Лабораторная работа №5 - OpenMP

Зайцев Захар Олегович. М3136

Минититульник

1. ФИО: Зайцев Захар Олегович. М3136
2. Ссылка на git
3. C++14 . Visual Studio 2022.

Содержание

1 Учим C++ за 24 часа

1.1 Настройка среды

Источники - Сайт для установки Visual Studio
Версия - C++14 Узнаем версию
Как сказано в ТЗ -
1) Свойства проекта (ПКМ по проекту в обозревателе проектов) - C/C++ - Language - OpenMP support - Yes.
2) Дополнительно стоит отключить SDL check, чтобы "немодные" с точки зрения msvc функции по типу fopen можно было использовать.

1.2 Базовый код

Источники - Видео на Ютуб про основы C++
Пример базовой программы -

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
        setlocale(LC_ALL, "rus");
        cout << "Привет World" << endl;
        system("pause");
        return 0;
}
```

Описание кода:

1. #include <iostream> - директив (библиотека в питоне) для включения ввода и ввывода
2. #include <cstdlib> - директив для system("pause");
3. using namespace std - чтобы не писать каждый раз std::
4. setlocale(LC_ALL, "rus") - чтобы можно было выводить фразы на русском
5. cout « "Привет World«< endl - Вывод на консоль
6. system("pause") - чтобы консоль не закрывалась сразу
7. return 0 - возвращаем результат программы.

Директивы в программировании - это специальные инструкции, которые сообщают компилятору или интерпретатору о том, как обрабатывать определенную часть кода.
Примеры дериктивов:

1. #include: Используется для включения содержимого других файлов в исходный код программы.
2. #define: Используется для определения макросов препроцессора.
   #define SQUARE(x) ((x) * (x))
   int result = SQUARE(5); // result будет равен 25
3. #error: Используется для генерации ошибки в процессе компиляции.

1.3 Аргументы командной строки

Источники - Добавление параметров - argc argv c++ что это. Параметры функции main argc argv

```
int main(int argc, char* argv[]) {
    setlocale(LC_ALL, "rus");
    for (int i = 0; i < argc; i++)
    {
        cout << argv[i] << endl;
    }
    }
```

1. argc - количество аргументов
2. argv - массив аргументов

```
int numThreads = atoi(argv[1]);
string inputFileName = argv[2];
string outputFileName = argv[3];
```

atoi - преобразуем строчку в число

1.4 Работа с памятью и файлами

Источник - Работа с файлами c++. Чтение из файла c++ ifstream.
Доп дериктив - #include <fstream>
1) Закрываем файл после прочтения -

```
ifstream inputFile(inputFileName);
inputFile >> radius >> totalPoints;
inputFile.close();
```

2) Запись в файл (Используем fprintf)

```
FILE* file = fopen(outputFileName.c_str(), "w");
fprintf(file, "Time (%i thread(s)): %g ms\n", numThreads, (end_time - start_time) * 1000);
fclose(file);
```

Работа с памятью - Утечка памяти и другие проблемы.
А так надеюсь просто с этим не сталкиваться.

1.5 Обработка ошибок

1) Недостаточное количество аргументов

```
if (argc < 4) {
    cerr << "There are not enough arguments. Usage: " << argv[0] << " <number of streams>
    <input filename> <output filename>" << endl;
    return 1;
}
```

2) Не удалось открыть файл
2.1) Чтение

```
ifstream inputFile(inputFileName);
if (!inputFile.is_open()) {
    cerr << "The file could not be opened " << inputFileName << endl;
    return 1;
}
```

2.2) Запись

```
if (file == NULL) {
    cerr << "File opening error!" << endl;
    return 1;
}
```

3) Формат файла не поддерживается
3.1) Чтение

```
try {
    inputFile >> radius >> totalPoints;
}
catch (const exception& e) {
    cerr << "Error reading the file: " << e.what() << endl;
    return 1;
}
```

3.2) Запись

```
try {
    fprintf(file, "Time (%i thread(s)): %g ms\n", numThreads, (end_time - start_time) * 1000);
}
catch (const exception& e) {
    cerr << "Error writing the file: " << e.what() << endl;
    return 1;
}
```

4) Неподдерживаемое количество потоков (меньше -1)

```
else {
    cerr << "Unsupported number of threads" << endl;
    return 1;
}
```

5) Отрицательный радиус или не число. Стоит отметить, что если число точек не целое, то оно округляется до целого числа.

```
if (inputFile >> radius >> totalPoints) {
    if (radius > 0 && totalPoints > 0) {
    }
    else {
        std::cerr << "Numbers are not positive" << std::endl;
        return 1;
    }
}
else {
    std::cerr << "Error reading numbers from file" << std::endl;
    return 1;
}
```

2 Знакомство с методом Монте-Карло

Источник - КОД журнал Яндекс Практикума

Под метдом Монте-Карло понимается численный метод решения математических задач при помощи моделирования случайных величин.

2.1 Easy

Кидать песчинки будем так: в качестве координат попадания X и Y будем брать случайные числа от 0 до 1. Это значит, что все числа попадут только в один квадрант — правый верхний:

Но так как в этом квадранте ровно четверть круга и ровно четверть квадрата, то соотношение промахов и попаданий будет таким же, как если бы мы бросали песчинки в целый круг и целый квадрат.

Чтобы проверить, попадает ли песчинка в круг, используем формулу длины гипотенузы: $X^2 + Y^2 = R^2$ (так как гипотенуза — это радиус окружности):

Если длина гипотенузы меньше единицы — точка попадает в круг. В итоге мы посчитаем и общее количество точек, и точек, которые попали в круг. Потом мы разделим одно на другое, умножим результат на 4 и получим приближённое значение площади круга.

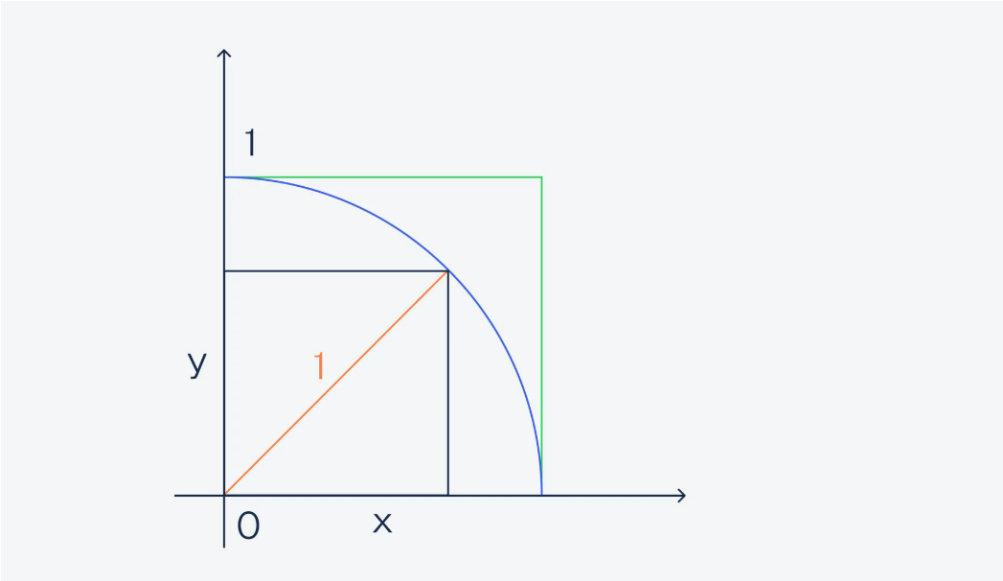


Рис. 1: Четверть круга на основе которой считаем площадь круга (Главное в конце умножить ответ на 4).

