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 Параллельные вычисления

Студент Артюхин Н.П.

Группа ИУ7-51Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Целью данной работы является изучение организации параллельных вычислений на базе алгоритма построения спектра отрезков по Брезенхему.

Многопоточность — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой [1]. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Смысл многопоточности — квазимногозадачность на уровне одного исполняемого процесса. Значит, все потоки процесса помимо общего адресного пространства имеют и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

Многопоточность стала популярной с конца 1990-х годов, так как усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились.

Далее будут приведены достоинства и недостатки многопоточности.

Достоинства:

- облегчение программы посредством использования общего адресного пространства;
- меньшие затраты на создание потока в сравнении с процессами;
- повышение производительности процесса за счёт распараллеливания процессорных вычислений;

- если поток часто теряет кэш, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов;
- с программной точки зрения аппаратная поддержка многопоточности является достаточно трудоемкой для программного обеспечения;
- проблема планирования потоков;
- специфика использования.

Каждый поток в процессе — это задача, которую должен выполнить процессор. Большинство современных процессоров могут выполнять одновременно две задачи на одном ядре, создавая дополнительное виртуальное ядро.

Эти процессоры называются многоядерными процессорами. Таким образом, четырехъядерный процессор имеет 8 ядер: 4 физических и 4 виртуальных. Каждое ядро может одновременно выполнять только один поток.

Для достижения поставленной цели требуется решить следующие задачи:

1) изучение основ параллельных вычислений (многопоточности);
2) изучение алгоритма Брезенхема для построения отрезков;
3) разработка параллельной и последовательной версии алгоритма построения спектра отрезков по Брезенхему (то есть с использованием многопоточности и без нее);
4) реализация параллельной и последовательной версий алгоритма построения спектра отрезков по Брезенхему;
5) сравнительный анализ времени выполнения реализаций последовательной и параллельной (при различном количестве потоков) версий алгоритма построения спектра отрезков по Брезенхему.

6) сравнительный анализ времени выполнения реализаций с использованием многопоточности и без при разной длине отрезка в спектре;

1 Аналитическая часть

В данном разделе будет представлено описание алгоритма Брезенхема для построения отрезков.

1.1 Алгоритм Брезенхема построения отрезка

Алгоритм Брезенхема — это алгоритм, определяющий, какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками. Это один из старейших алгоритмов в машинной графике — он был разработан Джеком Е. Брезенхемом в компании IBM в 1962 году [2].

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 [3].

Графическая интерпретация алгоритма Брезенхема представлена на рисунке 1.1.

