# 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 M3139**

| Результат работы программы | output file | stdout |
|---|---|---|
| | 36.0001 36.0015 | Time (12 thread(s)): 520.65 ms |
| Лучшее время (среднее) (мс) | 520.65 | |
| 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 расчитать 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 вляется 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 Xorshift64, так как он дает равномерно распределенный результат, что и требуется для расчета объема. 64-битный, так как тогда его период 2^64 - 1, что гарантированно покрывает количество итераций цикла.

Генератор соответственно у меня выглядит вот так:

```
struct xorshift64_generator {
    uint64_t a;
};

uint64_t xorshift64(struct xorshift64_generator *gen) {
  gen->a ^= gen->a << 13;
  gen->a ^= gen->a >> 7;
  gen->a ^= gen->a << 17;
  return gen->a;
}
```

```
float next_float(struct xorshift64_generator* gen) {
   return (uint32_t)xorshift64(gen) / (float)0xFFFFFFFFu;
}
```

Теперь чтобы генерировать тройки чисел (координаты) требуется генерировать одновремнно 3 числа каждый раз из генераторов инициализированных тремя своими начальными состояниями каждый. Поэтому в коде есть замечательный массив с тройками рандомных начальных состояний, размера которого точно хватит (16) на все потоки.

Итоговый генератор следующей псевдослучайной координаты выглядит вот так:

```
struct coord_generator {
    struct xorshift64_generator g1, g2, g3;
};

coord next_coord(struct coord_generator* gen) {
    return (coord){next_float(&gen->g1), next_float(&gen->g2),
next_float(&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 кооридинат не превышает 1.

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

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

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

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)

*chunck_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

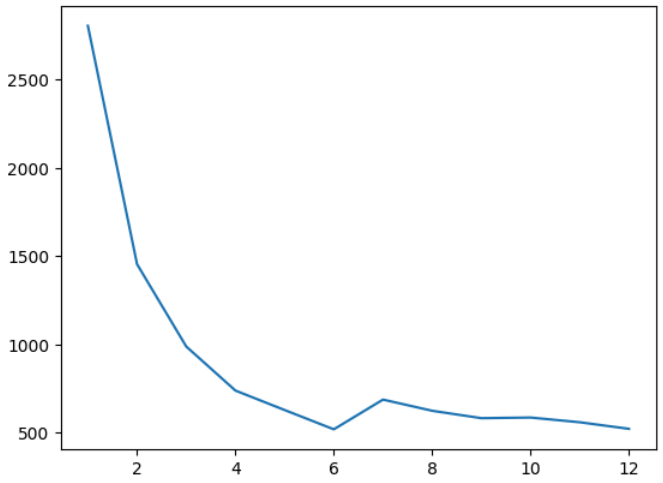
```
#pragma omp parallel num_threads(thread_count)
{
    struct coord_generator gen;
    init_coord_generator(&gen, &bases[omp_get_thread_num()]);
    int local_cnt = 0;
    #pragma omp for schedule(<...parameters...>)
    for(int i = 0; i < n; i++) {
        coord c = next_coord(&gen);
        if(c.x + 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 вирутальном 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

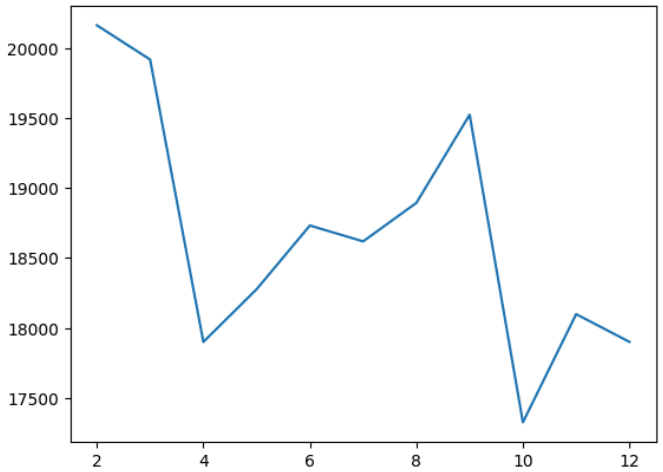
### 1.1 Shedule static, threads [1...12]

| Количество потоков | Время (мс) |
|---|---|
| 1 | 2806.68 |
| 2 | 1454.59 |
| 3 | 987.641 |
| 4 | 737.637 |
| 5 | 627.324 |
| 6 | 517.88 |
| 7 | 686.475 |
| 8 | 623.099 |
| 9 | 581.146 |
| 10 | 584.73 |
| 11 | 557.988 |
| 12 | 520.65 |