```
double calculateSequentially(double radius, int totalPoints) {
    int pointsInsideCircle = 0;

    for (int i = 0; i < totalPoints; i++) {
        double x = (double)rand() / RAND_MAX * radius;
        double y = (double)rand() / RAND_MAX * radius;

        if ((x * x + y * y) <= (radius * radius)) {
            pointsInsideCircle++;
        }
    }

    return (static_cast<double>(pointsInsideCircle) / totalPoints) * (4 * radius * radius);
}
```

2.2 Normal

Несколько изменений по сравнению с версией easy

1) Вместо радиуса - полудиаметр октаэдра. Нам даны три точки, которые однозначно задают октаэдр - это значит, что 2 точки лежат среди 4 "в квадрате а оставшаяся точка не "в квадрате"или же 2 точки лежат не "в квадрате"и 1 в нем. Нам без разницы какой случай - так как вертикальные и горизонтальные полудиагонали равны в октаэдре. Теперь мы строим 1/8 октаэдра - как прямоугольный тетраэдр - где все стороны от центральной точки до вершин равны длине полудиагонали.

2) На основе построения меняется и условие выбора лежит ли точки внутри или нет. Для этого достаточно просуммировать все координаты и если они меньше полудиагонали, то точка лежит в 1/8 октаэдра.

3) Меняется итоговая формула - (static_cast<double>(pointsInsideCircle) / totalPoints) * (diag * diag * diag) * 8

4) Координаты вычисляются по формулам.

```
double x = dis(gen);
double y = dis(gen);
double z = dis(gen);
```

5) Критерий по которому считаем, что точка лежит внутри октаэдра

```
if (abs(x) + abs(y) + abs(z) <= diag) {
    localCount++;
}
```

6) Можем генерировать числа в промежутке от 0 до 1 при этом сравнение будет выглядеть так (x + y + z <= 1) - мы условно разделили данное выражение заранее на diag.

3 Учим OpenMP за 24 часа

Источники - Документация OpenMP.

OpenMP (Open Multi-Processing) - это набор директив компилятора, функций и переменных среды выполнения для языков программирования C, C++ и Fortran, который позволяет разработчикам создавать параллельные программы, которые могут выполняться на многопроцессорных системах.

Чтобы ввести количество используемых потоков используем - **omp_set_num_threads**(numThreads); В документации есть целый порядок условий, которые могут определить количество потоков. И если ни одно из тех условий не выполняется, то будет подставлено значение "по умолчанию-определенное реализацией (Нам это нужно если numThreads = 0) "У меня по умолчанию стоит 8 потоков - проверить это можно с помощью omp_get_num_threads". Чтобы понять какой поток сейчас работает используем omp_get_thread_num().

3.1 #pragma omp parallel

#**pragma omp parallel** (Раздел 2.3) - Нужна, чтобы показать компилятору, что следующий блок будет выполняться параллельно.

Если динамическая настройка отключена, то количество потоков столько и будет.Если же динамическая настройка включена, то это будет максимальное количество потоков, которое может одновременно работать.

Если же динамическая настройка отключена а мы запросили слишком много потоков - то дальнейшее поведение зависит от реализации.

3.2 #pragma omp for

#**pragma omp for** (Раздел 2.4.1) - Нужен, чтобы цикл выполнялся параллельно.

3.2.1 Ограничения

1)Существуют некоторые ограничения - То есть цикл for() должен содержать только определенные комбинации ("ну там в документации есть").Ну то есть можно как обычно пишем - Пример (i = 1; i <= 10; i++). Что-то странное писать нельзя - Пример (i=1; i>-1; i++).

2) Цикл for должен быть структурированным блоком, и, кроме того, его выполнение не должно завершаться оператором break.

3) Значения управляющих выражений цикла for, связанного с директивой for должны быть одинаковыми для всех потоков в команде.

3.2.2 Чанки

Источник - OpenMP

1) Ограничения

Значение chunk_size, если оно указано, должно быть циклическим инвариантным целочисленным выражением с положительным значением.

2) Опеределение

OpenMP делит итерации на куски размером chunk-size и распределяет их между потоками в круговом порядке (При static scheduling).

Каждый поток выполняет свой кусок итераций, а затем запрашивает другой кусок, пока не останется свободных кусков (При dynamic scheduling). Динамический тип планирования подходит, когда итерации требуют разных вычислительных затрат. Это означает, что итерации плохо сбалансированы между собой.

Если размер chunk-size не указан, OpenMP делит итерации на примерно равные по размеру куски и распределяет каждому потоку не более одного куска.

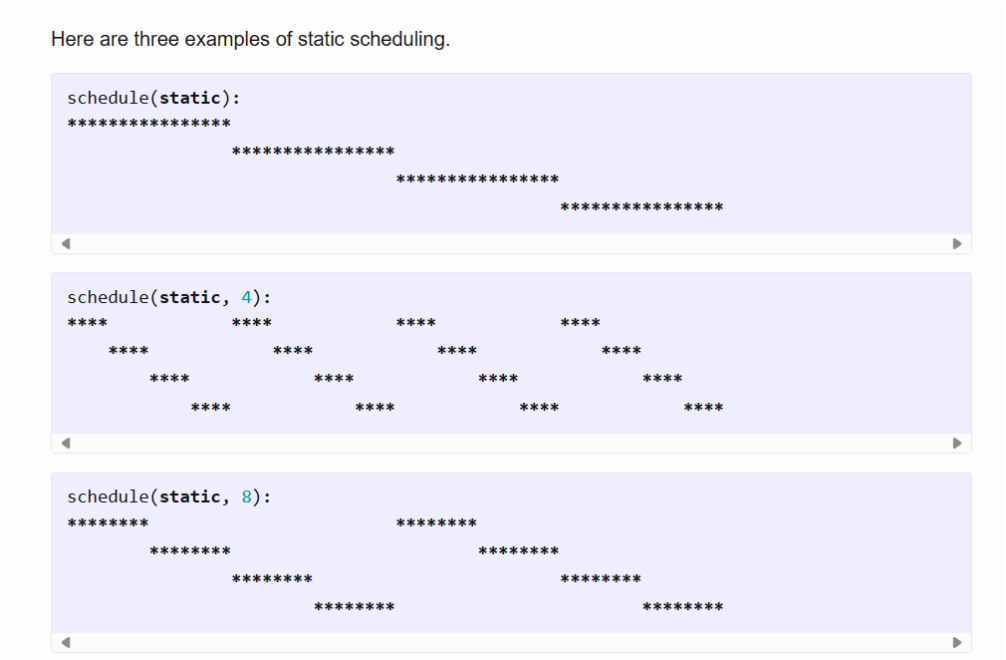


Рис. 2: Визуализация разделения на chunk_size для static scheduling

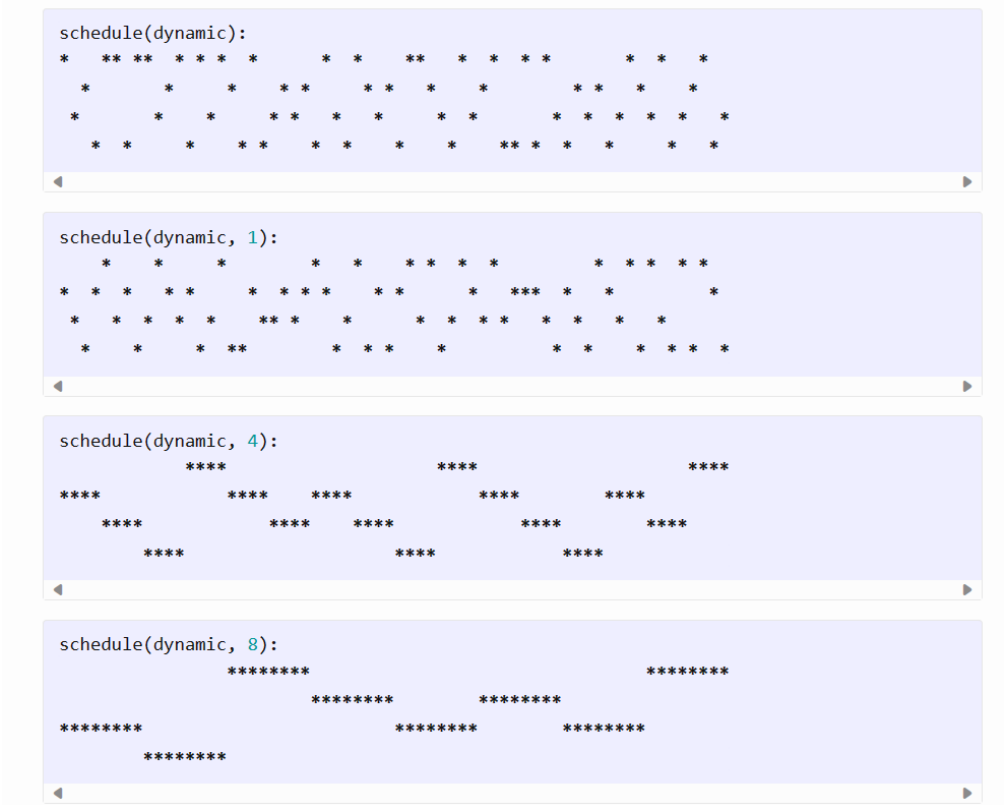


Рис. 3: Визуализация разделения на chunk_size для dynamic scheduling

3.2.3 Schedule

Сущесвуют некоторые надстройки - **schedule** - объясняет как итерации цикла будут распределены по потокам. - Есть 4 вариации (static, dynamic, guided, runtime). У первых трех присутсвует дополнительный параметр - chunk_size.

1) **static** - итерации делятся на чанки размера - chunk_size. Каждый чанк назначается потоком последовательно по кругу. (Например 1 потоку достанется 1,9,17... чанк - если было всего 8 потоков). Если данный параметр не указан, то максимально равномерно между всеми потоками и у каждого потока по 1 чанку.

2) **dynamic** - итерации распределяются среди потоков. Если поток отработал свой чанк, то он ждет пока ему назначят новый чанк - то есть ему достаются рандомные чанки. Итерации делятся на чанки размером - chunk_size. По умолчанию значение chunk_size = 1.

3) **guided** - Понял - Прикольно - Но использовать мы не будем. По мере работы программы остается все меньше и меньше непройденных итераций - поэтому чтобы компенсировать этот эффект придумали данный метод. Количество чанков, которые мы выдаем потоку зависит от количества оставшихся непройденных итераций. По умолчанию значение chunk_size = 1 - а тогда 1. Если значение chunk_size больше, то там распределяются итерации по экспоненциальному уменьшению.

$$\text{Размер чанка} = \left(\frac{\text{Количество оставшихся итераций}}{\text{Количество потоков}}\right) \tag{1}$$

4) **runtime** - решение относительно типа schedule относится на более поздний срок - до выполнения программы. schedule - будет выбран во время реализации программы - иначе по умолчанию - в зависимости от реализации.

schedule - Если не указан - то будет по умолчанию в зависимости от реализации.

Но не стоит полагаться на точно такое исполнение от компилятора - ведь каждый компилятор работает по-разному. Описания помогают с выбором нужного schedule.

В конце цикла - неявный барьер если нет nowait.

3.3 Other directives (Critical, Atomic)

Other directives (Раздел 2.6)

- **critical** - Выполнение блока одновременно только одним потоком. Пока все потоки не дойдут до блока - то выполнение critical не будет происходить. Также есть параметр name - по которому объединяются потоки.

- **atomic** - Обновление участка кода одним потоком.Atomic лучше оптимизирован чем critical - но мы при этом ограничены в функционале - но для нашей задачи подходит.

- atomic нужен чтобы не было гонки записи (race condition)

- **threadprivate** - создает для каждого потока приватную переменную.

3.4 Вычисление времени с помощью OpenMP

Вычисление времени происходит с помощью **omp_get_wtime()** (Раздел 3.3)