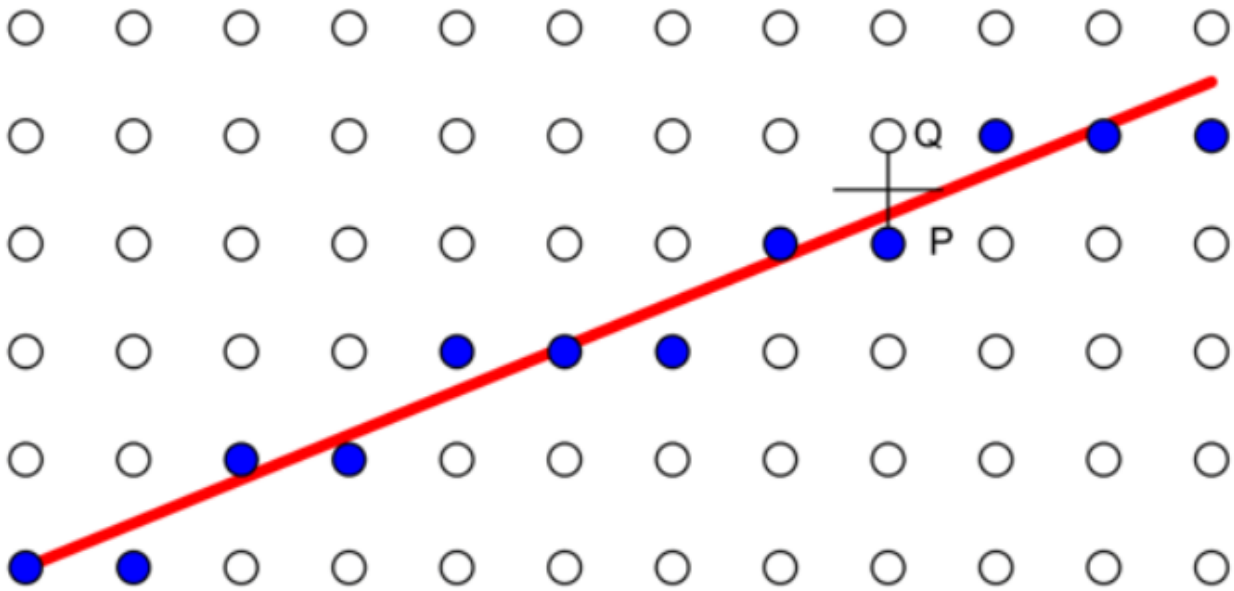

Рисунок 1.1 – Графическая интерпретация алгоритма Брезенхема

Вывод

В данном разделе были рассмотрены основные идеи, лежащие в основе алгоритма Брезенхема построения отрезка.

2 Конструкторская часть

В данном разделе будут представлены схемы алгоритмов построения спектра отрезков по Брезенхему с распараллеливанием и без него.

2.1 Алгоритм Брезенхема построения отрезка

На рисунке 2.1 приведена схема алгоритма Брезенхема построения отрезка.

