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»

Отчет по лабораторной работе №4 по курсу "Анализ алгоритмов"

Тема Многопоточное программирование

Студент Цветков И.А.

Группа ИУ7-53Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2021 г.

Содержание

Введение

В процессе развития компьютерных систем программисты стали все чаще сталкиваться с задачами, где трубется большое количество различных вычислений. Все вычисления происходят на одном процессоре. Как одно из решений, вычисления можно проводить на разных компьютерах. Но балгодаря развитию процессоров, было предложено распараллелить вычисления на одном процессоре, используя потоки, каждый из которых запусакется на отдельном логическом ядре. Тем самым появился термин "многопоточность который заключается в разделении процессов, выполняя их параллельно.

Целью данной работы является изучение параллельных вычислений на основе построения пучка отрезков по алгоритму Брезенхема. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить основы распараллеливания вычислений;
- реализовать алгоритм построения пучка отрезков по Брезенхему с использованием многопоточности и без;
- провести сравнительный анализ по времени на одной длине пучка и различном количестве потоков, а также без многопочности;
- провести сравнительный анализ по времени при разной длине пучка с использованием многопоточности и без;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В этом разделе будет представлена информация по поводу многопоточности, а также теоретически описан алгоритм Брезенхема для построения отрезка, построение пучка которых будет распараллелена в данной лабораторной работе.

1.1 Многопоточность

Многопоточность [1] – способность центрального процессора или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой.

Упомянем важные определения:

- **процесс** – это программа в ходе своего выполнения. Когда мы выполняем программу или приложение, запускается процесс. Каждый процесс состоит из одного или нескольких потоков;

- **поток** – это не что иное, как сегмент процесса. Потоки – исполняемые сущности, которые выполняют задачи, стоящие перед исполняемым приложением. Процесс завершается, когда все потоки заканчивают выполнение.

Каждый поток в процессе – это задача, которую должен выполнить процессор. Большинство процессоров сегодня умеют выполнять одновременно две задачи на одном ядре, создавая дополнительное виртуальное ядро. Это называется одновременная многопоточность или многопоточность $Hyper - Threading$, если речь о процессоре от Intel.

Эти процессоры называются многоядерными процессорами. Таким образом, двухъядерный процессор имеет 4 ядра: два физических и два виртуальных. Каждое ядро может одновременно выполнять только один поток.

Как упоминалось выше, один процесс содержит несколько потоков, и одно ядро процессора может выполнять только один поток за единицу времени. Если мы пишем программу, которая запускает потоки последовательно, то

есть передает выполнение в очередь одного конкретного ядра процессора, мы не раскрываем весь потенциал многоядерности. Остальные ядра просто стоят без дела, в то время как существуют задачи, которые необходимо выполнить. Если мы напишем программу таким образом, что она создаст несколько потоков для отнимающих много времени независимых функций, то мы сможем использовать другие ядра процессора, которые в противном случае пылились бы без дела. Можно выполнять эти потоки параллельно, тем самым сократив общее время выполнения процесса.

1.2 Алгоритм Брезенхема построения отрезка

Алгоритм Брезенхема [2] был предложен Джеком Е. Брезенхэмом в 1962 году и предназначен для рисования фигур точками на плоскости. Этот алгоритм находит широкое распространение в машинной графике для рисования линий на экране. Алгоритм определяет, какие точки двумерного растра необходимо закрасить.

Графическая интерпретация алгоритма Брезенхема представлена на рисунке 1.1.

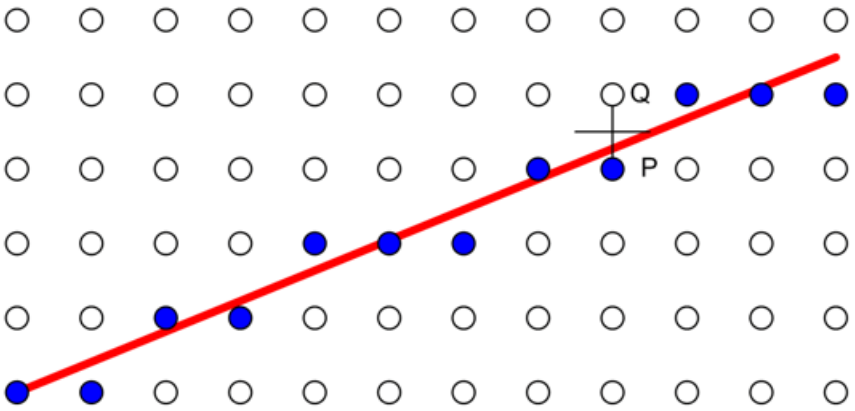

Рисунок 1.1 – Алгоритм Брезенхема

Для рисования прямых отрезков на плоскости с использованием алгоритма Брезенхема запишем уравнение прямой в общем виде

$$y = kx + b \tag{1.1}$$

или

$$f(x, y) = Ax + By + C = 0 \tag{1.2}$$

где коэффициенты A и B выражаются через коэффициенты k и b уравнения прямой. Если прямая проходит через две точки с координатами $(x_1; y_1)$ и $(x_2; y_2)$, то коэффициенты уравнения прямой определяются по формулам