```
start_time = omp_get_wtime();
circleArea = calculateSequentially(radius, totalPoints);
end_time = omp_get_wtime();
fprintf(file, "Time (%i thread(s)): %g ms\n", numThreads, (end_time - start_time) * 1000);
```

3.5 Вычисление площади с помощью OpenMP

- Как распараллелить?

Мы можем каждому потоку дать свой счетчик попаданий, а в конце сложить все полученные значения в одну переменную. Ведь нам абсолютно не важно в каком порядке мы попадали в заданную область. Чтобы сложения проходили верно - добавляем дериктив atomic - который не дает сразу нескольким потокам записывать результат.

Также мы с помощью #for распределяем итерации среди всех потоков. Значения координат независят от других потоков, так как у каждого потока уникальные номера итераций.

3.5.1 Сравниваем способы генерации точек на примере easy

Будем тестировать при радиусе = 1 и количестве точек = 100_000. Статистика на основе 100 запусков.

Сущесвует несколько вариантов генерации точек, которые мы здесь и рассмотрим

1) Использование rand() и srand() из библиотеки <cstdlib>:

Код с использованием cstdlib (easy)

Rand() - и у каждого потока свой seed - srand(100 + thread_id) - чтобы значения у каждого потока не повторялись.

1) Мы сделали 100 000 точек на 8 потоках с политикой dynamic и чанк сайзом

Таблица 1: Результаты с помощью директивы `cstdlib`

Statistic	Time	Value
Min	1.0839	3.13308
Max	4.1153	3.14808
Mean	1.4924	3.13788
Median	1.44935	3.13776

2)Использование <random> из стандарта C++11:

Код с использованием random

1) Мы сделали 100 000 точек на 8 потоках с политикой dynamic и чанк сайзом

Таблица 2: Результаты с помощью директивы `random`

Statistic	**Time**	**Value**
Min	2.453	3.13224
Max	44.2733	3.16036
Mean	3.312531	3.143149
Median	2.8652	3.14274

3.5.2 Сравниваем способы генерации точек на примере normal

Результаты на одном запуске теста из ТЗ normal. Используем 8 потоков.
Какие у нас есть варианты?

1. Генерировать числа типа float.
2. Генерировать числа типа double.
3. Генерировать 2 числа тип float и объединить в тип double (не будет использован)

Также надо решить какие встроенные функции использовать.

1. rand()
2. mt19937

3. mt19937_64 (не будет использован)

4. Использование других генераторов

Источники информации -
1)Генерация случайных чисел
2)StackOverFlow
3)Википедия - генераторы псевдослучайных чисел
4)Очередная статья на habr .Обзор генераторов псевдослучайных чисел
5)Википедия - Линейный конгруэнтный метод
Кратко

Таблица 3: Результаты при 8 потоках

Метод	**Время на ПК**	**Время на GIT**	**Значения**	**Код**
rand()	20573.7 ms	больше 5 минут на Linux	36.0022	2
random - double	38788.1 ms	156 sec на Linux и 16 sec на Windows	36.0003	But double in 4
random - float	28258.4 ms	137 sec на Linux и 9 sec на Windows	35.9971	4
lcg-gen	4440.6 ms	Около 20 секунд на Linux и Windows	36.0025	6

Подробнее

1) Использование rand() . Если тестить на гитхабе - **то оно работает как минимум 5 минут на Linux, что говорит нам о том, что этот метод очень медленно работает** (или код написан не оптимально). Важно что если мы используем rand() ,то он зависит от ОС - так на Windows RAND_MAX = 32767 и поэтому число будут часто повторятся (хотя можно сделать и генерации большого диапозона чисел как-то вот так(RAND_MAX * rand() + rand()). (Источник - 1)

2) Используем mt19937

mt19937 - подходит так как он платформено независимый и с большей точностью генерирует числа.Данный метод работает на основе вихря Мерсенна, но он относительно других методов работает дольше всех.

2.1) Разница между dynamic и static - практически незаметна. Но мне кажется, что легче и как бы более предсказуемее (если тестировать и на других данных где заранее не знаешь количество точек) вариант со static - при том что у нас все операции выполняются примерно за одно и тоже время.

Таблица 4: Результаты при 8 потоках

Configuration	**Time (8 thread(s))**	**Values**
dynamic + sizeChunk(2)	36578.3 ms	36 36.0001
static	36494.5 ms	36 36.0025

Доп. советы -

*плохо использовать еще random_device - так как он медленный и он платформозависим.

*для генерации времени можно использовать time(0), а можно использовать время в наносекундах - но в нашей работе это не имеет смысла.

2.2) Решаем проблему с временем на Linux - Попробуем использовать вместо uniform_real_distribution использовать другое.

1) generate_canonical<double, 32>(gen); Работает за 18 секунд, но точность плохая (35.9904) .

2) Пробуем увеличить количество знаков. Работает за 27 секунд, но точность хорошо увеличилась (36 35.9998).

– Но все равно время на Linux больше 100 секунд. –

2.3) Решение проблемы - Linear Congruential Generator Линейный конгруэнтный метод (LCG) - это метод генерации псевдослучайных чисел, который использует линейное рекуррентное соотношение. Он определяется следующим образом:

```
    unsigned long long seed;
    unsigned long long a = 1664525;
    unsigned long long c = 1013904223;
    unsigned long long m = 4294967296;

    unsigned long long lcg_rand(unsigned long long seed, unsigned long long a,
unsigned long long c, unsigned long long m) {
    return (a * seed + c) % m;
}
```

Выводы. В итоге выбираем метод LCG - так как он показал лучшие результаты по скорости в несколько раз лучше других методов (а мы смотрим на это в первую очередь) и не ужасные показатели по точности.

4 Результаты при использовании rand()

Результаты:
Процессор - 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
Наилучшее распределение - 8 потоков, schedule(dynamic, 2)
Дано - 15 запусков для каждого потока. Радиус = 1, количество точек = 100_000_000.

Таблица 5: Измерения результатов

Время (мс)	Результат
1073.75	3.14127
1077.8	3.14124
1079.74	3.14134
1081.96	3.14131
1084.53	3.14127
1083.92	3.14127
1082.89	3.14147
1082.72	3.1416
1446.72	3.14158
1602.37	3.14163
1596.83	3.14164
1579.34	3.14157
1596.52	3.14147
1562.64	3.1414
1571.03	3.14141
Среднее время	1381.27
Средний результат	3.14144

* Стоит отметить, что надо больше обращать внимание в сторону минимального значения, так как после 15 последовательных вычислений по 100 миллионов итераций - скорость программы уменьшается из-за нагрузки на процессор. При меньших вычислениях логично смотреть на средний результат (ну или на медианный в идеале).

4.1 Разное число потоков

Основные выводы :

