**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

# Отчёт по лабораторной работе №4 по курсу "Анализ алгоритмов"

**Тема** Распараллеливание алгоритма поворота массива точек в пространстве

**Студент** Ивахненко Д.А.

**Группа** ИУ7-56Б

**Преподаватель** Волкова Л.Л.

Москва — 2021 г.

# Содержание

# Введение

Многопоточность - способность центрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколь ко процессов или потоков, соответствующим образом поддерживаемых операционной системой

Другими словами, процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

*Процесс* - это выполняющаяся программа и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и т.д. *Поток* - это наименьшая последовательность запрограммированных команд, которые могут управляться планировщиком независимо. *Задача* - абстрактная концепция работы, которая должна быть выполнена.

Многопоточная парадигма стала более популярной с конца 1990-х годов, поскольку усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились.

Смысл многопоточности - квазимногозадачность на уровне одного исполняемого процесса, то есть все потоки выполняются в адресном пространстве процесса. Кроме этого, все потоки процесса имеют не только общее адресное пространство, но и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Преимущества многопоточности:

- отзывчивость: один из потоков может обеспечивать быстрый от клик, пока другой поток заблокирон или работает;
- разделение ресурсов: поток разделяет код, данные;
- экономичность: переключение контекста значительно быстрее, чем у процессов;
- масштабирование: однопоточные процессоры могут выполняются только на одном процессоре, многопоточные в многопроцессорных систе мах - на нескольких процессорах.

Недостатки многопоточности:

- необходимо заботиться о корректном распараллеливании задачи, иначе это может привести к блокировкам и ошибкам доступа;
- необходимо заботиться о разделении задачи на подзадачи сбалан сированно, чтобы использовать ресурсы максимально эффективно;
- нужна синхронизация потоков, если одна подзадача зависит от результата другой;
- сложнее тестирование и отладка;
- неправильное разделение данных приводит к ошибкам.

Целью работы является изучение и реализация принципов многопоточности на примере алгоритма поворота фигурыв в пространстве, представленной в виде массива точек. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить понятие параллельных вычислений, а также принцип работы алгоритма поворота массива точек в пространстве;
- разработать схему рассматриваемого алгоритма в последовательном и параллельном вариантах;
- описать используемые структуры данных;
- реализовать последовательный и параллельный алгоритм поворота массива точек в пространстве;
- протестировать разработанное ПО;
- сравнить временные характеристики реализованных версий алгоритма экспериментально;
- на основании проделанной работы сделать выводы.

# 1 Аналитическая часть

В данной лабораторной работе многопоточность изучается на примере алгоритма поворота в пространстве фигуры, представленной в виде массива точек;

Эта задача является весьма актуальной, т.к. компьютерная графика стала неотъемлемой частью повседневной интернет-жизни человека, и существует потребность в быстром рендеринге изображения, например, при анимации поворота фигуры.

Как известно, в экранной плоскости изображение представляет из себя набор пикселей (точек). Очевидно, что какая-либо фигура - это тоже набор точек экранной плоскости, и для поворота фигуры требуется над каждой ее точкой произвести преобразование для получения новой позиции.

Если использовать один поток для рендеринга изображения, то при большом количестве и сложности фигур изображение будет генерироваться ощутимо долго, что будет приносить человеку дискомфорт при восприятии, однако если распараллелить этот процесс, т.е. параллельно генерировать части изображения (т.к. эта операция выполняется независимо для каждой точки), то это может дать коллосальной прирост производительности.

Перейдем к рассмотрению алгоритма поворота в пространстве[1]

## 1.1 Алгоритм поворота точек в пространстве

Любое вращение в трёхмерном пространстве может быть представлено как композиция поворотов вокруг трёх ортогональных осей (например, вокруг осей декартовых координат). Этой композиции соответствует матрица, равная произведению соответствующих трёх матриц поворота [2]

Матрицами вращения вокруг оси декартовой системы координат на угол $\alpha$ в трёхмерном пространстве с неподвижной системой координат являются:

Вращение вокруг оси $x$ (1.1)

$$M_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \tag{1.1}$$

Вращение вокруг оси $y$ (1.2)

$$M_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}, \tag{1.2}$$

Вращение вокруг оси $z$ (1.3)

$$M_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{1.3}$$

Положительным углам при этом соответствует вращение вектора против часовой стрелки в правой системе координат, и по часовой стрелке в левой системе координат, если смотреть против направления соответствующей оси[2]. Например, при повороте на угол $\alpha = 90^\circ$ вокруг оси $z$ ось $x$ переходит в $y$: $M_z(90^\circ) \cdot \mathbf{e}_x = \mathbf{e}_y$. Аналогично, $M_y(90^\circ) \cdot \mathbf{e}_z = \mathbf{e}_x$ и $M_x(90^\circ) \cdot \mathbf{e}_y = \mathbf{e}_z$.