$$A = y_2 - y_1 \tag{1.3}$$

$$B = x_1 - x_2 \tag{1.4}$$

$$C = y_1 \cdot x_2 - y_2 \cdot x_1 \tag{1.5}$$

Для любой растровой точки с координатами $(x_i; y_i)$ значение функция

$$f(x_i, y_i) = 0, \text{если точка лежит на прямой;} \tag{1.6}$$

$$f(x_i, y_i) > 0, \text{если точка лежит ниже прямой;} \tag{1.7}$$

$$f(x_i, y_i), \text{где i – номер отображаемой точки.} \tag{1.8}$$

Таким образом, одним из методов решения того, какая из точек P или Q будет отображена на следующем шаге, является сравнение середины отрезка $|P - Q|$ со значением функции $f(x, y)$. Если значение $f(x, y)$ лежит ниже средней точки отрезка $|P - Q|$, то следующей отображаемой точкой будет точка P, иначе — точка Q. Запишем приращение функции

$$df = A \cdot dx + B \cdot dy \tag{1.9}$$

После отображения точки с координатами (x_i, y_i) принимается решение о следующей отображаемой точке. Для этого сравниваются приращения dx и dy, характеризующие наличие или отсутствие перемещения по соответствующей координате. Эти приращения могут принимать значения 0 или 1. Следовательно, когда мы перемещаемся от точки вправо,

$$df = A, \tag{1.10}$$

когда мы перемещаемся от точки вправо и вниз, то

$$df = A + B, \tag{1.11}$$

когда мы перемещаемся от точки вниз, то

$$df = B \tag{1.12}$$

Нам известны координаты начала отрезка, то есть точки, заведомо лежащей на искомой прямой. Ставим туда первую точку и принимаем $f = 0$. От текущей точки можно сделать два шага — либо по вертикали (по горизонтали), либо по диагонали на один пиксель. Направление движения по вертикали или горизонтали определяется коэффициентом угла наклона. В случае если угол наклона меньше 45^{o}, и

$$|A| < |B| \tag{1.13}$$

с каждым шагом осуществляется движение по горизонтали или диагонали. Если угол наклона больше 45^{o}, с каждым шагом движение осуществляется по вертикали или диагонали.

1.3 Вывод

В данном разделе было рассмотрено понятие многопоточности, а также алгоритм Брезенхема для построения отрезка, распараллеливание пучка которого и будет рассматриваться в данной лабораторной работе.

Программа имеет графический интерфейс. На вход через поля ввода будет подаваться длина отрезка в пучке, а также – количество потоков, если выбран алгоритм с распараллеливанием. При неверном вводе – пустое поле ввода или значения меньше нуля – будет выведено сообщение об ошибке.

Реализуемое ПО дает возможность выбрать построение алгоритма с распараллеливанием или без него, а также выбрать количество потоков, по которым будет идти распараллеливание. Также имеется возможность провести

тестирование по времени для разного количества потоков или разной длины отрезков в пучке.

2 Конструкторская часть

В этом разделе будут представлено описание используемых типов данных, а также схемы алгоритмов построения пучка отрезков по методу Брезенхема с распараллеливанием и без него.

2.1 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие типы данных:

- количество потоков - целое число типа *int*;
- длина отрезка - целое число типа *int*;
- структура *request* - содержит информацию о типе алгоритма (с использованием многопоточности или без нее), необходимые параметры о канвасе, а также параметр *is_draw* - определяет, нужно ли отрисовывать на канвас спектр отрезков Брезенхема (не отрисовывается при замере времени, так как происходит множество прогонов);
- структура *beam_settings* - содержит информацию о спектре - длина отрезка в спектре, количество потоков при построении (если требуется);
- структура *point* - содержит информацию о точке - ее координаты по x и y.

2.2 Структура разрабатываемого ПО

В данном ПО будет реализован метод структурного программирования. Для взаимодействия с пользователем будет разработан интерфейсом, на котором можно будет задать нужные параметры, выбрать - распараллеливать ли решаемую задачу или нет, а также интерфейс будет содержать окно вывода того, что отрисовалось (канвас).

Для работы будут разработаны следующие процедуры:

- процедура, реализующая построение отрезка по алгоритму Брезенехема;
- процедура построения спектров без использования многопоточности, входные данные - длина отрезка в спектре, выходные - множество отрезков на канвасе;
- процедура построения спектров с использованием многопоточности, входные данные - длина отрезка в спектре, количество потоков, выходные - множество отрезков на канвасе;
- процедуры замера времени построения спектров отрезков по Брезенехему с использованием многопоточности и без, входные данные - длина отрезка в спектре (если время замеряется для разного количества потоков), количество потоков (если замеряется время для разной длины отрезков в спектре), выходные - результаты замеров времени;
- процедура для построения графиков по полученным временным замерам, входные данные - замеры времени, выходные - график.

2.3 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма для построения отрезка по алгоритму Брезенхема. На рисунках 2.2 схема алгоритма построения спектра отрезков по Брезенехему без использования многопоточности, а на 2.3 - с использованием многопоточности. Также на рисунке 2.4 представлена схема алгоритма построения части отрезков Брезенхема для параллельного вычисления.