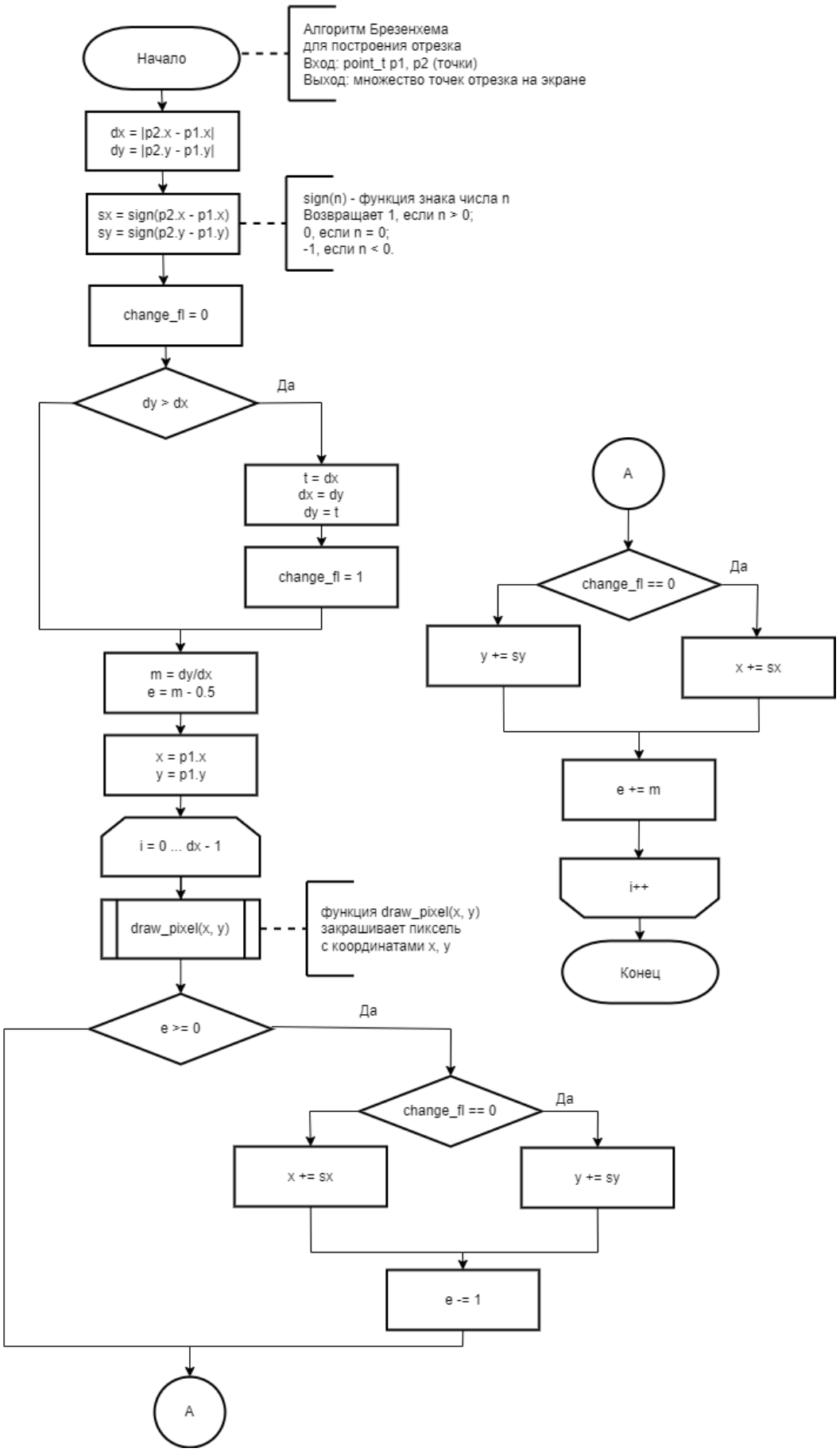

Рисунок 2.1 – Схема алгоритма Брезенхема построения отрезка

2.2 Алгоритм построения спектра отрезков по Брезенхему (последовательный)

На рисунке 2.2 приведена схема алгоритма построения спектра отрезков по Брезенхему без многопоточности (последовательный).