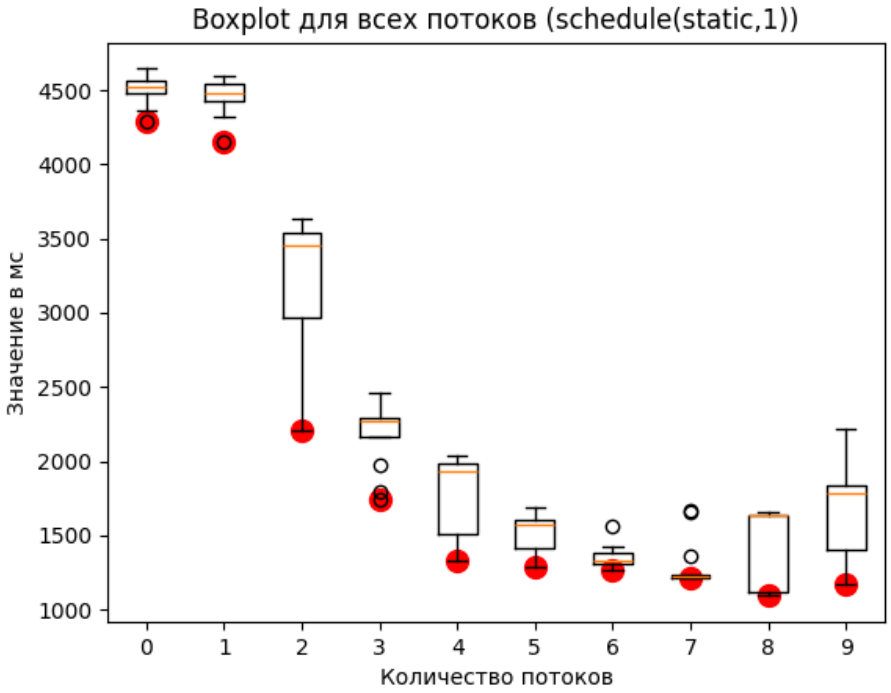
1) При schedule(dynamic, 1) время работы увеличивается с увеличением числа потоков. Скорее всего это происходит из-за того, что программе надо слишком часто назначать потокам различные итерации (так как чанки маленькие и потоки быстро с ними справлются) и из-за этого программа замедляется.

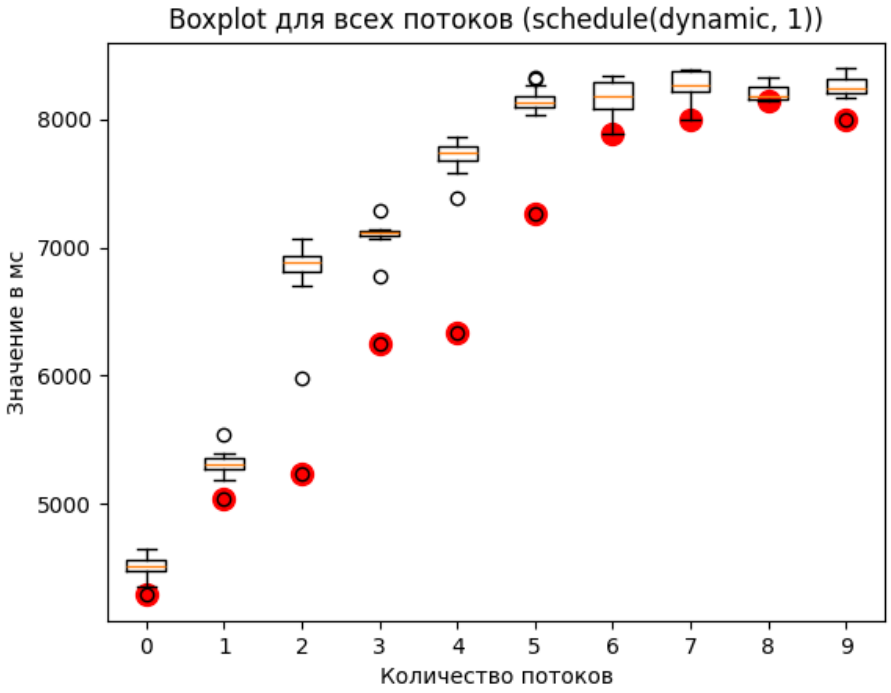
2) При schedule(static, 1) время работы уменьшается до определенной степени с увеличением числа потоков. На 8 потоках наблюдается минимальное время после же наблюдается рост времени - это объясняется тем ,что у моего процессора максимум 8 потоков.

3) Если сравнивать значения между включенным 1 потоком и без openMP то особой разницы нет, но быстрее работает программа без openMP.

*(Пояснение к графикам) Красным отмечено минимальное время. А так на графиках нарисованы ящики с усами для каждого потока.

* На позиции 0 - находится программа без OpenMP


Boxplot для всех потоков (schedule(static,1))
Значение в мс
4500
4000
3500
3000
2500
2000
1500
1000
0
1
2
3
4
5
6
7
8
9
Количество потоков


Boxplot для всех потоков (schedule(dynamic, 1))
Значение в мс
8000
7000
6000
5000
0
1
2
3
4
5
6
7
8
9
Количество потоков

4.2 Разный тип и размер чанков у schedule

Будем работать на 8 потоках с различными chunk_size и двумя типами schedule.

Основные выводы :

1) Для static относительно не важно какого размера будет chunk_size.

2) Для dynamic с увеличением chunk_size до определенного момента (2) уменьшается время работы программы.