Пусть необходимо повернуть точку $P(x, y, z)$ вокруг оси $Z$ на угол $\phi$. Изображение новой точки обозначим через $P'(x', y', z')$. Координаты новой точки будут рассчитываться согласно выражениию 1.4

$$\begin{cases} x' = x \cdot \cos\phi - y \cdot \sin\phi \\ y' = x \cdot \sin\phi + y \cdot \cos\phi \\ z' = z \end{cases} \tag{1.4}$$

Таким образом, распараллеливание будет заключаться в том, что массив точек будет разбиваться на подмассивы, для каждого из которых независимо от других будет решаться задача преобразования.

## 1.2 Вывод

Таким образом, к разрабатываемой программе предъявляются следующие требования:

1. Входными данными программы является текстовый файл, содержащий набор точек $x, y, z$, которые необходимо преобразовать и точку $x_c, y_c, z_c$,

относительно которой необходимо осуществить поворот, а также углы поворота по каждой из трех осей угол $\phi_x, \phi_y, \phi_z$, на который необходимо повернуть фигуру.

2. Выходными данными программы также является текстовый файл, содержащий набор преобразованных точек фигуры.

3. Необходимо предоставить пользователю возможность выбора количества потоков, которые будут использованы выполнения необходимых преобразований, в частности возможность выбрать только один поток выполнения.

4. Необходимо предоставить возможность работы в двух режимах: проведения эксперимента и ручного тестирования.

5. Программа должна корректно реагировать на ошибки пользователя и невалидные данные о фигуре или угле поворота; в таком случае программа должна сообщить пользователю о некорректности данных или невозможности преобразования;

# 2 Конструкторская часть

В данном разделе представлены последовательная и параллельная схемы алгоритма, описанного в аналитической части.

## 2.1 Схемы алгоритмов

На рисунке 2.1 представлена схема последовательного алгоритма поворота точек в пространстве.