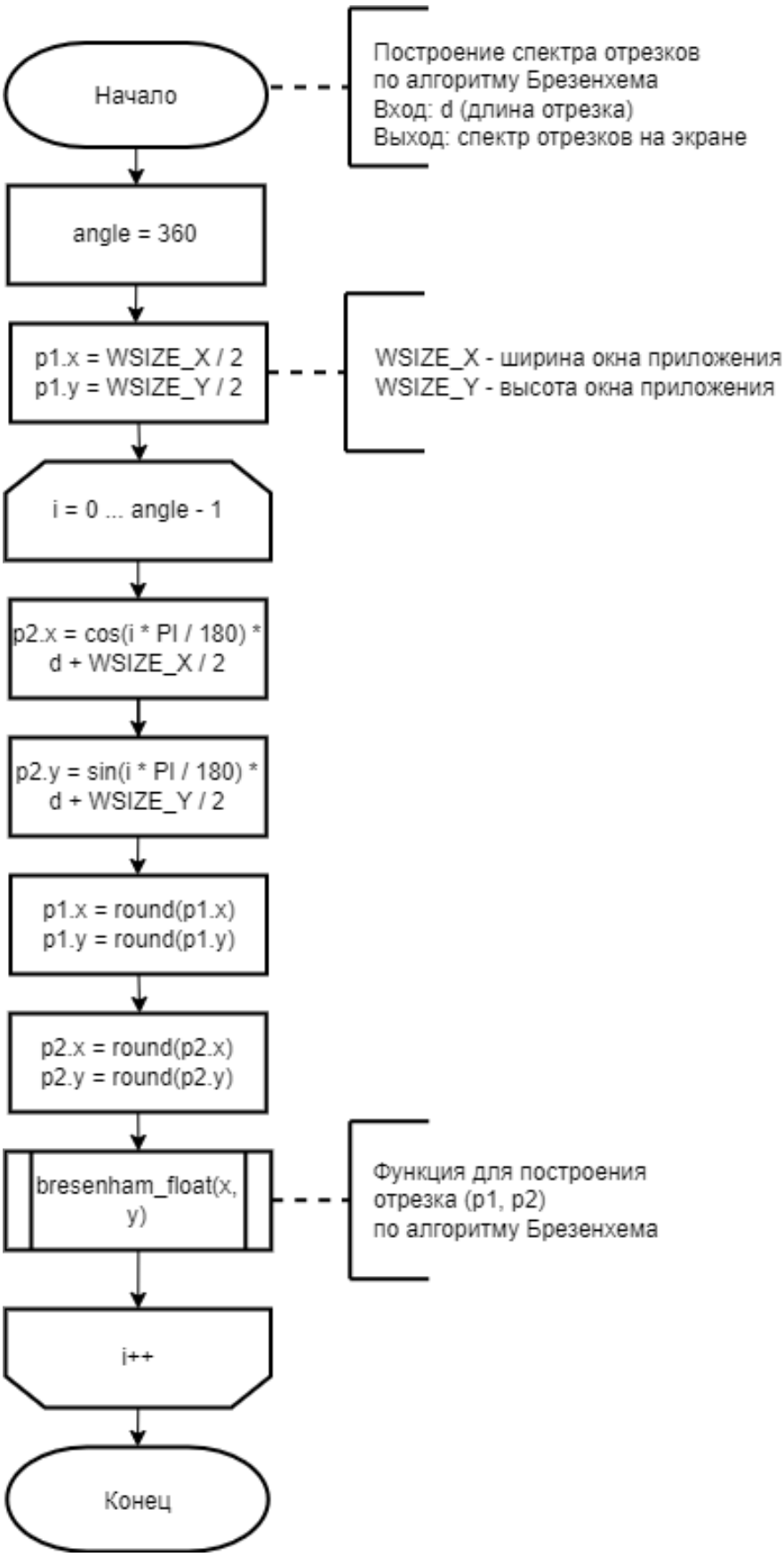

Рисунок 2.2 – Схема алгоритма построения спектра отрезков по Брезенхему без многопоточности

2.3 Алгоритм построения спектра отрезков по Брезенхему (параллельный)

На рисунках 2.3 и 2.4 приведена схема алгоритма построения спектра отрезков по Брезенхему с многопоточностью (параллельный).