3) С некоторого момента с увеличением chunk_size время работы с dynamic меньше чем со static. Это объясняется тем, что при static у каждого потока определено заранее количество итераций ,а при dynamic подходе каждый поток берет небольшой кусок итераций и если он не наелся, то берет новый. И из-за того что потоки могут не совсем одинаково работать и возникает выигрыш в -20 мс - даже относительно это всего лишь 2 процента.

Выводим оптимальную формулу для chunk_size в зависимости от числа точек и при условии, что потоков 8 а тип schedule == dynamic. На основе графиков для различных количеств точек (1000, миллион, 100 миллионов).

$$\text{sizeChunk} = \max\left(1, \left\lfloor \frac{2^{\lceil 1.6 \times \log_{10}(\text{totalPoints}) \rceil}}{8} \times \max\left(1, \text{numThreads}\right) \right\rfloor\right) \tag{2}$$

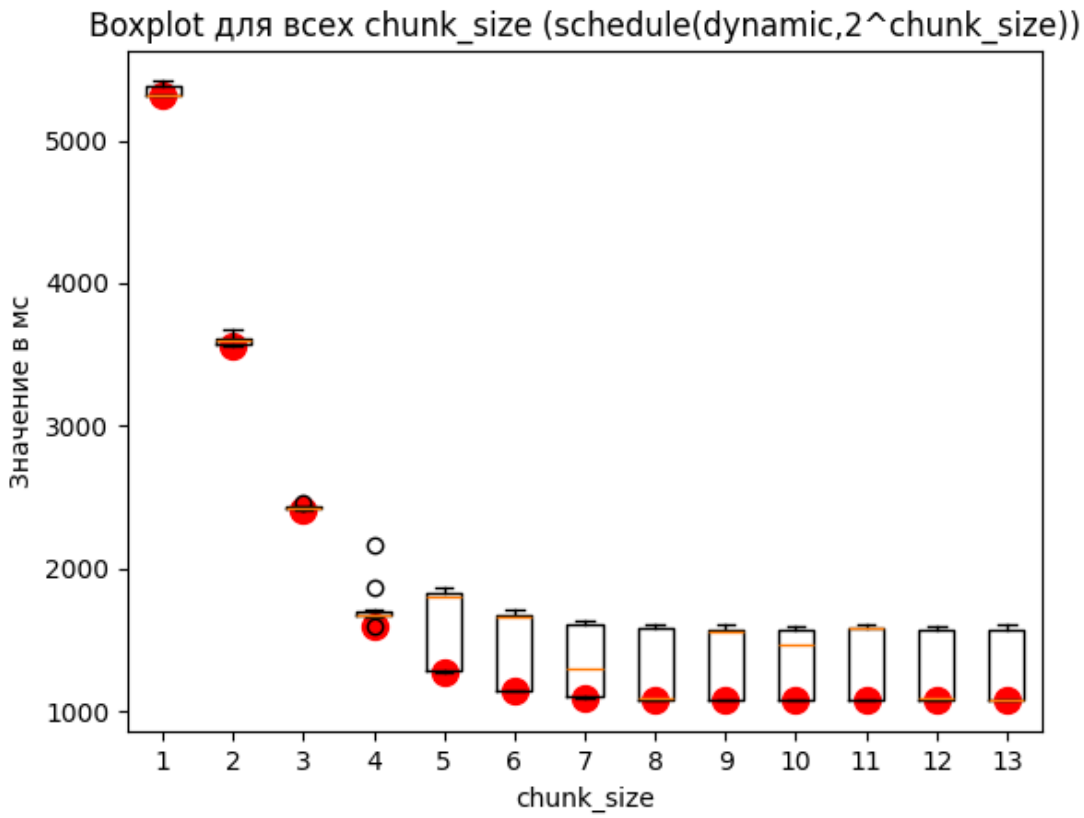


Рис. 4: Потоков - 8. Точек - 100_000_000

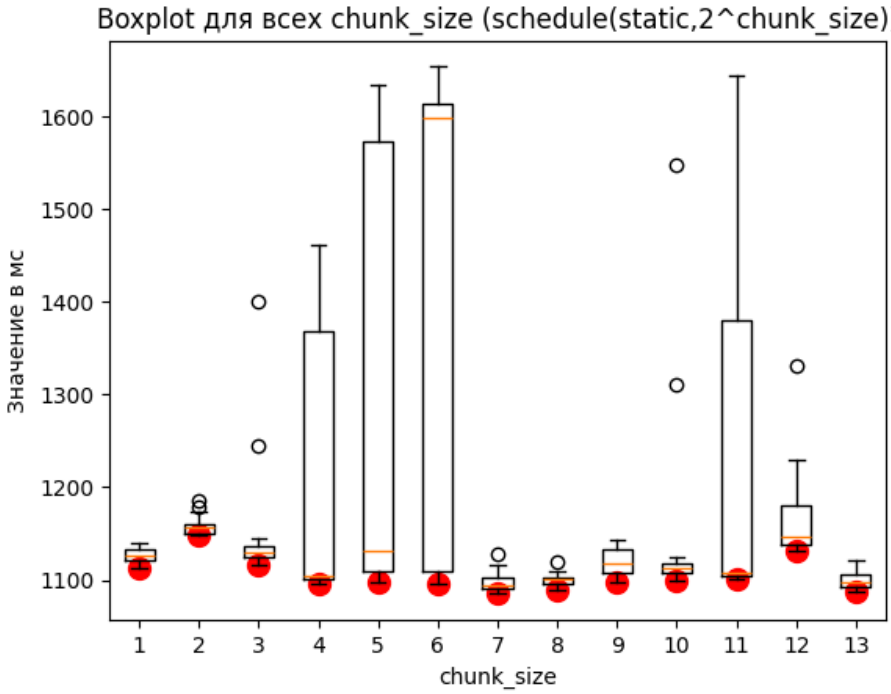


Рис. 5: Потоков - 8. Точек - 100_000_000

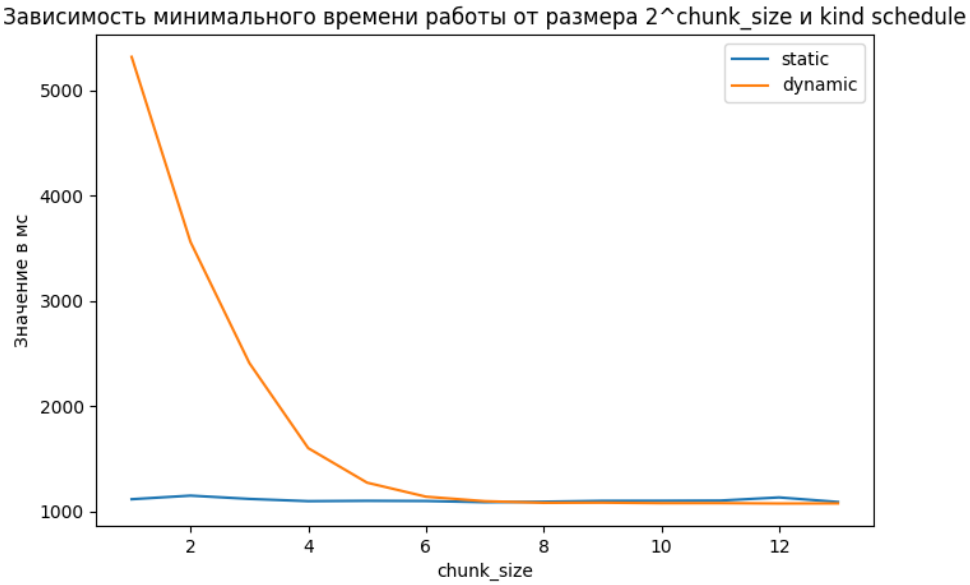


Рис. 6: Потоков - 8. Точек - 100_000_000

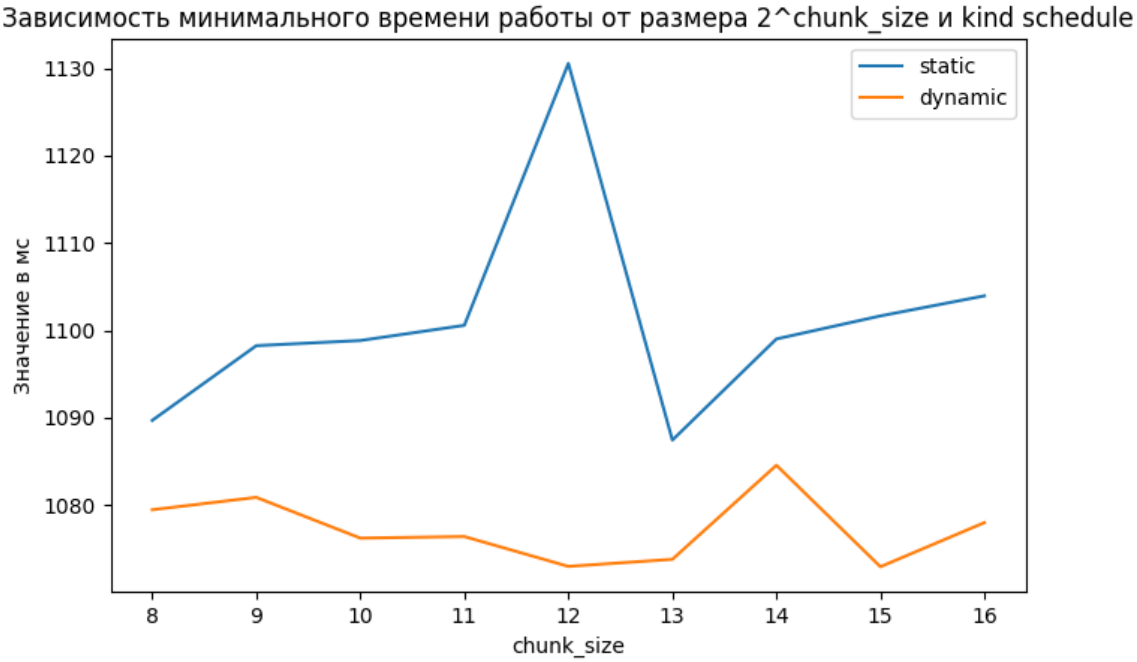


Рис. 7: Потоков - 8. Точек - 100_000_000

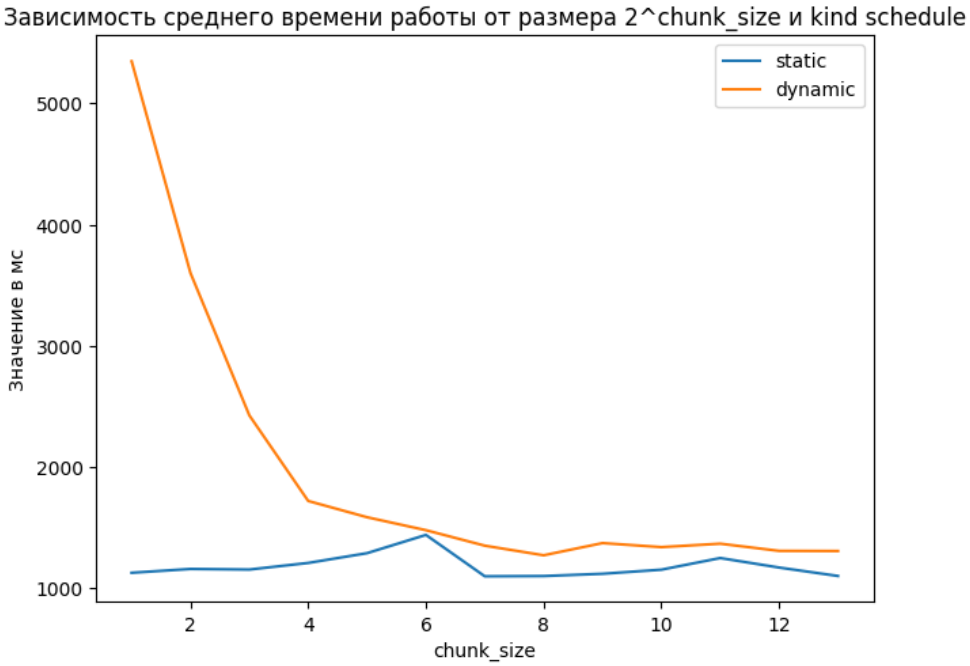


Рис. 8: Потоков - 8. Точек - 100_000_000

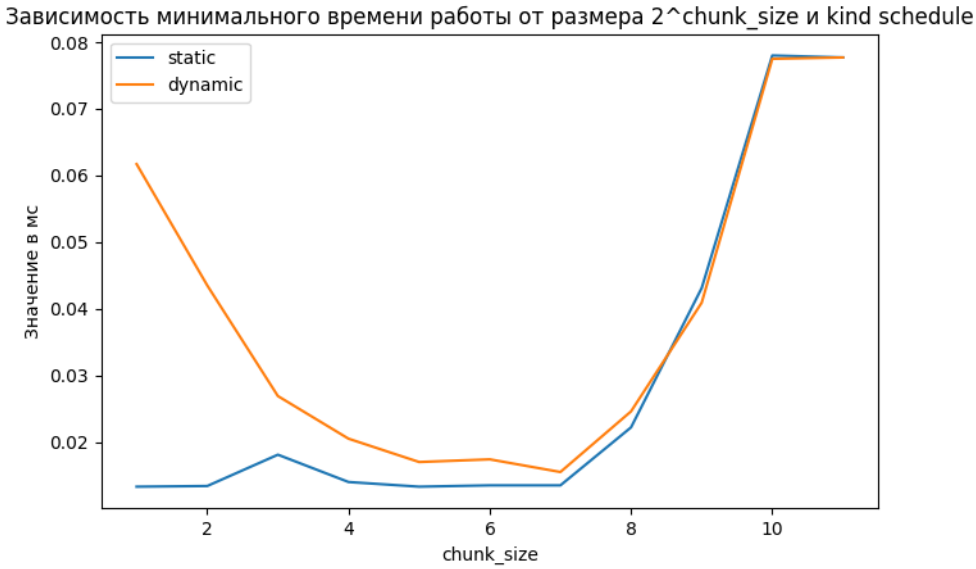


Рис. 9: Потоков - 8. Точек - 1000

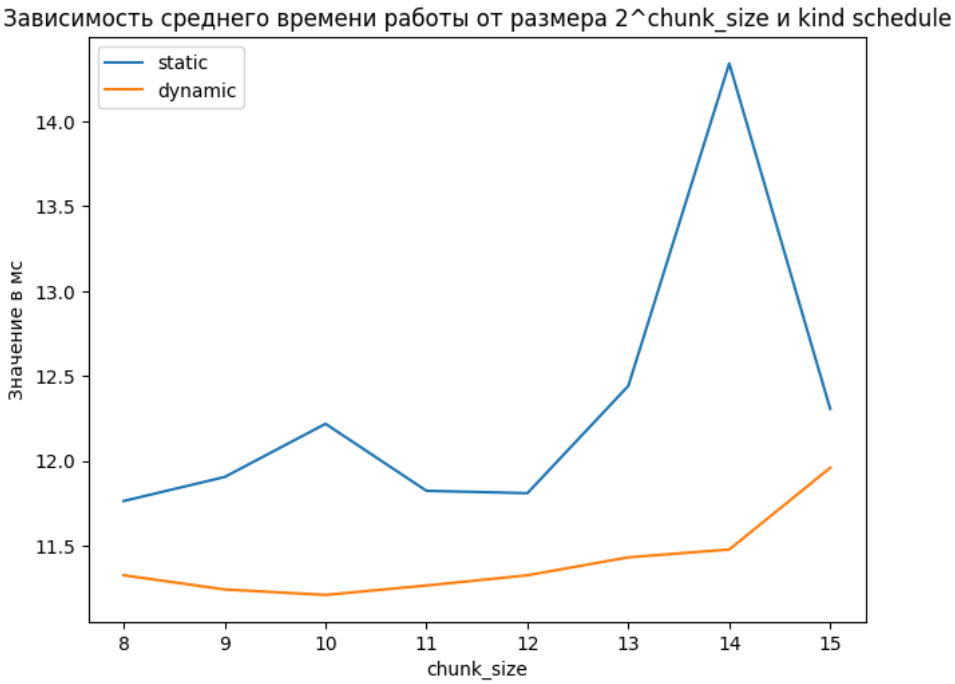


Рис. 10: Потоков - 8. Точек - 1_000_000

5 Приложение

Листинг 1: Код с использованием cstdlib (easy)

