

# Отчёт по лабораторной работе № 5

## OpenMP

Инструментарий:

gcc (GCC) 12.2.1

C11

<https://github.com/skkv-itmo2/itmo-comp-arch-2023-omp-Ad-mex>

Хромов Адам М3139

|                            |  |                                 |
|----------------------------|--|---------------------------------|
| Результат работы программы | output file                              | stdout                          |
|                            | 36.0001 36.0003                          | Time (12 thread(s)): 2544.61 ms |
| Лучшее время (среднее) (с) | 2.536909                                 |                                 |
| Использовано потоков       | 12                                       |                                 |
| Процессор                  | Intel Core i7-10750H, 6 ядер, 12 потоков |                                 |

## Кратко по коду

1. Определяем сторону октаэдра как минимум из длин сторон треугольника, задаваемого тремя вершинами из входного файла. Так как хотя бы две стороны из трех равны, то берем минимум из двух.

```
float get_len(coord* a, coord* b, coord* c) {
    coord ba = get_vec(b, a);
    coord bc = get_vec(b, c);
    float ba_square_len = ba.x * ba.x + ba.y * ba.y + ba.z * ba.z;
    float bc_square_len = bc.x * bc.x + bc.y * bc.y + bc.z * bc.z;
    return sqrtf(min(ba_square_len, bc_square_len));
}
```

2. Точная формула, чтобы рассчитать объем через сторону равна соответственно

$$\frac{a^3 \sqrt{3}}{2}$$

3. Для метода Монте-Карло повернем октаэдр, чтобы его диагонали были параллельны соответствующим осям координат, пересекались в точке (0; 0; 0) и уменьшим в  $\frac{a}{\sqrt{2}}$  раз. Теперь он вписан в куб (-1;-1;-1) - (1; 1;1), а его объем уменьшился соответственно в  $\frac{a^3}{2\sqrt{2}}$  раз. Объем получившегося октаэдра равен 8 объемам тетраэдров, на которые он разбивается. То есть если  $v$  - объем тетраэдра, то итоговый объем равен

$$8v \cdot \frac{a^3}{2\sqrt{2}} = v \cdot d^3$$

где  $d$  - диаметр октаэдра и равен  $\sqrt{2}a$

4. Рандом для Монте-Карло.

Сишный `rand()` не является thread safe, а `rand_r()` есть только на Linux, поэтому было принято тяжелое решение написать свой генератор псевдослучайных чисел от 0 до 1, в качестве которого был выбран генератор последовательности Холтона, так как он дает равномерно распределенный результат, что и требуется для расчета объема.

Его принцип и псевдокод были позаимствованы [отсюда](#) и генератор у меня выглядит вот так:

```
struct halton_generator {
    int base;
    int n, d;
};

float next_halton(struct halton_generator* gen) {
    int x = gen->d - gen->n;
    if(x == 1) {
        gen->n = 1;
        gen->d *= gen->base;
    } else {
        int y = gen->d / gen->base;
```

```

        while(x <= y) {
            y /= gen->base;
        }
        gen->n = (gen->base + 1) * y - x;
    }
    return gen->n / (float) gen->d;
}

```

Теперь чтобы генерировать тройки чисел (координаты) требуется генерировать одновременно 3 числа каждый раз из генераторов инициализированных тремя взаимнопростыми базами. Поэтому в коде есть замечательный массив с тройками небольших взаимнопростых, размера которого точно хватит (16) на все потоки.

Итоговый генератор следующей псевдослучайной координаты выглядит вот так:

```

struct coord_halton_generator {
    struct halton_generator g1, g2, g3;
};

coord next_halton_coord(struct coord_halton_generator* gen) {
    return (coord){next_halton(&gen->g1), next_halton(&gen->g2),
next_halton(&gen->g3)};
}

```

5. Соответственно вся программа сводится к подсчету числа успешных попаданий в тетраэдр (половинку кубика содержащую точку (0; 0; 0). Условие попадания точки - сумма координат не превышает 1.