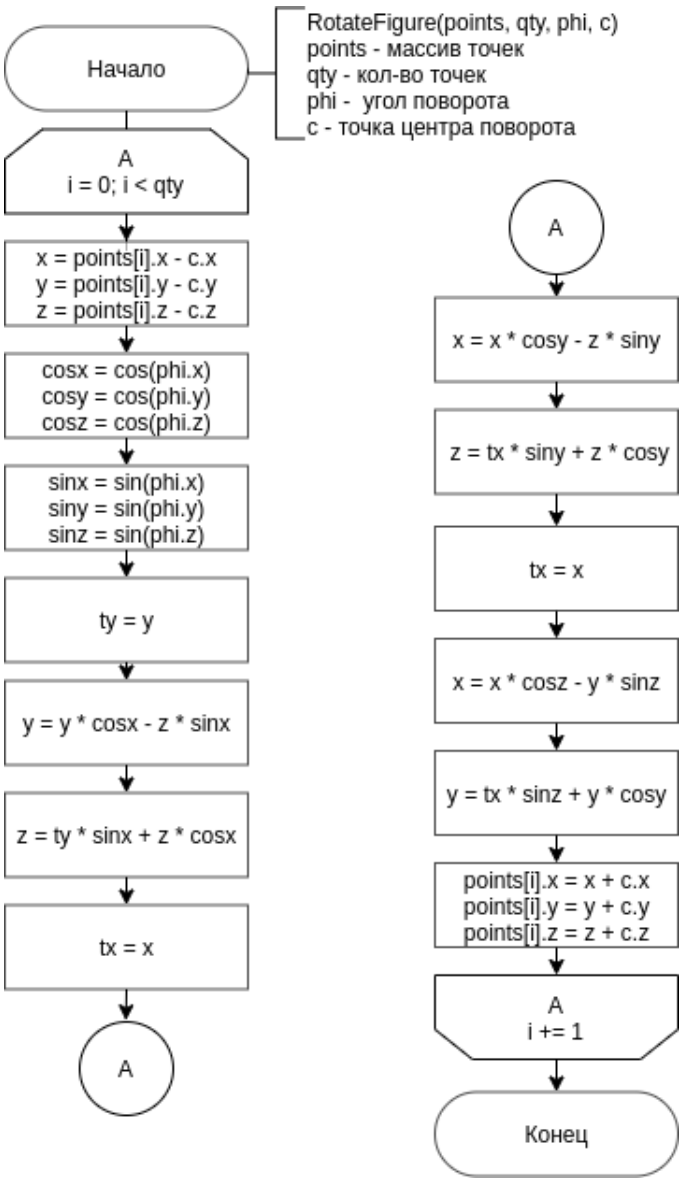


Рисунок 2.1 – Схема последовательного алгоритма поворота точек

На рисунке 2.2 представлена схема распараллеливания алгоритма поворота точек двумерного растра.

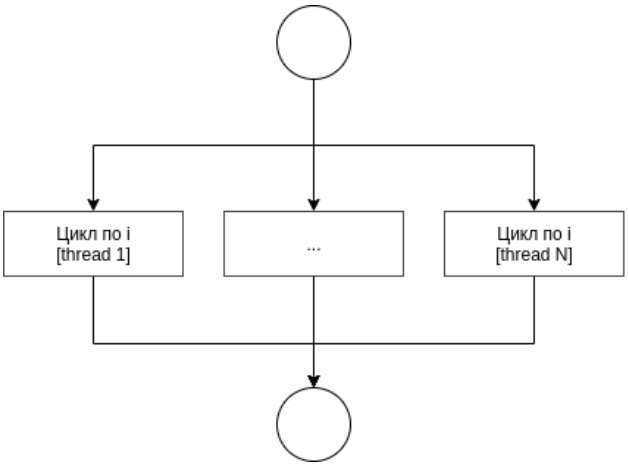


Рисунок 2.2 – Схема распараллеливания алгоритма поворота точек

На рисунке 2.3 представлена схема паралелльного алгоритма поворота точек в пространстве.

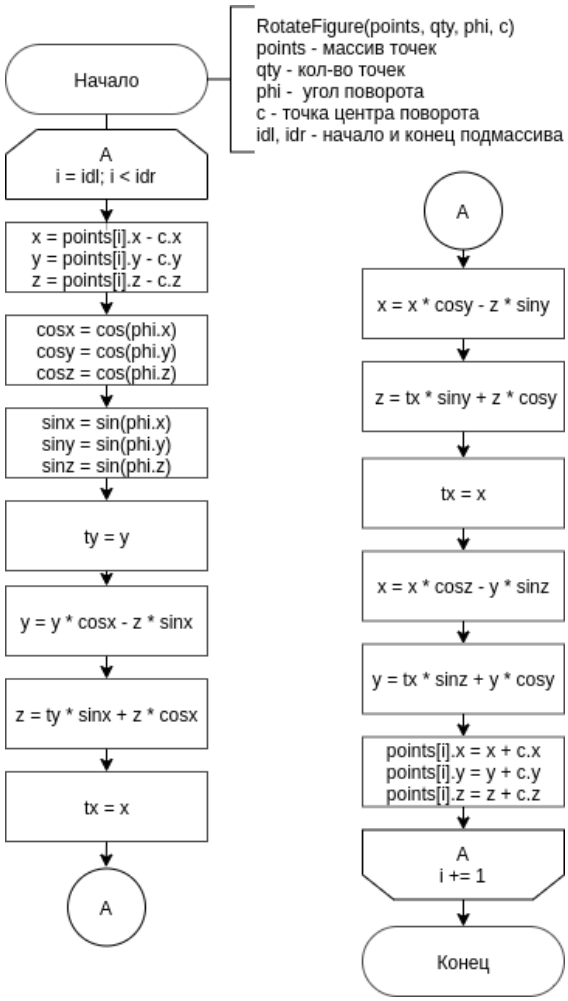


Рисунок 2.3 – Схема параллельного алгоритма поворота точек

На рисунке 2.4 представлена схема функции создания потоков и запуска параллельных реализаций вычислений


Рисунок 2.4 – Схема параллельного алгоритма поворота точек

## 2.2 Выделение классов эквивалентности

Для тестирования программы были выделены следующие классы эквивалентности:

- поворот на нулевой угол;
- поворот на положительный угол;
- поворот на отрицательный угол;
- ни одной точки на вход;
- одна точка на вход;
- несколько точек на вход;
- выбран 1 поток исполнения;
- выбрано несколько потоков исполнения;

- выбрано 0 потоков исполнения;
- некорректные координаты точки;
- некорректный угол поворота.

