:
#   OMP thread 0 is assigned to place 0
#   OMP thread 1 is assigned to place 1
#   OMP thread 2 is assigned to place 2
#   OMP thread 3 is assigned to place 3
```

OMP_PROC_BIND=spread

```
# The number of threads 4 is smaller than the number of places 8, so four subpartitions are formed.
# 8 is evenly divided by 4, so the thread assignment is as follows:
#   OMP thread 0 is assigned to place 0
#   OMP thread 1 is assigned to place 2
#   OMP thread 2 is assigned to place 4
#   OMP thread 3 is assigned to place 6
```