Семинар 6
Стандарт OpenMP (часть 6)

Михаил Курносов

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

Цикл семинаров «Основы параллельного программирования»
Институт физики полупроводников им. А. В. Ржанова СО РАН
Новосибирск, 2015

#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_nu
            break;

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

3 потока, 3 блока

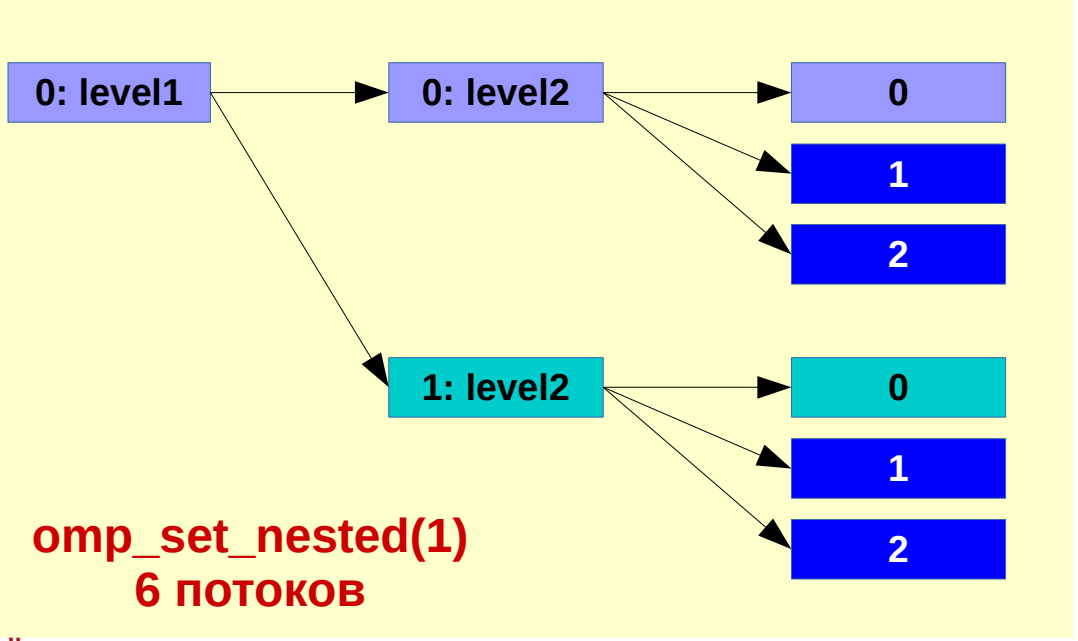
```
$ ./sections
Section 3: thread 2 / 3
Section 1: thread 1 / 3
Section 0: thread 0 / 3
```

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)