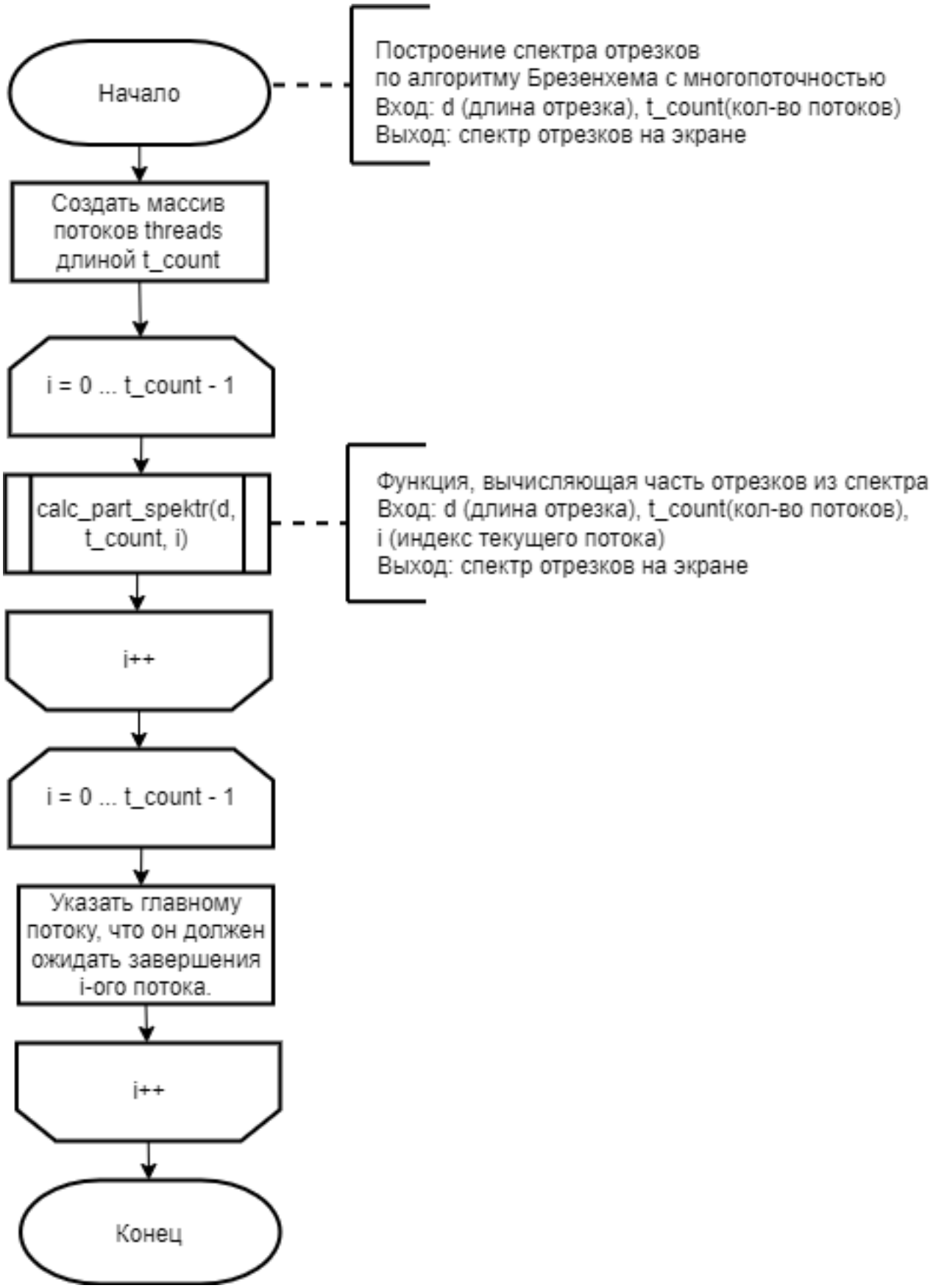

Рисунок 2.3 – Схема алгоритма построения спектра отрезков по Брезенхему с многопоточностью