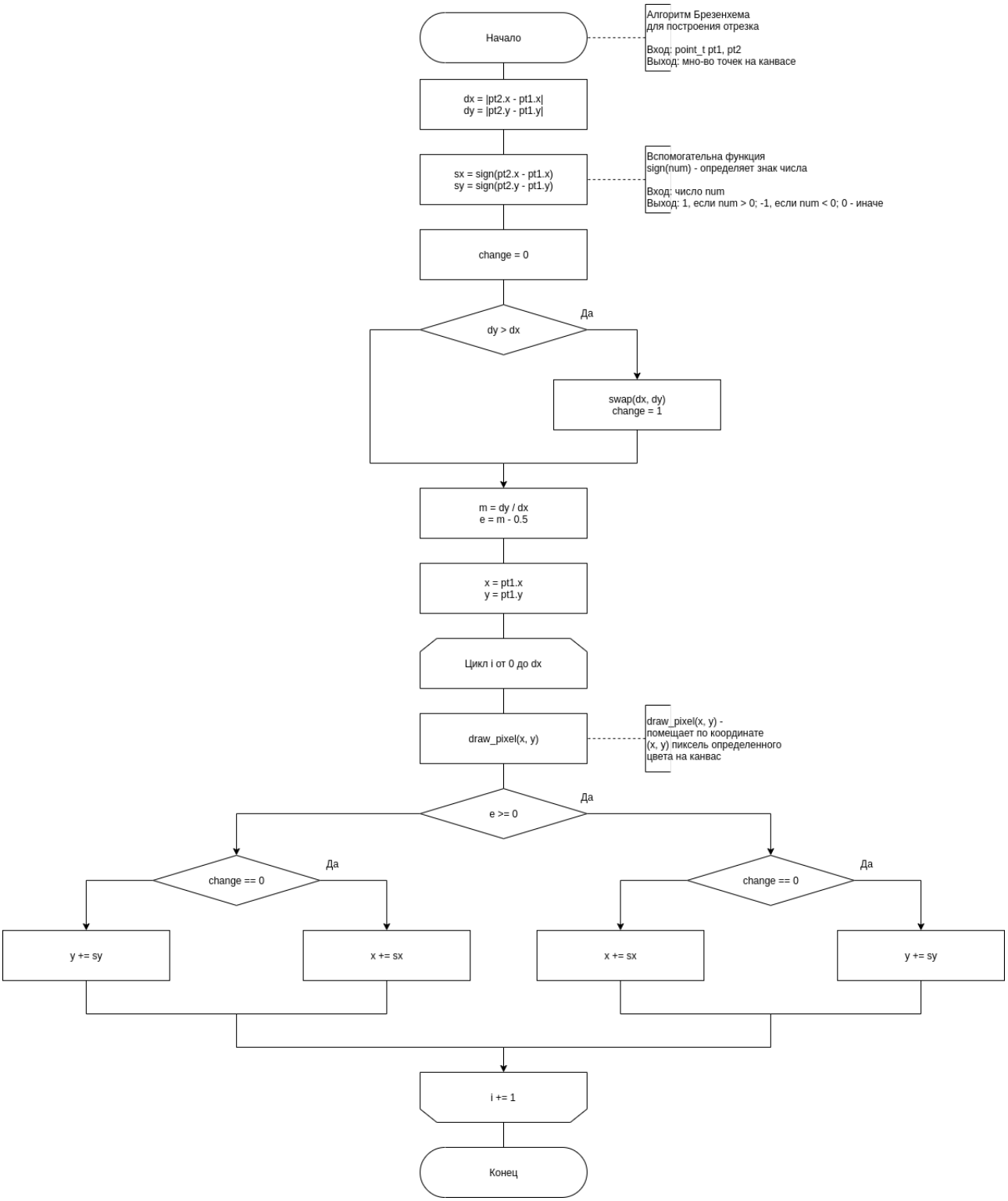

Рисунок 2.1 – Схема алгоритма Брезенхема для построения отрезка

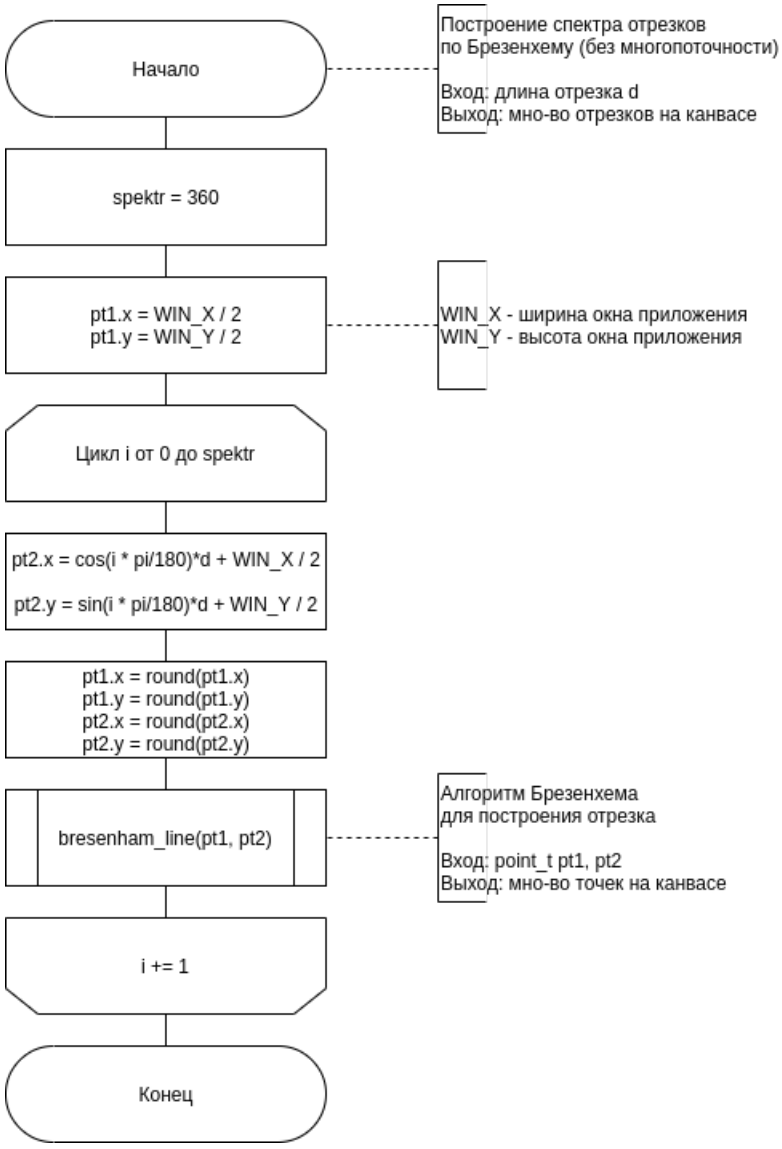

Рисунок 2.2 – Схема построения спектра отрезков по Брезенехему (без многопоточности)