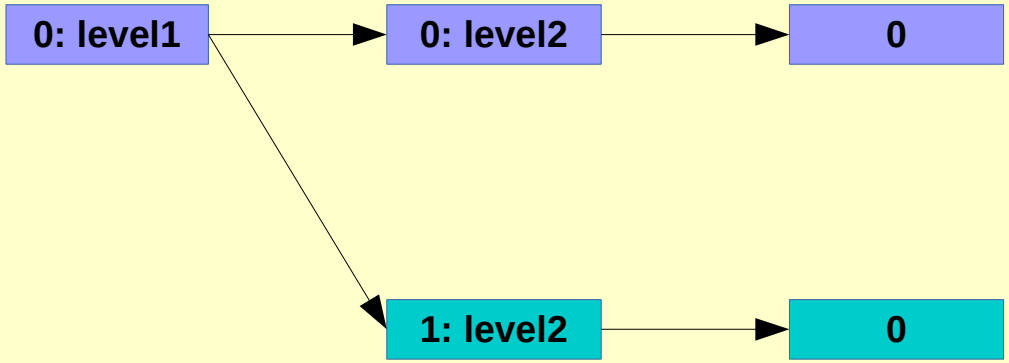
```
void level2(int parent)
{
    #pragma omp parallel num_threads(3)
    {
        #pragma omp critical
        printf("L2: parent %d, thread %d / %d, level %d (nested
               parent, omp_get_thread_num(), omp_get_num_threads
    }
}

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_threads
    }
}

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

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 1 поток — поток, который вызвал функцию level 3

Всего потоков 2 * 3 = 6

Определение числа потоков

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

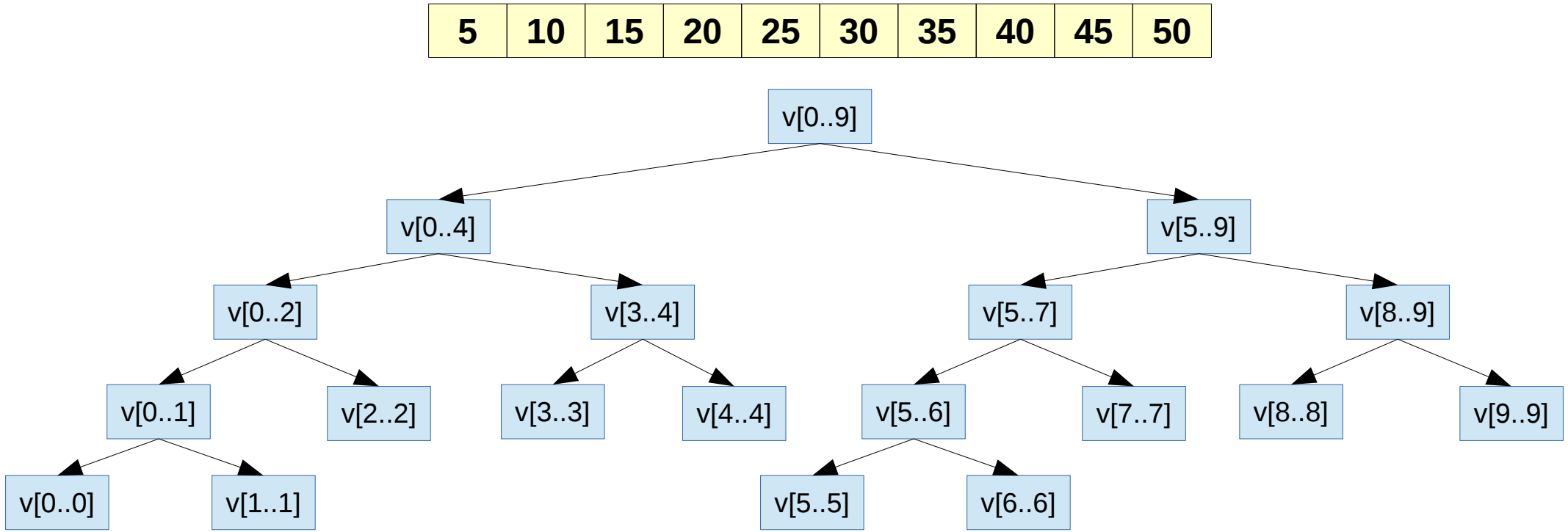
- **OMP_THREAD_LIMIT** — максимальное число потоков в программе
- **OMP_NESTED** — разрешает/запрещает вложенный параллелизм
- **OMP_DYNAMIC** — разрешает/запрещает динамическое управление числом потоков в параллельном регионе
- **ActiveParRegions** — число активных вложенных параллельных регионов
- **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);
}
```