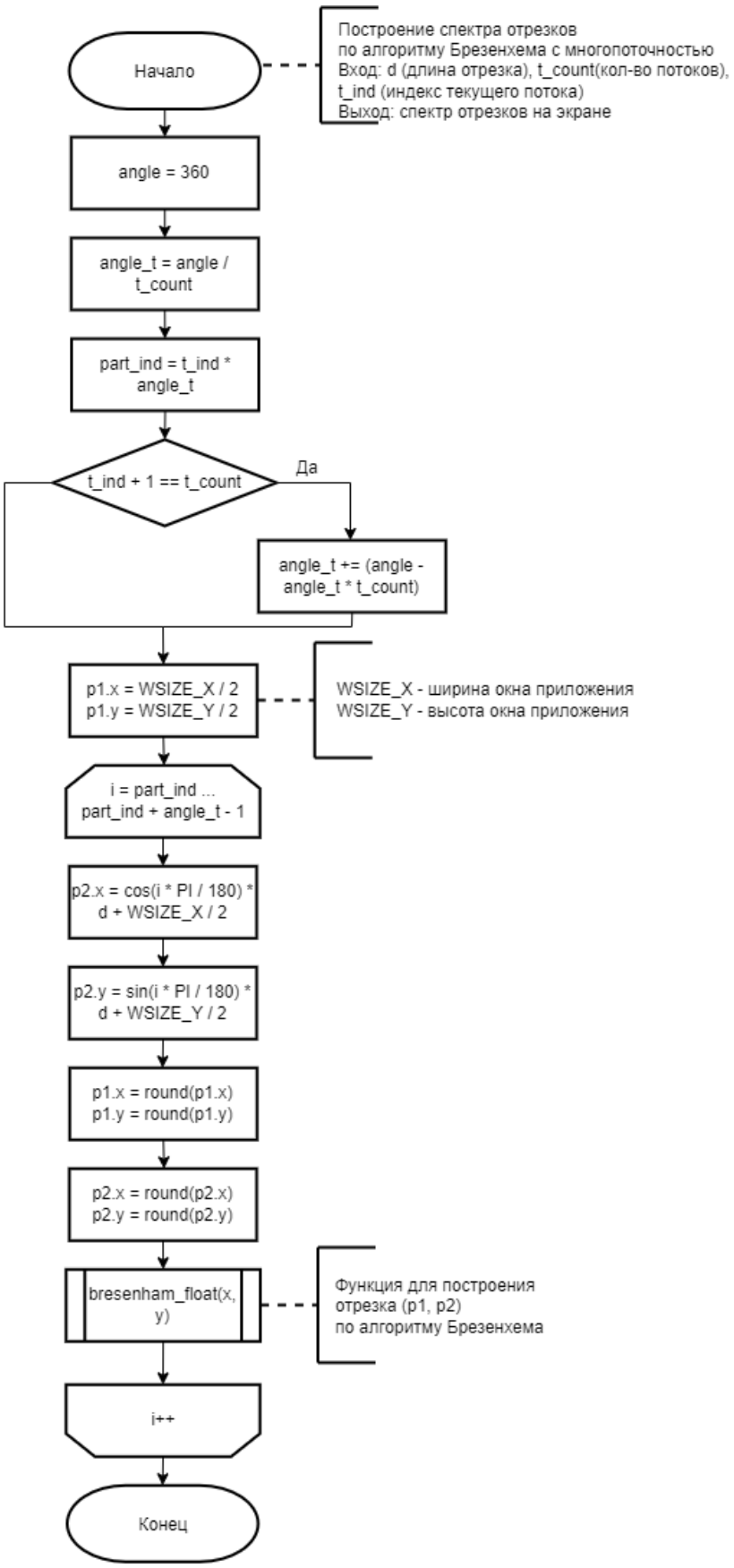

Рисунок 2.4 – Схема алгоритма построения спектра отрезков по Брезенхему с многопоточностью

Вывод

В данном разделе были разработаны схемы алгоритмов построения спектра отрезков по Брезенхему с распараллеливанием и без него.

3 Технологическая часть

В данном разделе будут представлены требования к программному обеспечению, средства реализации, листинги кода и тесты.

3.1 Требования к программному обеспечению

Пользователю предоставляется графический интерфейс для ввода данных.

Вход: длина отрезка (целое число), режим построения отрезка (с распараллеливанием или без него), количество потоков (в случае выбора режима построения отрезка с распараллеливанием), действие (построение спектра отрезков, очистка экрана, замер процессорного времени работы реализаций алгоритмов построения спектра отрезков);

Выход: спектр отрезков на эране/результаты замеров процессорного времени в консоли.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C++ [4]. Данный язык программирования позволяет замерить время выполнения работы реализации алгоритма, а также организовать многопоточность для параллельных вычислений.

Время выполнения реализаций алгоритма построения спектра отрезков было замерено с помощью функции *std::chrono::system_clock::now(...)* из библиотеки *chrono* [5].

В качестве среды разработки был выбран QtCreator [6].

Графики были построены с использованием языка программирования Python [7].

3.3 Реализация алгоритмов

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

Листинг 3.1 – Функция алгоритма Брезенехема для построения отрезка

```
void bresenham_float(const request_t &r, const point_t &p1, const
    point_t &p2)
{
    int dx = abs(p2.x - p1.x);
    int dy = abs(p2.y - p1.y);

    int sx = sign(p2.x - p1.x);
    int sy = sign(p2.y - p1.y);

    int change_fl = 0;

    if (dy > dx)
    {
        swap(dx, dy);
        change_fl = 1;
    }

    double m = (double)dy / (double)dx;
    double e = m - 0.5;

    int x = p1.x;
    int y = p1.y;

    std::mutex mut;

    for (int i = 0; i < dx; i++)
    {
        if (r.is_draw)
        {
            mut.lock();
            draw_pixel(r, x, y);
            mut.unlock();
```

```
32          }
33          if (e >= 0)
34          {
35              if (change_fl == 0)
36              {
37                  y += sy;
38              }
39              else
40              {
41                  x += sx;
42              }
43
44              e -= 1;
45          }
46          if (change_fl == 0)
47          {
48               x += sx;
49          }
50          else
51          {
52              y += sy;
53          }
54
55          e += m;
56      }
57  }
```