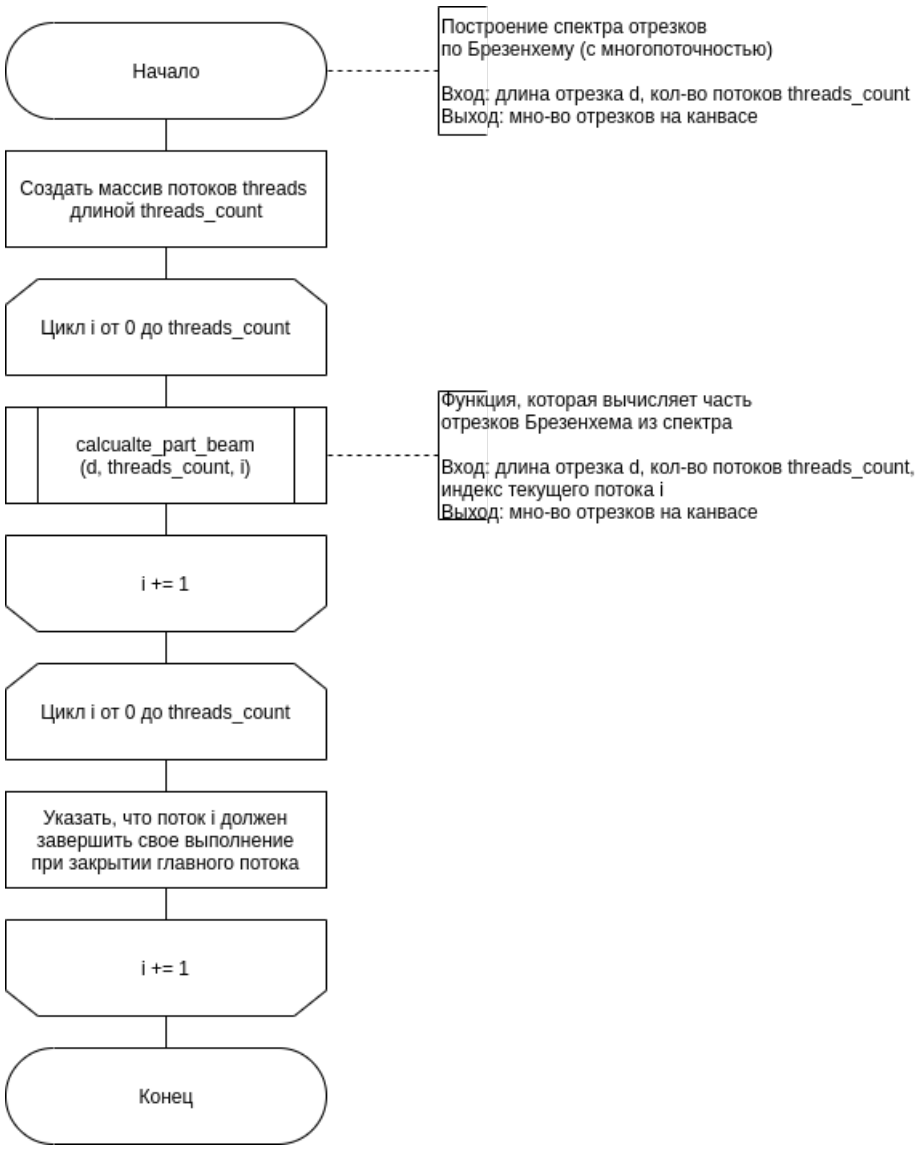

Рисунок 2.3 – Схема построения спектра отрезков по Брезенехему (с многопоточностью)