## 2.3 Вывод

На основе теоретических данных, полученных из аналитического раздела, были разработаны схемы требуемых алгоритмов, а также выделены классы эквивалентности для последующего функционального тестирования.

# 3 Технологическая часть

В данном разделе приведны средства реализации, результаты тестирования и листинги кода.

## 3.1 Средства реализации

Для реализации распараллеливания алгоритма был выбран язык программирования `C++` [3], что обусловлено поддержкой нативных потоков ядра, а также простотой интерфейса библиотеки для взаимодейсвтия с ними. В качестве среды разработки был выбран `CLion`[4], как наиболее популярная IDE для `C++`.

## 3.2 Функциональное тестирование

В соответствии с выделеными классами эквивалентности были разработаны тесты, представленные в таблице 3.1.

Таблица 3.1 – Тестирование функций

| Массив точек | Центр | Углы, ° | Потоки | Ожидаемый рез. | Реальный рез. |
|---|---|---|---|---|---|
| $[(1,0)]$ | $(0,0)$ | $(0,0,0)$ | 1 | $[(1,0)]$ | $[(1,0)]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,90)$ | 1 | $[(0,1)]$ | $[(0,1)]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,-90)$ | 1 | $[(-1,0)]$ | $[(-1,0)]$ |
| $[]$ | $(5,2)$ | $(1,1,1)$ | 1 | $[]$ | $[]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,0)$ | 1 | $[(1,0)]$ | $[(1,0)]$ |
| $[(1,0),(1,0)]$ | $(0,0)$ | $(0,0,90)$ | 1 | $[(0,1),(0,1)]$ | $[(0,1),(0,1)]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,0)$ | 1 | $[(1,0)]$ | $[(1,0)]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,0)$ | 8 | $[(1,0)]$ | $[(1,0)]$ |
| $[(1,0)]$ | $(0,0)$ | $(0,0,0)$ | 0 | Ошибка | Ошибка |
| $[(q,0)]$ | $(2,1)$ | $(0,0,0)$ | 1 | Ошибка | Ошибка |
| $[(0,0)]$ | $(1,2)$ | $(0,q,0)$ | 1 | Ошибка | Ошибка |

Все тесты пройдены успешно.

## 3.3 Реализация алгоритмов

В листинге 3.1 представлена реализация алгоритма поворота точек.

Листинг 3.1 – Реализация алгоритма поворота точек

```
struct double3 {
    double x;
    double y;
    double z;
};

void rotate_points(std::vector<double3>& points, double3& center,
   const double3& rot_data, int idx0, int idx1) {
    for (int i = idx0; i <= idx1; ++i) {
        translate_point(points[i], {-center.x, -center.y, -center.z
           });
        rotate_point(points[i], rot_data);
        translate_point(points[i], {center.x, center.y, center.z});
    }
}

void rotate_point(double3& point, const double3& rot_data) {
    rotate_x_axis(point, rot_data.x);
    rotate_y_axis(point, rot_data.y);
    rotate_z_axis(point, rot_data.z);
}


void translate_point(double3& p, const double3& tr_data) {
    p.x += tr_data.x;
    p.y += tr_data.y;
    p.z += tr_data.z;
}
```

В листинге 3.2 представлена реализация алгоритима поворота точки вокруг каждой из осей.