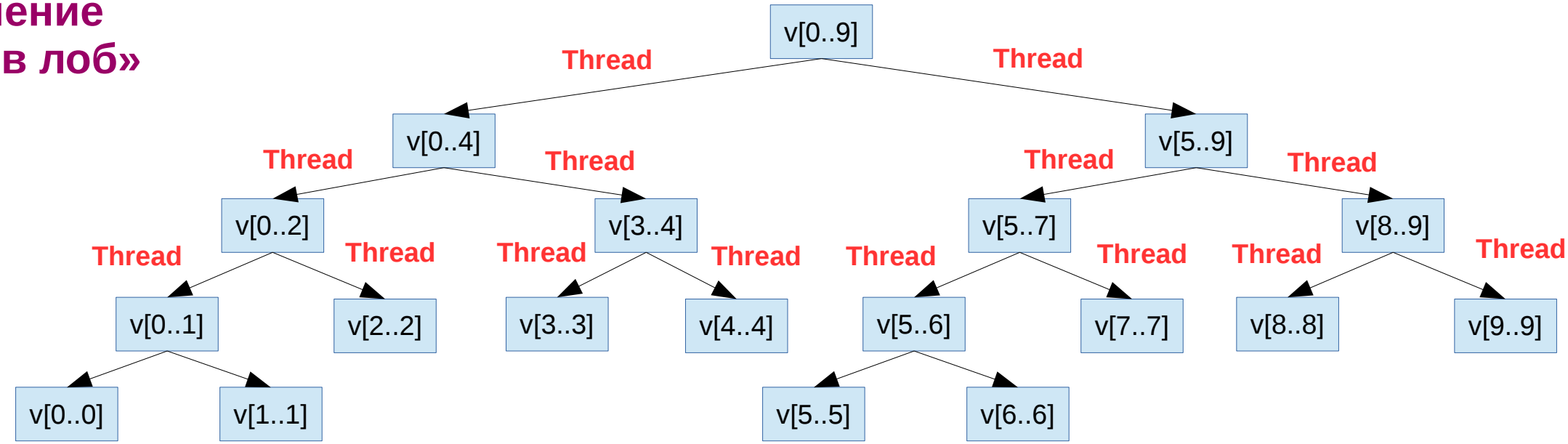


Параллельное рекурсивное суммирование

```
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);
}
```

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

На хватает ресурсов для поддержания пула потоков

```
# 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(i
    double res = sum_omp(v, 0,
}
```

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

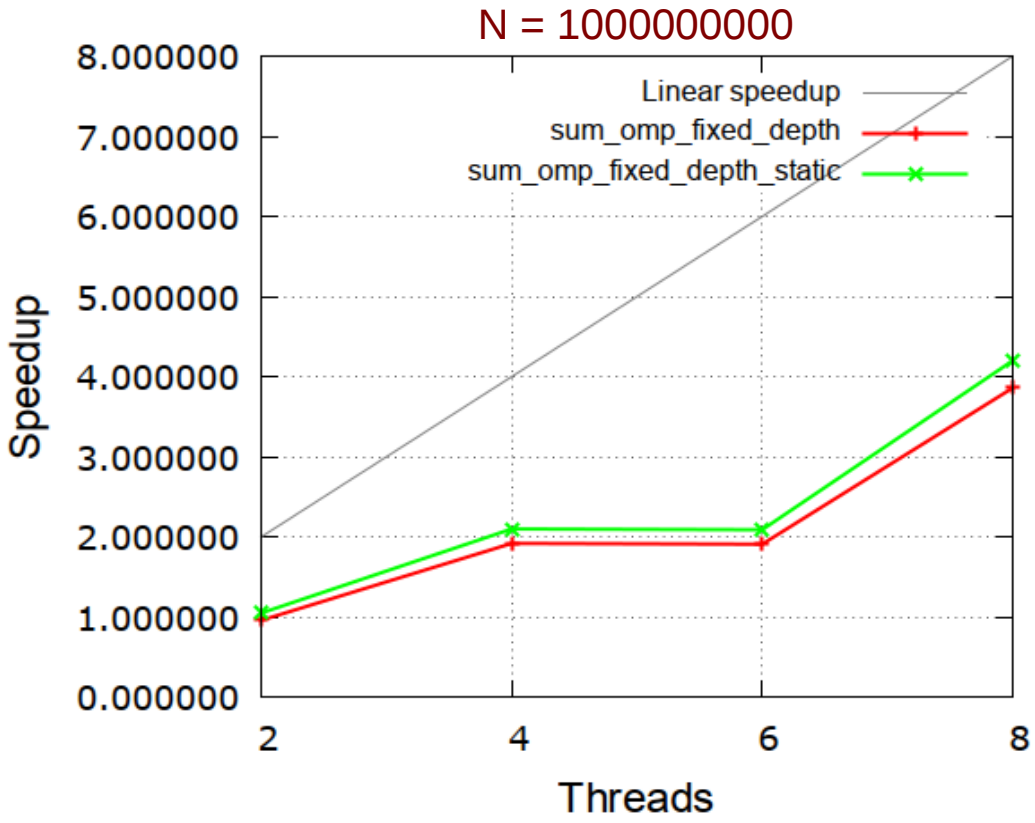
Рекурсивные вызовы в разных потоках

```
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;
}
```