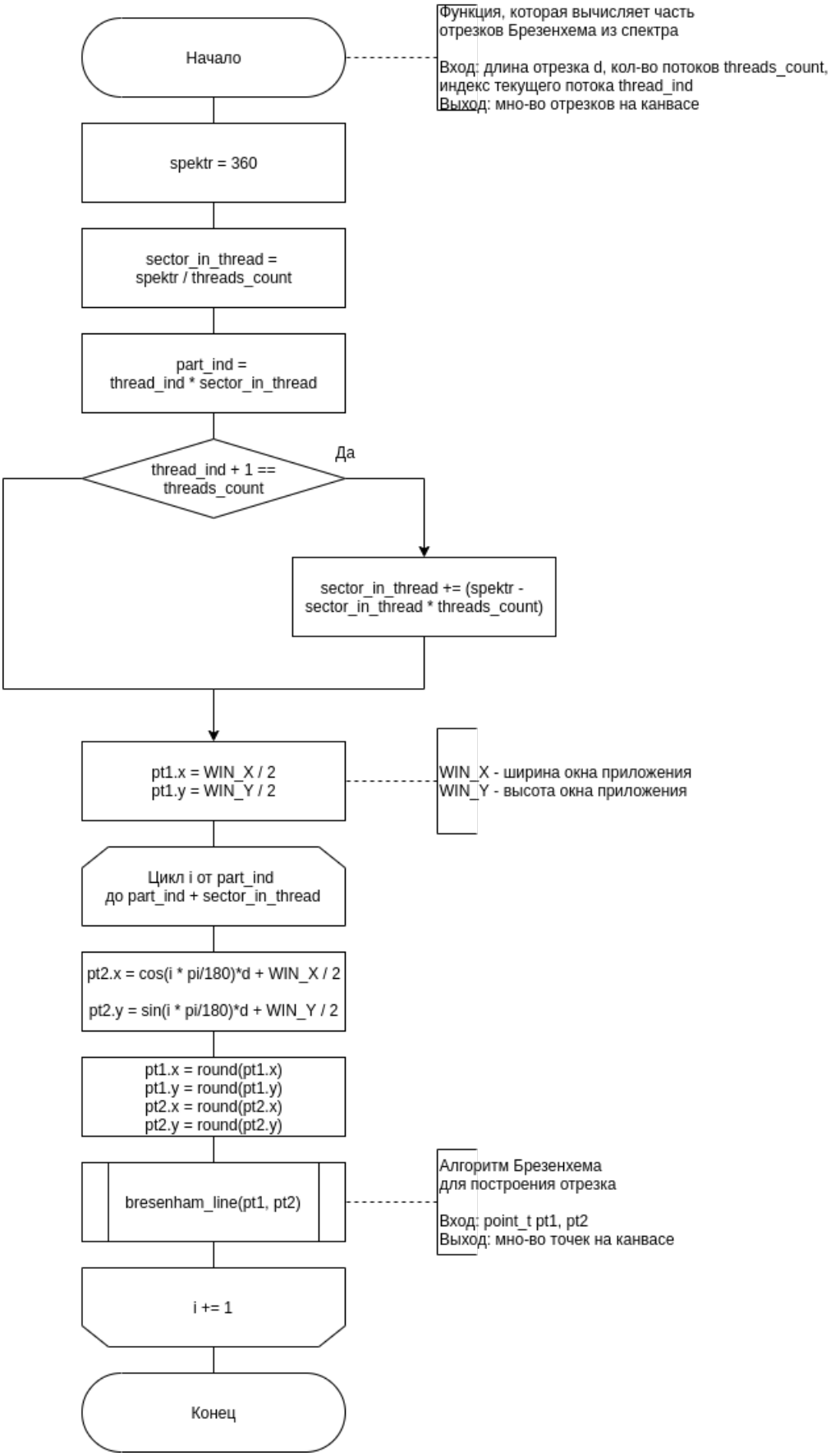

Рисунок 2.4 – Схема построения части отрезков спектра для текущего потока

2.4 Классы эквивалентности при тестировании

Для тестирования выделены классы эквивалентности, представленные ниже.

1. Не введена длина отрезка в спектре или длина меньше или равно нулю.
2. Не введено количество потоков или их количество меньше или равно нулю.
3. Корректный ввод всех параметров.

2.5 Вывод

В данном разделе были построены схемы алгоритмов умножения матриц рассматриваемых в лабораторной работе, были описаны классы эквивалентности для тестирования, структура программы.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги алгоритма Брезенехема для построенияотрезка отрезка, а также листинги алгоритмов построения спектра таких отрезков с использованием многопоточности и без.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $C++$ [3]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также реализовать принципы многопоточного алгоритма. Все эти инструменты присутствуют в выбранном языке программирования.

Функции построения графиков были реализованы с использованием языка программирования *Python* [4].

Время замерено с помощью *std::chrono::system_clock::now(...)* - функции из библиотеки *chrono* [5].

3.2 Листинги кода

В листинге 3.1 представлены реализация алгоритма Брезенехема для построения отрезка, а в листингах 3.2-3.3 - алгоритмы для построения спектра отрезков по Брезенехему с многопоточностью и без нее соответственно.

Листинг 3.1 – Алгоритм Брезенехема для построения отрезка

```
1  void breshenham_float(const request_t &request, const point_t &pt1,
       const point_t &pt2){
2      int dx = abs(pt2.x - pt1.x);
3      int dy = abs(pt2.y - pt1.y);
4      int sx = sign(pt2.x - pt1.x);
5      int sy = sign(pt2.y - pt1.y);
6      int change = 0;
7
8      if (dy > dx){
9          SWAP(int, dx, dy);
10         change = 1;
11     }
12     double m = (double)dy / (double)dx;
13     double e = m - 0.5;
14     int x = pt1.x;
15     int y = pt1.y;
16
17     for (int i = 0; i < dx; i++){
18         draw_pixel(request, x, y);
19         if (e >= 0){
20             if (change == 0){
21                 y += sy;
22             }
23             else{
24                 x += sx;
25             }
26             e -= 1;
27         }
28         if (e < 0){
29             if (change == 0){
30                 x += sx;
31             }
32             else{
33                 y += sy;
34             }
35             e += m;
36         }
37     }
38 }
```

Листинг 3.2 – Алгоритм построения спектра отрезков по Брезенехему (с многопоточностью)

```
1  void calculate_part_beam(const request_t &request, const
       beam_stgs_t &settings, int thread_ind){
2      int spektr = 360;
3      int sector_in_thread = spektr / settings.threads_count;
4      int part_ind = thread_ind * sector_in_thread;
5
6      if (thread_ind + 1 == settings.threads_count){
7          sector_in_thread += (spektr - sector_in_thread *
               settings.threads_count);
8      }
9      point_t pt1, pt2;
10     pt1.x = WIN_X / 2;
11     pt1.y = WIN_Y / 2;
12
13     for (int i = part_ind; i < part_ind + sector_in_thread; i += 1){
14         pt2.x = cos(i * PI / 180) * settings.d + WIN_X / 2;
15         pt2.y = sin(i * PI / 180) * settings.d + WIN_Y / 2;
16
17         // Round for Bresenham float
18         pt1.x = round(pt1.x);
19         pt1.y = round(pt1.y);
20         pt2.x = round(pt2.x);
21         pt2.y = round(pt2.y);
22         bresenham_line(request, pt1, pt2);
23     }
24 }
25
26 void calculate_beam_parallel(const request_t &request, const
       beam_stgs_t &settings){
27     std::vector<std::thread> threads(settings.threads_count);
28
29     for (int i = 0; i < settings.threads_count; i++){
30         threads[i] = std::thread(calculate_part_beam, request,
               settings, i);
31     }
32     for (int i = 0; i < settings.threads_count; i++){
33         threads[i].join();
34     }
35 }
```