Листинг 3.2 – Реализация алгоритма поворота точек

```
void rotate_x_axis(double3& p, const double angle) {
    double cos_theta = cos(to_rad(angle));
    double sin_theta = sin(to_rad(angle));
```

```
    double temp_y = p.y;
    p.y = p.y * cos_theta - p.z * sin_theta;
    p.z = temp_y * sin_theta + p.z * cos_theta;
}

static void rotate_y_axis(double3& p, const double angle) {
    double cos_theta = cos(to_rad(angle));
    double sin_theta = sin(to_rad(angle));

    double temp_x = p.x;
    p.x = p.x * cos_theta - p.z * sin_theta;
    p.z = temp_x * sin_theta + p.z * cos_theta;
}

static void rotate_z_axis(double3& p, const double angle) {
    double cos_theta = cos(to_rad(angle));
    double sin_theta = sin(to_rad(angle));

    double temp_x = p.x;
    p.x = p.x * cos_theta - p.y * sin_theta;
    p.y = temp_x * sin_theta + p.y * cos_theta;
}
```

В листинге 3.3 представлена реализация алгоритма распределения точек по потокам.

Листинг 3.3 – Реализация алгоритма поворота точек

```
void handle(vector<double3>& points, double3& center, const double3&
    rotate_data, int num_of_threads = 1) {
    int points_per_thread = int(points.size()) / num_of_threads;
    int remaining_data = int(points.size()) - points_per_thread *
       num_of_threads;
    vector<thread> threads;

    int last = -1, from, to;
    for (int i = 0; i < num_of_threads; ++i) {
        from = last + 1;
        if (i < remaining_data) {
            to = last + points_per_thread + 1;
            last += points_per_thread + 1;
        } else {
            to = last + points_per_thread;
```

```
14                last += points_per_thread;
15            }
16            threads.push_back(thread(rotate_points, points, center,
                  rotate_data, from, to));
17        }
18
19        for (auto& t: threads)
20            t.join();
21    }
```

## 3.4 Вывод

В данном разделе были реализованы последовательная и распараллеленная версии алгоритма поворота точек фигуры в пространстве. Данные реализации алгоритмов были протестированы функциональными тестами, построенными на основе выделенных классов эквивалентности.

# 4 Исследовательская часть

## 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент:

- операционная система: Ubuntu[5] Linux x86_64;
- память: 16 GiB;
- процессор: AMD Ryzen™ 7 4700U[6].

## 4.2 Проведение эксперимента

Для каждого числа потоков было произведено 10 замеров процессорного времени, после чего определено среднее время выполнения алгоритма.

В таблице 4.1 представлены замеры процессорного времени (в мс) для массива точек размера 2073600 (стандартное разрешение 1920 x 1080).

Результаты замеров представлены в табл. 4.1 и на рис. 4.1.

Таблица 4.1 – Время выполнения алгоритмов (в мс) для отсортированного массива

| Кол-во потоков | Время выполнения (мс) |
|---|---|
| 1 | 157.784 |
| 2 | 86.275 |
| 4 | 49.0306 |
| 8 | 31.3017 |
| 16 | 30.8065 |
| 32 | 28.8583 |
| 64 | 27.9632 |