Листинг 3.2 – Функция алгоритма построения спектра отрезков по Брезенехему (без многопоточности)

```
1  void calculate_spek_no_parallel(const request_t &r, const spektr_t
       &params)
2  {
3      int spektr = 360;
4
5      point_t p1, p2;
6      p1.x = WSIZE_X / 2;
7      p1.y = WSIZE_Y / 2;
8
9      for (int i = 0; i < spektr; i += 1)
10     {
11         p2.x = cos(i * PI / 180) * params.d + WSIZE_X / 2;
```

```
12          p2.y = sin(i * PI / 180) * params.d + WSIZE_Y / 2;
13
14          // rounding for Bresenham float algorithm
15          p1.x = round(p1.x);
16          p1.y = round(p1.y);
17          p2.x = round(p2.x);
18          p2.y = round(p2.y);
19
20
21          calculate_line(r, p1, p2);
22      }
23  }
```

Листинг 3.3 – Функция алгоритма построения спектра отрезков по Брезенехему (с многопоточностью)

```
1  void calculate_part_spek(const request_t &r, const spektr_t &
       params, int t_ind)
2  {
3      int spektr = 360;
4      int sector_in_t = spektr / params.t_count;
5      int part_ind = t_ind * sector_in_t;
6
7      if (t_ind + 1 == params.t_count)
8      {
9          sector_in_t += (spektr - sector_in_t * params.t_count);
10     }
11
12     point_t p1, p2;
13     p1.x = WSIZE_X / 2;
14     p1.y = WSIZE_Y / 2;
15
16     for (int i = part_ind; i < part_ind + sector_in_t; i += 1)
17     {
18
19         p2.x = cos(i * PI / 180) * params.d + WSIZE_X / 2;
20         p2.y = sin(i * PI / 180) * params.d + WSIZE_Y / 2;
21
22         // rounding for Bresenham float algorithm
23         p1.x = round(p1.x);
24         p1.y = round(p1.y);
25         p2.x = round(p2.x);
```

```
26          p2.y = round(p2.y);
27
28          calculate_line(r, pt1, pt2);
29      }
30  }
31
32
33  void calculate_spek_parallel(const request_t &r, const spektr_t &
        params)
34  {
35      std::vector<std::thread> thrs(params.t_count);
36
37      for (int i = 0; i < params.t_count; i++)
38      {
39          thrs[i] = std::thread(calculate_part_spek, r, params, i);
40      }
41
42      for (int i = 0; i < params.t_count; i++)
43      {
44          thrs[i].join();
45      }
46  }
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов построения спектра отрезков. Все тесты были пройдены успешно.

Таблица 3.1 – Тесты

№	Длина отрезка	Кол-во потоков	Ожидаемый результат
1	0	0	Сообщение об ошибке
2	50	-1	Сообщение об ошибке
3	-100	2	Сообщение об ошибке
4	200	1	Спектр с отрезками длины 200
5	300	8	Спектр с отрезками длины 300

Вывод

В данном разделе были представлены требования к программному обеспечению и средства реализации, реализованы и протестированы алгоритмы построения спектра отрезков по Брезенхему с распараллеливанием и без него.