Расчет количества попаданий (без многопоточки) производится соответственно следующим кодом:

```

struct coord_halton_generator gen;
init_coord_generator(&gen, bases[0]);
for(int i = 0; i < n; i++) {
    coord c = next_halton_coord(&gen);
    if(c.x + c.y + c.z <= 1) {
        cnt++;
    }
}

```

Ну и ответ выглядит как  $d^3 \cdot \frac{cnt}{n}$

## 6. OpenMP собственно

Отводим блок кода, который будет выполняться параллельно используя директиву `#pragma omp parallel num_threads(thread_count)`, в которую передаем максимально допустимое число потоков, отводимое программе.

Так как есть проблемы с параллельным доступом к переменным, заводим для каждого потока свой сумматор `local_cnt`, который потом будет складываться в атомарную переменную `cnt` через директиву `#pragma omp atomic` соответственно.

Ну и собственно директива, выполняющая основную работу по распараллеливанию `#pragma omp for schedule(<parameters>)`

Параметры: 1) *dynamic* - после выполненного блока запрашивается новый свободный. *static* - заранее известно какому потоку отойдет какой блок итераций. 2) *chunk\_size* - размер выделяемого блока итераций в обоих случаях, если его нет в *dynamic* по умолчанию считается равным 1, в *static* равным  $\frac{n}{\text{thread\_count}}$

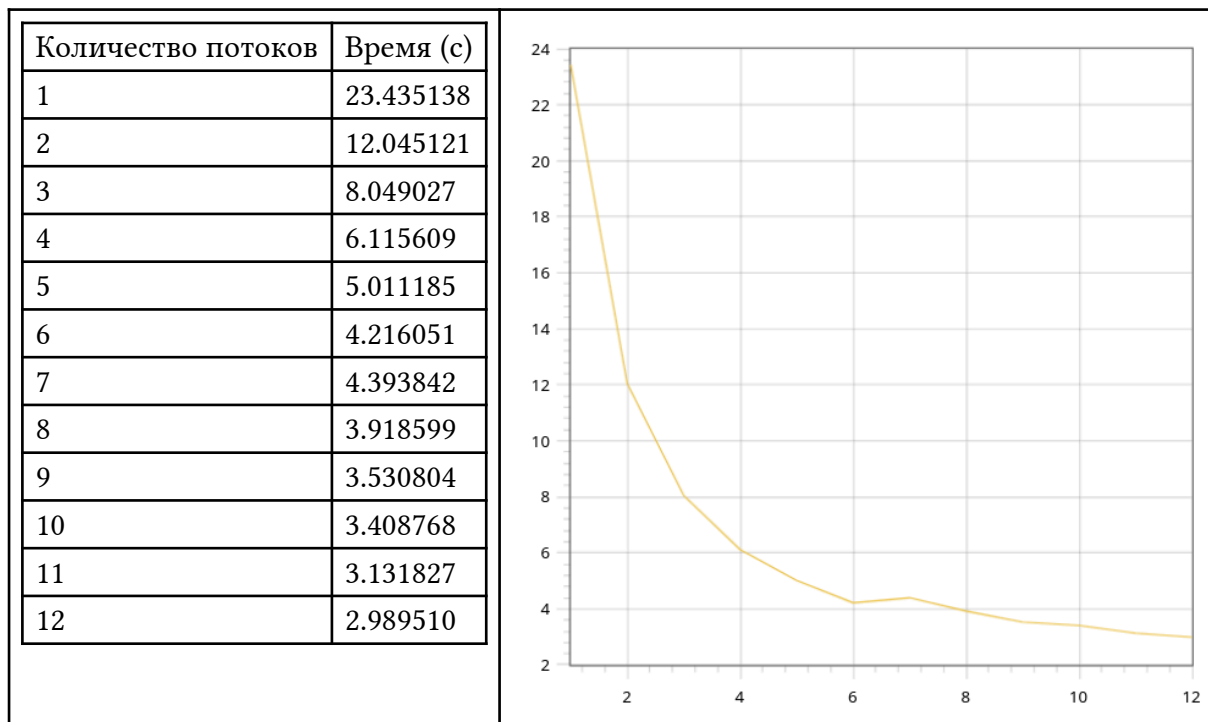
Итого код выглядит примерно следующим образом

```
#pragma omp parallel num_threads(thread_count)
{
    struct coord_halton_generator gen;
    init_coord_generator(&gen, bases[omp_get_thread_num()]);
    int local_cnt = 0;
    #pragma omp for schedule(<...parameters...>)
    for(int i = 0; i < n; i++) {
        coord c = next_halton_coord(&gen);
        if(c.x + c.y + c.z <= 1) {
            local_cnt++;
        }
    }
    #pragma omp atomic
    cnt += local_cnt;
}
```