Листинг 3.3 – Алгоритм построения спектра отрезков по Брезенехему (без многопоточности)

```
1  void calculate_beam_no_parallel(const request_t &request, const
       beam_stgs_t &settings){
2      int spektr = 360;
3
4      point_t pt1, pt2;
5      pt1.x = WIN_X / 2;
6      pt1.y = WIN_Y / 2;
7
8      for (int i = 0; i < spektr; i += 1){
9          pt2.x = cos(i * PI / 180) * settings.d + WIN_X / 2;
10         pt2.y = sin(i * PI / 180) * settings.d + WIN_Y / 2;
11
12         // Round for Bresenham float
13         pt1.x = round(pt1.x);
14         pt1.y = round(pt1.y);
15         pt2.x = round(pt2.x);
16         pt2.y = round(pt2.y);
17
18         bresenham_line(request, pt1, pt2);
19     }
20 }
```

3.3 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.cpp* - файл, содержащий функцию, вызывающую интерфейс программы;
- *mainwindow.h* и *mainwindow.cpp* - файлы, содержащие код всех методов, реализующих интерфейс программы и взаимодействующие с ним;
- *building.h* и *building.cpp* - файлы, содержащие код алгоритма Брезенехема для построения отрезка, а также алгоритмов построения спектра отрезков с использованием многопоточности и без нее, а также функции замера времени;

- *graph_build.py* - файл, содержащий функции построения графиков для замеров по времени.

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций программы. Тесты *для всех функций* пройдены успешно.

Таблица 3.1 – Функциональные тесты

Длина отрезка	Кол-во потоков	Ожидаемый результат
-5	4	Сообщение об ошибке
350	0	Сообщение об ошибке
350	4	Вывод спектра отрезков на канвас
10	32	Вывод спектра отрезков на канвас
228	1	Вывод спектра отрезков на канвас

3.5 Вывод

Были представлены листинги всех алгоритмов построения спектра отрезков по Брезенехему с распараллеливанием и без него, а также сам алгоритм Брезенехема для генерации отрезка. Также в данном разделе была приведена информации о выбранных средствах для разработки алгоритмов и сведения о модулях программы, проведено функциональное тестирование.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программа, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Ubuntu 20.04.3 [6] Linux [7] x86_64;
- память: 8 GiB;
- процессор: Intel® Core™ i5-7300HQ CPU @ 2.50GHz [8];
- 4 физических ядра, 4 логических ядра [8].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

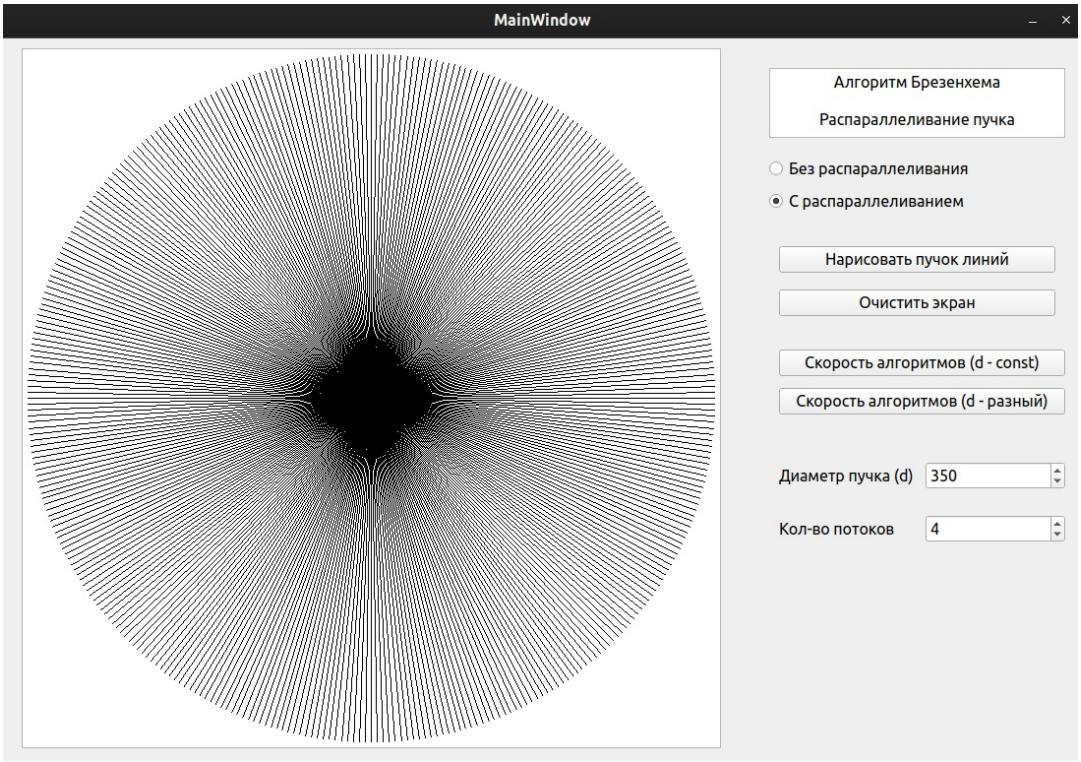

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени *std::chrono::system_clock::now(...)* из библиотеки *chrono* на C++. Функция возвращает процессорное время типа float в секундах.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Замеры проводились для разной длины отрезка в спектре, а также для разного количества потоков по 300 прогонов, для получения более точного значения времени.