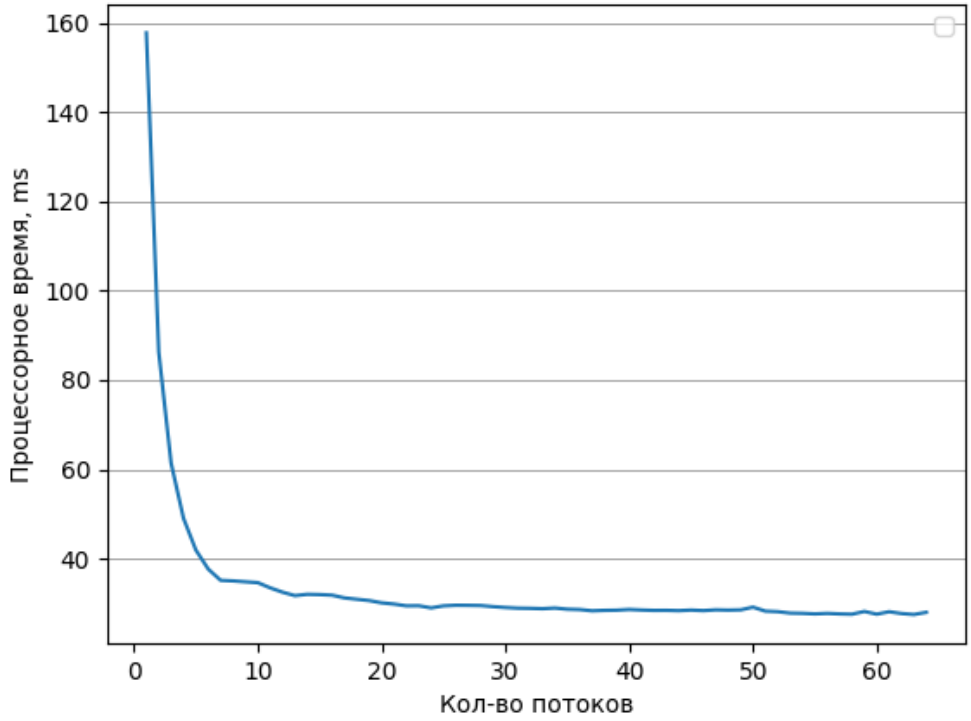


Рисунок 4.1 – Зависимость времени работы алгоритма от кол-ва потоков

В таблице 4.2 и на рисунке 4.1 представлены замеры процессорного времени (в мс) для количества потоков, равного количеству логических ядер процессора.

Таблица 4.2 – Замеры процессорного времени для кол-ва потоков, равного кол-ву логических ядер процессора

| Кол-во точек | Последовательный алгоритм | Параллельный алгоритм |
|---|---|---|
| 2000000 | 156.657 | 31.310 |
| 1500000 | 113.098 | 24.814 |
| 1000000 | 79.139 | 16.042 |
| 500000 | 36.657 | 7.952 |

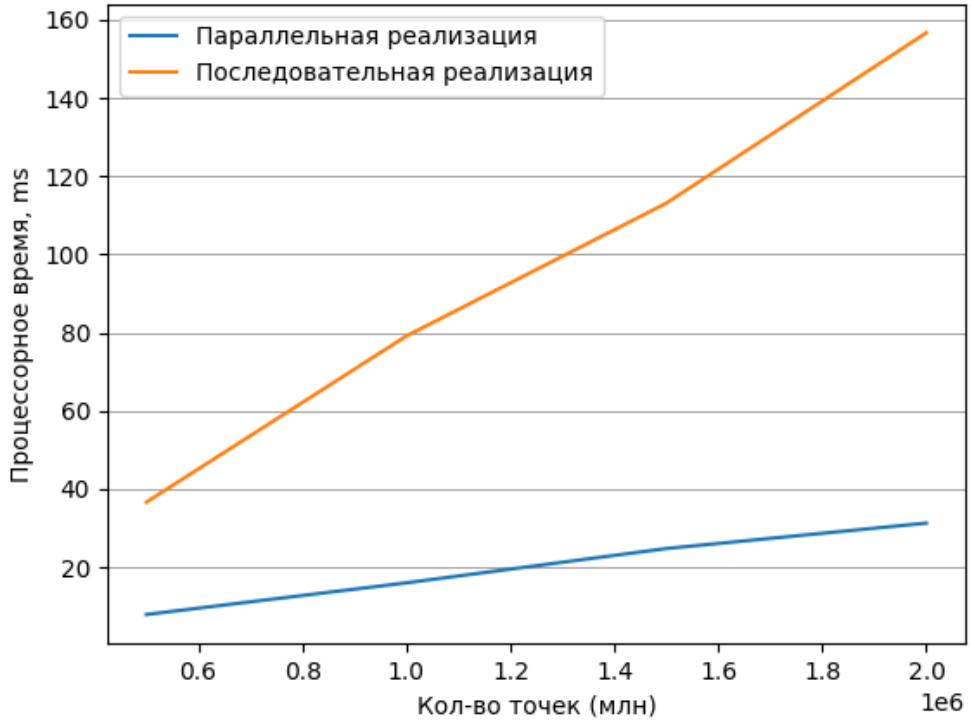


Рисунок 4.2 – Зависимость процессорного времени от размерности массива (количество потоков равно 16)

## 4.3 Вывод

Из полученных выше результатов замеров процессорного времени можно сделать вывод, что использование многопоточности однозначно может дать существенной прирост эффективности. В данной лабораторной работе в качестве примера рассматривался алгоритм поворота массива точек фигуры в пространстве, и при использовании многопоточности удалось добиться как минимум 6-кратного увеличения производительности.