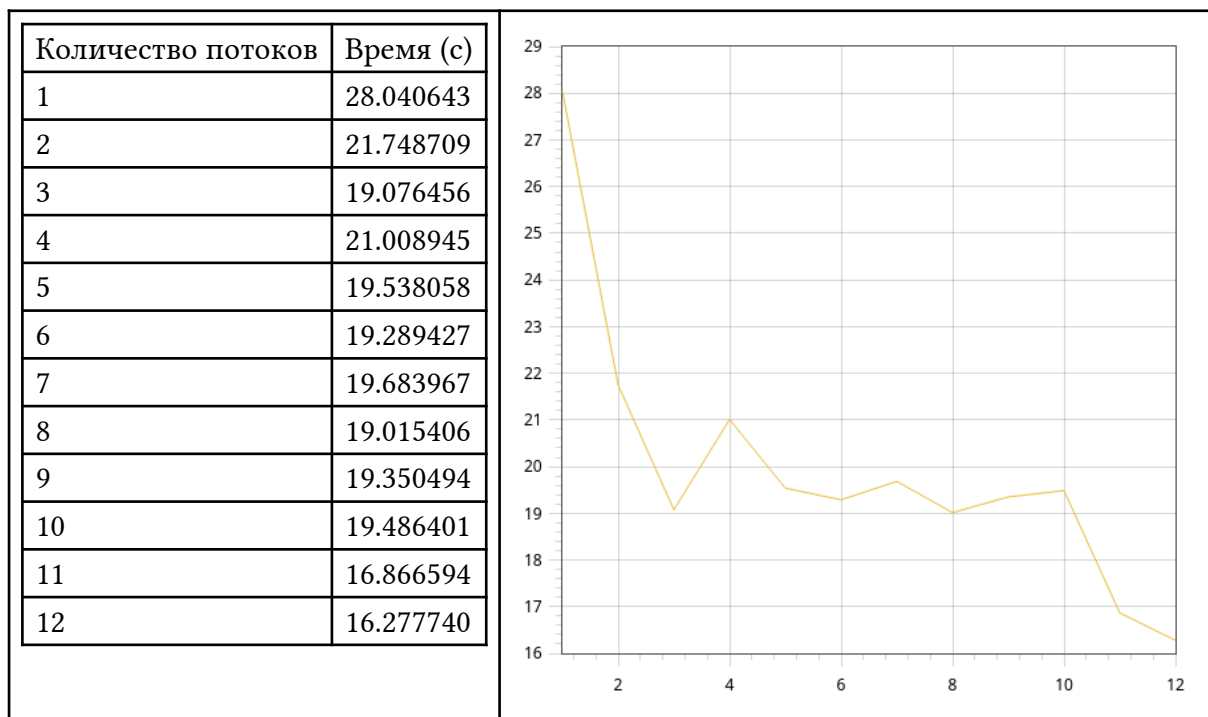
Важно: делать больше потоков для замера времени работы, чем допускает количество виртуальных потоков процессора (у меня 12) не имеет смысла, так как ядра процессора всё так же будут загружены полностью, но будет происходить постоянное переключение между потоками программы на одном виртуальном потоке проца. Поэтому количество используемых программой потоков это `omp_get_num_procs()` в том случае, когда стоит значение по умолчанию и `thread_count` иначе (ну и считаем что 0 если выполнялся запуск без `openMP`)