```
#include <iostream>
#include <cstdlib>
#include <ctime>

pair<double, int> calculateWithOpenMP(double radius, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
    int size_chunk = max(1, int(pow(2, (int)(ceil(1.6 * log10(totalPoints))))
    / 8 * max(1, numThreads)));
    #pragma omp parallel
    {
        int localCount = 0;
        int thread_id = omp_get_thread_num();
        srand(100 + thread_id);
        numThreads = omp_get_num_threads();
        #pragma omp for schedule(dynamic, size_chunk)
        for (int i = 0; i < totalPoints; i++) {
            double x = (double)rand() / RAND_MAX * radius;
            double y = (double)rand() / RAND_MAX * radius;

            if ((x * x + y * y) <= (radius * radius)) {
                localCount++;
            }
        }
        #pragma omp atomic
        pointsInsideCircle += localCount;
    }
    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints)
    * (4 * radius * radius), numThreads);
}
```

Листинг 2: Код с использованием cstdlib (normal)

```
#include <iostream>
#include <cstdlib>
#include <ctime>

pair<double, int> calculateWithOpenMP(double radius, int totalPoints, int numThreads)

pair<double, int> calculateWithOpenMP(double diag, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
    int get_num = 0;

    #pragma omp parallel
    {
        int localCount = 0;
        int thread_id = omp_get_thread_num();
        srand(1000000 * thread_id);

        #pragma omp for schedule(static)
```

```
        for (int i = 1; i <= totalPoints; i++) {
            double x = (double)rand() / RAND_MAX * diag;
            double y = (double)rand() / RAND_MAX * diag;
            double z = (double)rand() / RAND_MAX * diag;

            if (abs(x) + abs(y) + abs(z) <= diag) {
                localCount++;
            }
        }

        #pragma omp atomic
        pointsInsideCircle += localCount;
    }

    #pragma omp parallel
    {
        get_num = omp_get_num_threads();
    }

    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints) *
        (diag * diag * diag) * 8
        , get_num);
}
```

Листинг 3: Код с использованием random

```
#include <iostream>
#include <random>
#include <omp.h>
#include <cmath>

using namespace std;

pair<double, int> calculateWithOpenMP(double radius, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
    int size_chunk = max(1, static_cast<int>(pow(2,
        static_cast<int>(ceil(1.6 * log10(totalPoints)))) / 8 * max(1, numThreads)));
    int get_num = 0;

    #pragma omp parallel
    {
        int localCount = 0;
        random_device rd;
        mt19937 gen(rd());
        uniform_real_distribution<double> dis(0.0, radius);
        get_num = omp_get_num_threads();

        #pragma omp for schedule(dynamic, size_chunk)
        for (int i = 0; i < totalPoints; i++) {
            double x = dis(gen);
            double y = dis(gen);

            if ((x * x + y * y) <= (radius * radius)) {
```

```
                localCount++;
            }
        }
        #pragma omp atomic
        pointsInsideCircle += localCount;
    }
    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints)
        * (4 * radius * radius), get_num);
}
```

Листинг 4: Код с использованием random (normal) (float)

```
pair<double, int> calculateWithOpenMP(float diag, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
    int get_num = 0;
#pragma omp parallel
    {
        int localCount = 0;
        get_num = omp_get_num_threads();
        random_device rd;
        mt19937 gen(rd());
        uniform_real_distribution<float> dis(0.0, diag);