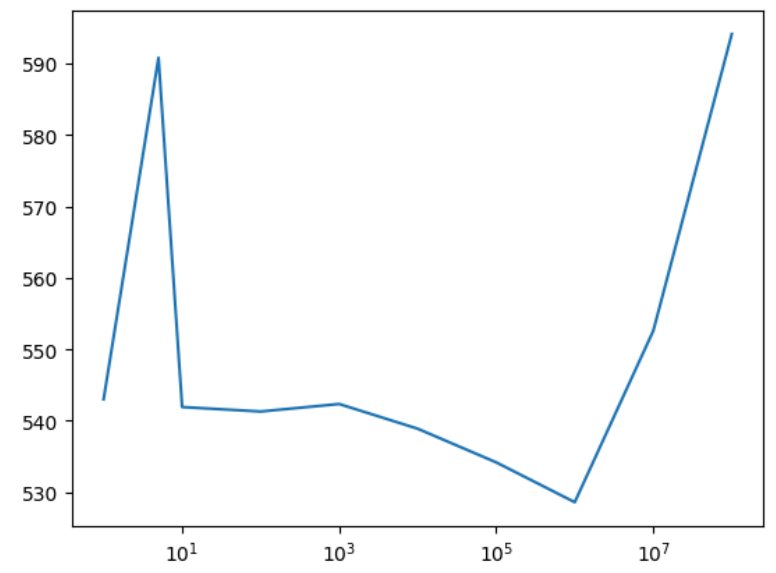

### 1.2 Shedule dynamic, threads [1...12]

| Количество потоков | Время (мс) |
|---|---|
| 1 | 10966.3 |
| 2 | 20162 |
| 3 | 19917.2 |
| 4 | 17901.2 |
| 5 | 18277.4 |
| 6 | 18732.2 |
| 7 | 18618.9 |
| 8 | 18894 |
| 9 | 19523.4 |
| 10 | 17327 |
| 11 | 18099.2 |
| 12 | 17901.4 |



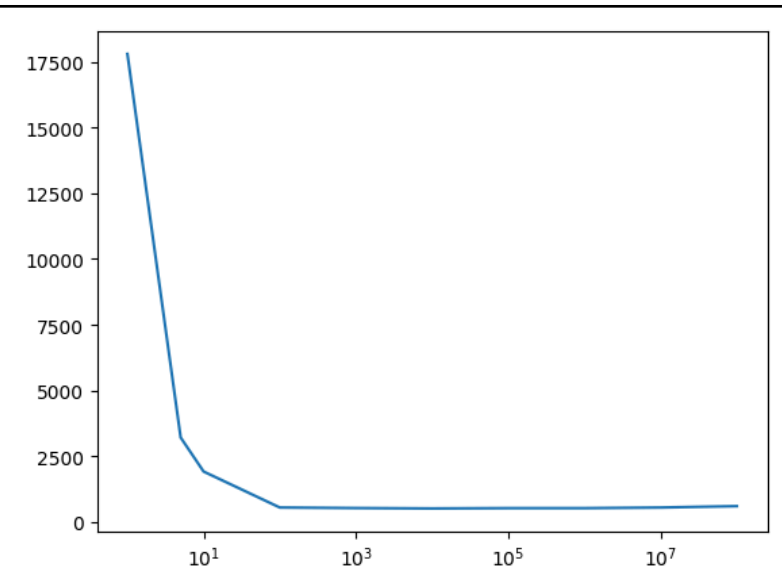
### 2.1 Shedule static, threads = 12, chunk_size ...

| Размер блока | Время (мс) |
|---|---|
| 1 | 543.015 |
| 5 | 590.786 |
| 10 | 541.93 |
| 100 | 541.307 |
| 1000 | 542.356 |
| 10000 | 538.897 |
| 100000 | 534.189 |
| 1000000 | 528.618 |
| 10000000 | 552.588 |
| 100000000 | 594.11 |




### 2.2 Shedule dynamic, threads = 12, chunk_size ...

| Размер блока | Время (мс) |
|---|---|
| 1 | 17794.4 |
| 5 | 3218.22 |
| 10 | 1924.85 |
| 100 | 557.186 |
| 1000 | 536.294 |
| 10000 | 521.566 |
| 100000 | 533.068 |
| 1000000 | 533.041 |
| 10000000 | 556.586 |
| 100000000 | 609.142 |




### 3. Без OpenMP

| Время (мс) | 6110.24 |
|---|---|

**Вывод**

Стабильно почти лучшее время показывает обычный *static* без параметров и *dynamic* с размеров блока равным 1e4 (примерно одинаковые по скорости), но так как 1e4 константа, зависящая от процессора и количества потоков очень сильно, то стабильнее себя будет вести дефолтный *static*, следовательно он и будет использован в коде.

## Источнки, которыми пользовался

1) Дока по openMP - https://www.openmp.org//wp-content/uploads/cspec20.pdf

2) Рандом (Xorshift 64) - https://en.wikipedia.org/wiki/Xorshift

3) То, почему `rand()` не потокобезопасен - https://pubs.opengroup.org/onlinepubs/9699919799/functions/rand.html