## Анализ и всякие графики

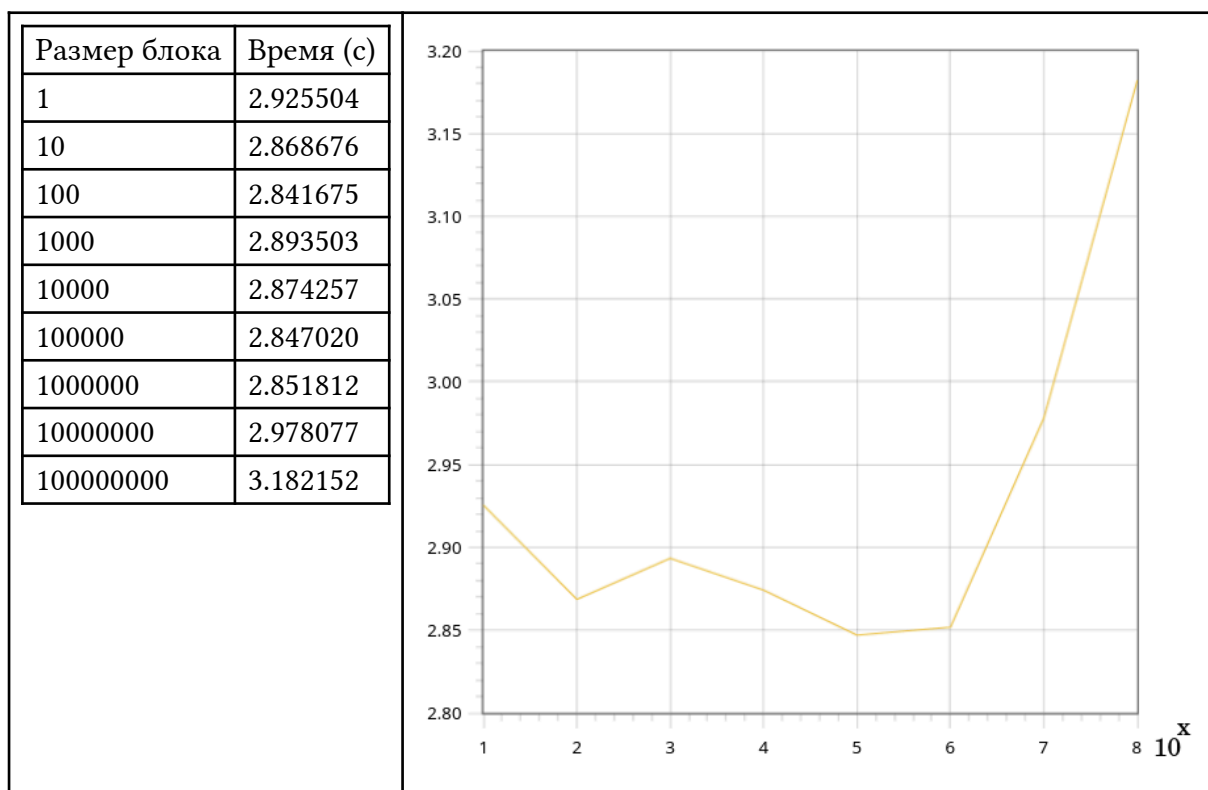
### 1.1 Shedule static, threads [1...12]