#pragma omp for schedule(static)
        for (int i = 1; i <= totalPoints; i++) {
            float x = dis(gen);
            float y = dis(gen);
            float z = dis(gen);

            if (abs(x) + abs(y) + abs(z) <= diag) {
                localCount++;
            }
        }

#pragma omp atomic
        pointsInsideCircle += localCount;
    }

    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints) *
        (diag * diag * diag) * 8
        , get_num);
}
```

Листинг 5: lcg-gen

```
unsigned long long lcg_rand(unsigned long long seed, unsigned long long a,
unsigned long long c, unsigned long long m) {
    return (a * seed + c) % m;
}
```

Листинг 6: Код с использованием lcg-gen (normal) (double)

```
pair<double, int> calculateWithOpenMP(float diag, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
```

```
    int get_num = 0;
    unsigned long long seed;
    unsigned long long a = 1664525;
    unsigned long long c = 1013904223;
    unsigned long long m = 4294967296;

    #pragma omp parallel private(seed)
    {
        int localCount = 0;
        get_num = omp_get_num_threads();
        seed = omp_get_thread_num()*10;

        #pragma omp for schedule(static)
        for (int i = 1; i <= totalPoints; i++) {
            seed = lcg_rand(seed, a, c, m);
            double x = static_cast<double>(seed) / m;

            seed = lcg_rand(seed, a, c, m);
            double y = static_cast<double>(seed) / m;

            seed = lcg_rand(seed, a, c, m);
            double z = static_cast<double>(seed) / m;

            if (x + y  + z <= 1) {
                localCount++;
            }
        }
        #pragma omp atomic
        pointsInsideCircle += localCount;
    }
    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints) *
        (diag * diag * diag) * 8
        , get_num);
}
```

Листинг 7: Весь код

```
#include <cstdlib>
#include <omp.h>
#include <iostream>
#include <ctime>
#include <fstream>
#include <cmath>
#include <random>
#include <vector>
#include <cstdio>

using namespace std;

unsigned long long lcg_rand(unsigned long long seed, unsigned long long a,
unsigned long long c, unsigned long long m) {
    return (a * seed + c) % m;
}
```

```
pair<double, int> calculateWithOpenMP(float diag, int totalPoints, int numThreads) {
    int pointsInsideCircle = 0;
    int get_num = 0;
    unsigned long long seed;
    unsigned long long a = 1664525;
    unsigned long long c = 1013904223;
    unsigned long long m = 4294967296;

    #pragma omp parallel private(seed)
    {
        int localCount = 0;
        get_num = omp_get_num_threads();
        seed = omp_get_thread_num()*10;

        #pragma omp for schedule(static)
        for (int i = 1; i <= totalPoints; i++) {
            seed = lcg_rand(seed, a, c, m);
            double x = static_cast<double>(seed) / m;

            seed = lcg_rand(seed, a, c, m);
            double y = static_cast<double>(seed) / m;

            seed = lcg_rand(seed, a, c, m);
            double z = static_cast<double>(seed) / m;

            if (x + y  + z <= 1) {
                localCount++;
            }
        }


        #pragma omp atomic
        pointsInsideCircle += localCount;
    }

    return make_pair((static_cast<double>(pointsInsideCircle) / totalPoints) *
        (diag * diag * diag) * 8
        , get_num);
}


double calculateSequentially(double diag, int totalPoints) {
    int pointsInsideCircle = 0;
    unsigned long long seed = 10;
    unsigned long long a = 1664525;
    unsigned long long c = 1013904223;
    unsigned long long m = 4294967296;

    for (int i = 0; i < totalPoints; i++) {
        seed = lcg_rand(seed, a, c, m);
        double x = static_cast<double>(seed) / m;
```

```
        seed = lcg_rand(seed, a, c, m);
        double y = static_cast<double>(seed) / m;

        seed = lcg_rand(seed, a, c, m);
        double z = static_cast<double>(seed) / m;

        if (x + y + z <= 1) {
            pointsInsideCircle++;
        }
    }

    return (static_cast<double>(pointsInsideCircle) / totalPoints) *
        (diag * diag * diag) * 8;
}



int main(int argc, char* argv[]) {
    setlocale(LC_ALL, "rus");
    setlocale(LC_ALL, "C");

    if (argc < 4) {
        cerr << "There are not enough arguments. Usage: " << argv[0]
        << " <number of streams> <input filename> <output filename>" << endl;
        return 1;
    }

    int numThreads = atoi(argv[1]);
    string inputFileName = argv[2];
    string outputFileName = argv[3];

    int totalPoints;
    double p1[3], p2[3], p3[3];
    FILE* file;


    file = fopen(inputFileName.c_str(), "r");

    if (file == NULL) {
        printf("Error openning file");
        return 1;
    }

    fscanf(file, "%d\n", &totalPoints);
    fscanf(file, "(%lf %lf %lf)\n", &p1[0], &p1[1], &p1[2]);
    fscanf(file, "(%lf %lf %lf)\n", &p2[0], &p2[1], &p2[2]);
    fscanf(file, "(%lf %lf %lf)\n", &p3[0], &p3[1], &p3[2]);

    fclose(file);

    double r12 = sqrt(pow(p1[0] - p2[0], 2) + pow(p1[1] - p2[1], 2)
```

```
+ pow(p1[2] - p2[2], 2));
double r13 = sqrt(pow(p1[0] - p3[0], 2) + pow(p1[1] - p3[1], 2)
+ pow(p1[2] - p3[2], 2));
double r23 = sqrt(pow(p2[0] - p3[0], 2) + pow(p2[1] - p3[1], 2)
+ pow(p2[2] - p3[2], 2));

double diag = 0;
if (abs(r12 - r13) < 0.001) {
    diag = r23 / 2;
}
else if (abs(r12 - r23) < 0.001) {
    diag = r13 / 2;
}
else {
    diag = r12 / 2;
}
double realVolumeOct = pow(diag, 3) / 2 / 3 * 8;

double start_time, end_time;
double volumeOct = 0;

if (numThreads == -1) {
    start_time = omp_get_wtime();
    volumeOct = calculateSequentially(diag, totalPoints);
    end_time = omp_get_wtime();
    printf("Time (%i thread(s)): %g ms\n", 0, (end_time - start_time) * 1000);

}
else if (numThreads >= 0) {
    if (numThreads == 0) {
    }
    else {
        omp_set_num_threads(numThreads);
    }
    int numThreadsToPrint = 0;
    start_time = omp_get_wtime();
    pair<double, int> result = calculateWithOpenMP(diag, totalPoints, numThreads);
    volumeOct = result.first;
    numThreadsToPrint = result.second;
    end_time = omp_get_wtime();

    printf("Time (%i thread(s)): %g ms\n", numThreadsToPrint,
    (end_time - start_time) * 1000);

}
else {
    cerr << "Unsupported number of threads" << endl;
    return 1;
}


FILE* file2 = fopen(outputFileName.c_str(), "w");
```

```
    if (file2 == NULL) {
        printf("Error openning file");
        return 1;
    }

    try {
        fprintf(file2, "%g %g\n", realVolumeOct, volumeOct);
    }
    catch (const exception& e) {
        cerr << "Error writing the file: " << e.what() << endl;
        return 1;
    }
    fclose(file2);


    return 0;

}
```