Результаты замеров приведены в таблицах 4.1-4.3 (время в с).

Таблица 4.1 – Результаты замеров времени (разное количество потоков, длина отрезка в спектре - 500)

Кол-во потоков	Результат
1	0.000581
2	0.000284
4	0.000245
8	0.00031
16	0.000413
24	0.000545
32	0.000733

Таблица 4.2 – Результаты замеров времени (без распараллеливания, длина отрезка в спектре - 500)

Номер	Результат
1	0.000902

Таблица 4.3 – Результаты замеров времени
(длина отрезка в спектре - разная)

Длина отрезка	4 потока	Без многопоточности
1000	0.0006	0.001557
2000	0.001058	0.003055
3000	0.001515	0.004551
4000	0.002919	0.006259
5000	0.002382	0.007511
6000	0.002842	0.009324
7000	0.003277	0.010391
8000	0.003699	0.012398
9000	0.004686	0.013251
10000	0.004546	0.014771

Также на рисунках 4.2–4.3 приведены графические результаты замеров.

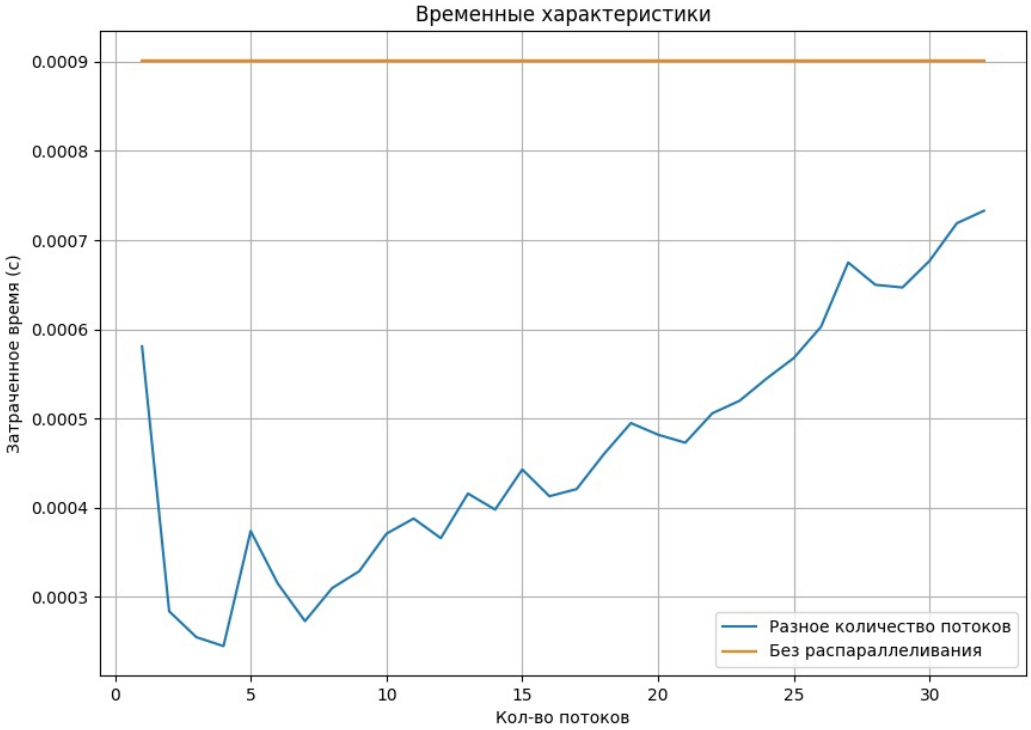

Рисунок 4.2 – Сравнение по времени алгоритмов построения спектра отрезков по Брезенехему – разное количество потоков и без распараллеливания