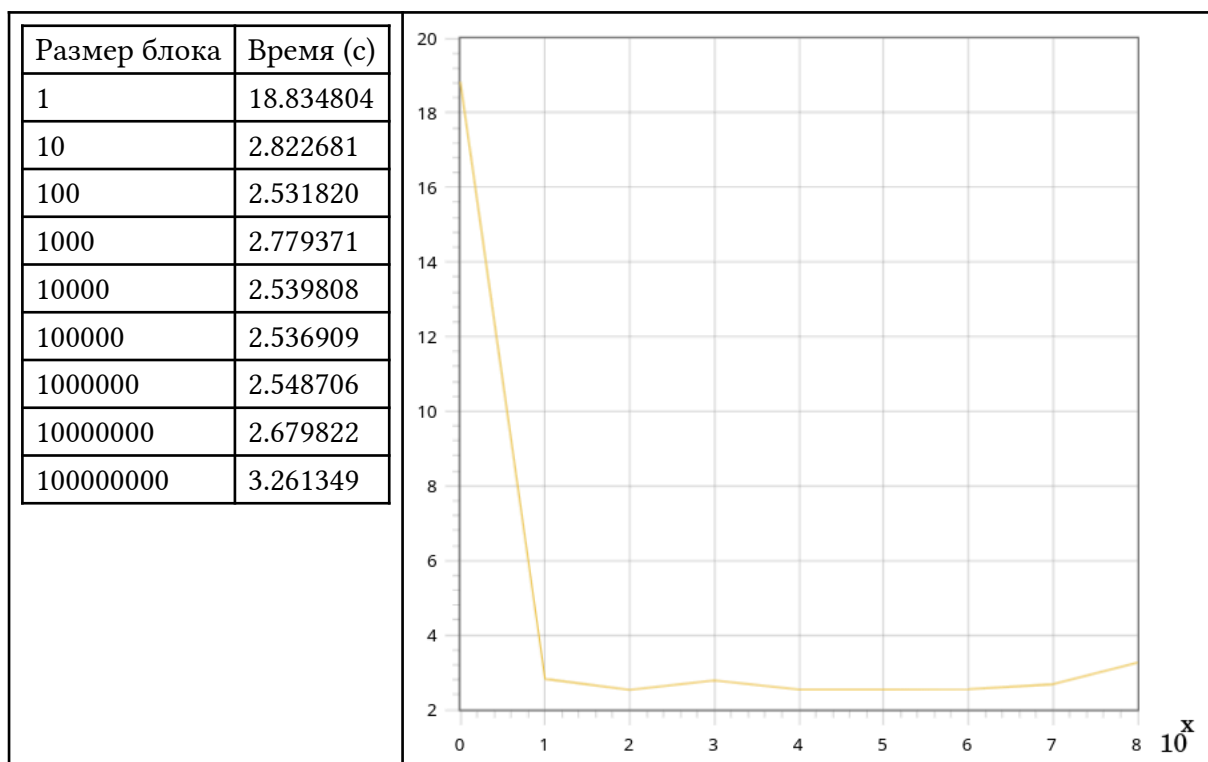
### 1.2 Shedule dynamic, threads [1...12]



## 2.1 Shedule static, threads = 12, chunk\_size ...



## 2.2 Shedule dynamic, threads = 12, chunk\_size ...



## 3. Без OpenMP

|           |           |
|-----------|-----------|
| Время (с) | 25.081323 |
|-----------|-----------|

## **Источники, которыми пользовался**

- 1) Дока по openMP - <https://www.openmp.org/wp-content/uploads/cspec20.pdf>
- 2) Рандом (Последовательность Холтона) - [https://en.wikipedia.org/wiki/Halton\\_sequence](https://en.wikipedia.org/wiki/Halton_sequence)
- 3) Пруф, почему ее можно использовать для равномерного распределения - [https://en.wikipedia.org/wiki/Low-discrepancy\\_sequence](https://en.wikipedia.org/wiki/Low-discrepancy_sequence)
- 4) То, почему `rand()` не потокобезопасен - <https://pubs.opengroup.org/onlinepubs/9699919799/functions/rand.html>