Даже при использовании двух потоков, вместо одонго, производительность увеличилась почти в 2 раза, однако при количестве потоков, равном количеству физических ядер (8 ядер), рост производительности начинает замедляться, а после количества потоков, равного 16 - столько логических ядер имеет процессор, время выполнения алгоритма практически стабильно.

Максимальное количество потоков, при которых выполнялись замеры - 64 и при этом количестве производительность выросла в 6 раз, однако нельзя утверждать, что так будет всегда - если использовать слишком много потоков

и часть данных для обработки в результате будет малой, то больше времени будет тратиться на создание и завершение потока, чем на обработку части информации.

Также можно сделать вывод, что использовать 1 поток неэффективно, т.к. по сути это эквивалентно обыкновенному последовательному алгоритму, однако сверх этого тратятся ресурсы системы на создание и закрытие потока, и в результате программа работает медленее.

# Заключение

В ходе выполнения лабораторной работы была выполнены поставленные задачи, а именно:

- рассмотрено и изучено понятие параллельных вычислений, а также принцип работы алгоритма поворота массива точек в пространстве;
- разработана схема рассматриваемого алгоритма в последовательном и параллельном вариантах;
- реализованы последовательный и параллельный алгоритмы поворота массива точек в пространстве;
- выделены классы эквивалентности для данного алгоритма;
- на основе выделенных классов эквивалентности разработаны функциональные тесты для программы;
- экспериментально проведен сравнительный анализ быстродействия алгоритмов.

Экспериментально были подтверждены различия между последовательной и параллельной реализациями алгоритма:

Из полученных выше результатов замеров процессорного времени можно сделать вывод, что использование многопоточности однозначно может дать прирост эффективности.

В данной лабораторной работе в качестве примера рассматривался алгоритм поворота массива точек фигуры в пространстве, и при использовании много поточности удалось добиться как минимум 6-кратного увеличения производительности.

Даже при использовании двух потоков, вместо одонго, производительность увеличилась почти в 2 раза, однако при количестве потоков, равном количеству физических ядер (8 ядер), рост производительности начинает замедляться, а после количества потоков, равного 16 - столько логических ядер имеет процессор, время выполнения алгоритма практически стабильно.

Максимальное количество потоков, при которых выполнялись замеры - 64 и при этом количестве производительность выросла в 6 раз, однако нельзя

утверждать, что так будет всегда - если использовать слишком много потоков и часть данных для обработки в результате будет малой, то больше времени будет тратиться на создание и завершение потока, чем на обработку части информации.

Также можно сделать вывод, что использовать 1 поток неэффективно, т.к. по сути это эквивалентно обыкновенному последовательному алгоритму, однако сверх этого тратятся ресурсы системы на создание и закрытие потока, и в результате программа работает медленее.

Таким образом, можно сделать вывод, что количество потоков же лательно выбирать взвешенно - надо избегать напрасной траты ресурсов системы, например, не создавать один поток или не создавать слишком много потоков. Нужно отследить, при каком минимальном количестве потоков достигается минимально приемлемый выигрыш во времени, т.е. выбрать пик производительности при возможном минимуме потоков, и в дальнейшем использовать это число.

Также следует учитывать, что наиболее эффективно распаралле ливать анализ больших данных, т.к. маленькие задачи и так выполнятся достаточно быстро с учетом высокой производительности современных компьютеров.

# Литература

[1] Д. Роджерс Дж. Адамс. Повороты в пространстве // Алгоритмические основы машинной графики. - М.: Мир, 2001. С. 143–144.

[2] Д. Роджерс Дж. Адамс. Матрицы поворота // Математические основы машинной графики. - М.: Мир, 2001. С. 20–24.

[3] C++ Programming Language [C++20] [Электронный ресурс]. Режим доступа: `https://devdocs.io/cpp/` (дата обращения: 28.09.2021).

[4] Кросс-платформенная IDE для C и C++ [Электронный ресурс]. Режим доступа: `https://www.jetbrains.com/ru-ru/clion/` (дата обращения: 28.09.2021).

[5] Ubuntu 20.04.3 LTS [Электронный ресурс]. Режим доступа: `https://ubuntu.com/download/desktop` (дата обращения: 28.09.2020).

[6] Проессор AMD Ryzen™ 7 4700U [Электронный ресурс]. Режим доступа: `https://www.amd.com/ru/products/apu/amd-ryzen-7-4700u` (дата обращения: 28.09.2021).