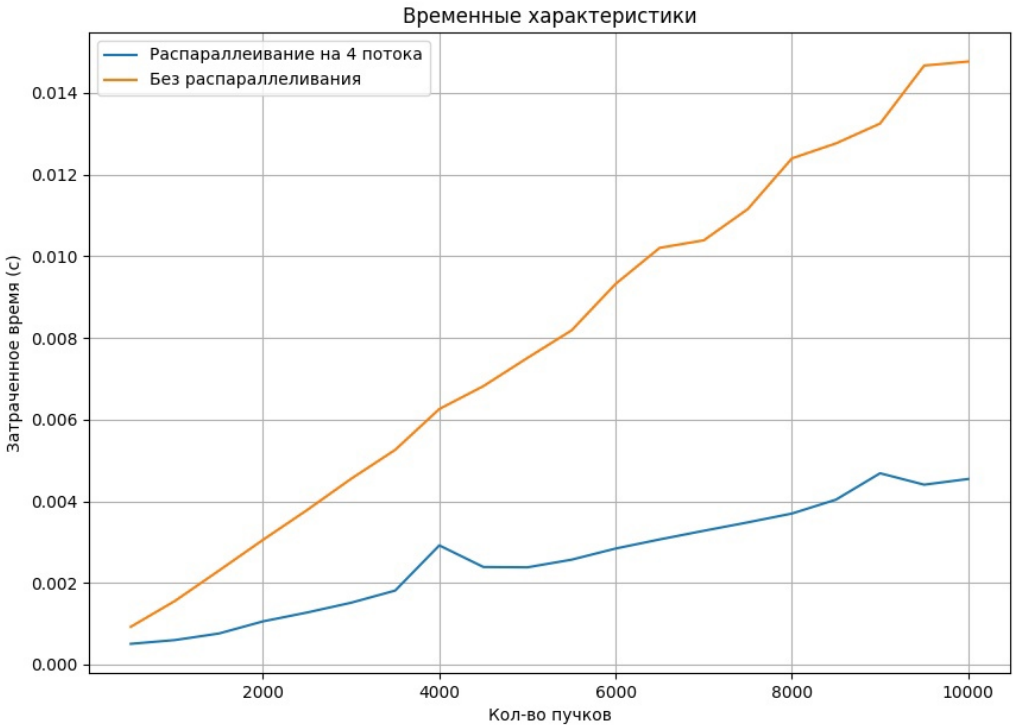

Рисунок 4.3 – Сравнение по времени алгоритмов построения спектра отрезков по Брезенехему – разная длина отрезка в спектре

4.4 Вывод

В результате эксперимента было получено, что при использовании 4 потоков, многопоточная реализация алгоритма построения спектра отрезков по Брезенехему лучше реализации без многопоточности в 3.7 раза на длине отрезка в спектре, равной 500. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводилось тестирование ПО, имеется всего 4 логических ядра, а следовательно максимальное количество потоков, которого можно добиться, равно четырем. Также важную роль играет то, что работа распределяется примерно равно между всеми потоками, остатки работы отдаются крайнему потоку. При 4 потоках остатка не будет. Именно поэтому лучшие результаты достигаются именно на 4 потоках, даже не смотря на ресурсы, которые дополнительно затрачиваются на содержание потоков. В итоге, можно сказать, что при таких данных следует использовать многопоточную реализацию алгоритма.

Также при проведении эксперимента было выявлено, что при увеличении длины отрезка в спектре, много поточная реализация выдает лучшие результаты. Так, при длине отрезка, равно 3000 многопоточная реализация лучше реализации без многопоточности в 3 раза, а уже на длине отрезка, равной 10000 - в 3.3 раза. Следовательно, стоит использовать многопоточную реализацию при большой длине отрезка в спектре (свыше 5 тысяч).

Стоит также упомянуть, что при реализации алгоритма построения спектра отрезков пришлось воспользоваться таким понятием, как mutex. Дело в том, что при многопоточной обработке может возникнуть ситуация, когда несколько потоков в один момент времени в разные места пытаются поместить на канвас пиксель. В итоге, происходит неопределенное поведение – часть пискселей помещается, а часть – пропадает. Из-за этого была реализована блокировка части кода, которая отвечает за помещение пикселя на канвас с помощью mutex. Из-за этого эта работа, по сути, происходит в однопоточном режиме, а все вычисления параллельно – потоки ждут своей очереди, чтобы поместить пиксель на канвас.

Заключение

В результате исследования было определено, что на данном ноутбуке лучшие результаты достигаются на 4 потоках, так как имеется всего 4 потока из-за наличия всего 4 логических ядер, а также из-за особенности реализации алгоритма, что остаток, который не удалось распределить между всеми потоками, отдается последнему потоку (что не происходит при 4 потоках). Преимущество во времени работы при использовании многопоточности на 4 потоках над алгоритмом, который реализован без использования многопоточности, велико - примерно в 3.7 раза лучше. Также важно, что при увеличении количества потоков растет время выполнения программы из-за того, что также время уходит на поддержку каждого потока, поэтому при большом количестве потоков (около 64) их использованием станет нецелесообразным. При этом на 4 потоках при увеличении длины отрезка в спектре видно, что многопоточность на длине в 3000 лучше реализации без многопоточности в 3 раза, а при 10 тысячах – в 3.3 раза, то есть при большой длине отрезка стоит использовать многпоточную реализацию.

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены принципы многопотчности и реализованы алгоритмы построения спектров по алгоритму Брезенехема с использованием многопоточности и без нее;
- был реализован алгоритм Брезенехема для генерации отрезка;
- проведен сравнительный анализ по времени алгоритмов с многопоточностью на разном количестве потоков и без многопоточности;
- проведен сравнительный анализ по времени алгоритмов с многопоточностью и без нее на разной длине отрезка в спектре;
- подготовлен отчет о лабораторной работе.

Список литературы

[1] Что такое многопоточность [Электронный ресурс]. Режим доступа: https://ru.coursera.org/lecture/ios-multithreading/chto-takoie-mnoghopotochnost-4MMgN (дата обращения: 23.10.2021).

[2] Алгоритм Брезенхема [Электронный ресурс]. Режим доступа: http://www.mari-el.ru/mmlab/home/kg/Lection5/3.html (дата обращения: 23.10.2021).

[3] Программирование на C/C++ [Электронный ресурс]. Режим доступа: http://www.c-cpp.ru/ (дата обращения: 23.10.2021).

[4] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 23.10.2021).

[5] Date and time utilities [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono (дата обращения: 23.10.2021).

[6] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: https://releases.ubuntu.com/20.04/ (дата обращения: 15.10.2021).

[7] Linux [Электронный ресурс]. Режим доступа: https://www.linux.org/forums/#linux-tutorials.122 (дата обращения: 15.10.2021).

[8] Процессор Intel® Core™ i5-7300HQ [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/97456/intel-core-i5-7300hq-processor-6m-cache-up-to-3-50-ghz.html (дата обращения: 04.10.2021).