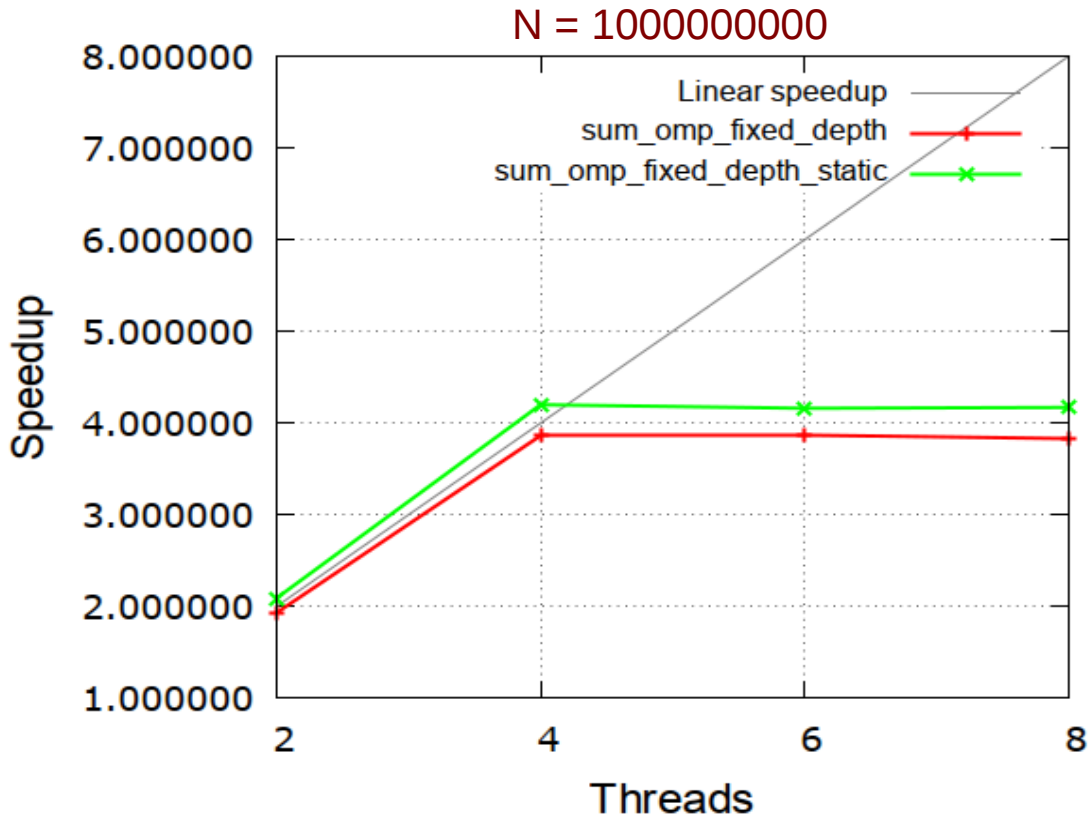
1. Ограничили глубину рекурсивных вызовов

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)

Задание

- Разработать на OpenMP параллельную версию функции sum — написать код функции sum_omp
- Провести анализ масштабируемости параллельной программы
- Шаблон программы находится в каталоге _task