4 Исследовательская часть

В текущем разделе будут представлены примеры работы разработанного программного обеспечения, постановка эксперимента и сравнительный анализ реализованных алгоритмов.

4.1 Пример работы программного обеспечения

На рисунке 4.1 представлен результат работы программы при выбранном режиме построения — с распараллеливанием, длине отрезка — 300, количестве потоков — 8. В результате выполнения программы на экране (канвас) появился спектр отрезков длины 300.

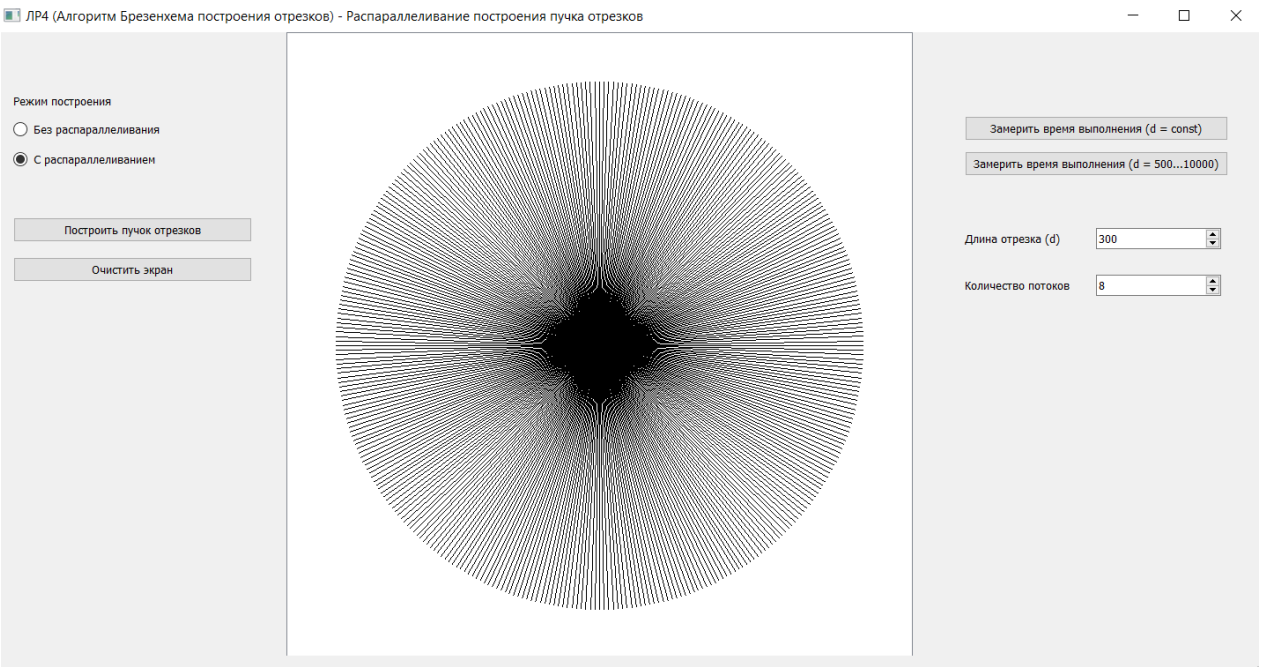

Рисунок 4.1 – Пример работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнись замеры времени:

- операционная система: Windows 10 [8];
- оперативная память: 16 Гб;
- процессор: Intel® Core™ i5 10300H 2.5 ГГц;
- 4 физических ядра, 4 логических ядра.

Во время замеров времени выполнения реализаций алгоритмов ноутбук был включен в сеть питания и нагружен только встроенными приложениями окружения и системой тестирования.

4.3 Время выполнения реализаций алгоритмов построения спектра отрезков

Для замера времени выполнения реализованных алгоритмов использовалась функция *system_clock::now(...)* из библиотеки *chrono*. Данная функция возвращает время в наносекундах.

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

Замеры времени проводились для разной длины отрезка в спектре (от 1000 до 10000 с шагом 1000) и для разного количества потоков (1, 2, 4, 8, 16, 32), в качестве результата бралось среднее время из 500 итераций и переводилось из наносекунд в секунды посредством деления на 1e9.

Результаты замеров времени работы реализаций алгоритмов приведены в таблицах 4.1, 4.2.

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

Количество потоков	Время выполнения (с)
без многопоточности	0.000730
1	0.000794
2	0.000457
4	0.000403
8	0.000511
16	0.000976
32	0.002083

Таким образом, реализация алгоритма построения спектра отрезков по Брезенхему работает наиболее эффективно по времени при распараллеливании на 4 потока. Последовательная реализация чуть быстрее многопоточной реализации с 1 рабочим потоком из-за дополнительных затрат по времени на создание рабочего потока.

Таблица 4.2 – Результаты замеров времени (длина отрезка в спектре от 1000 до 10000 с шагом 1000) (в секундах)

Длина отрезка	4 потока	Без многопоточности
1000	0.000544	0.001202
2000	0.000876	0.002332
3000	0.001216	0.003467
4000	0.001524	0.004576
5000	0.001854	0.005697
6000	0.002167	0.006801
7000	0.002412	0.007922
8000	0.002691	0.009031
9000	0.003144	0.010260
10000	0.003465	0.011408

На рисунке 4.2 представлено сравнение времени работы последовательной и многопоточной реализаций алгоритмов построения спектра отрезков по Брезенехему.

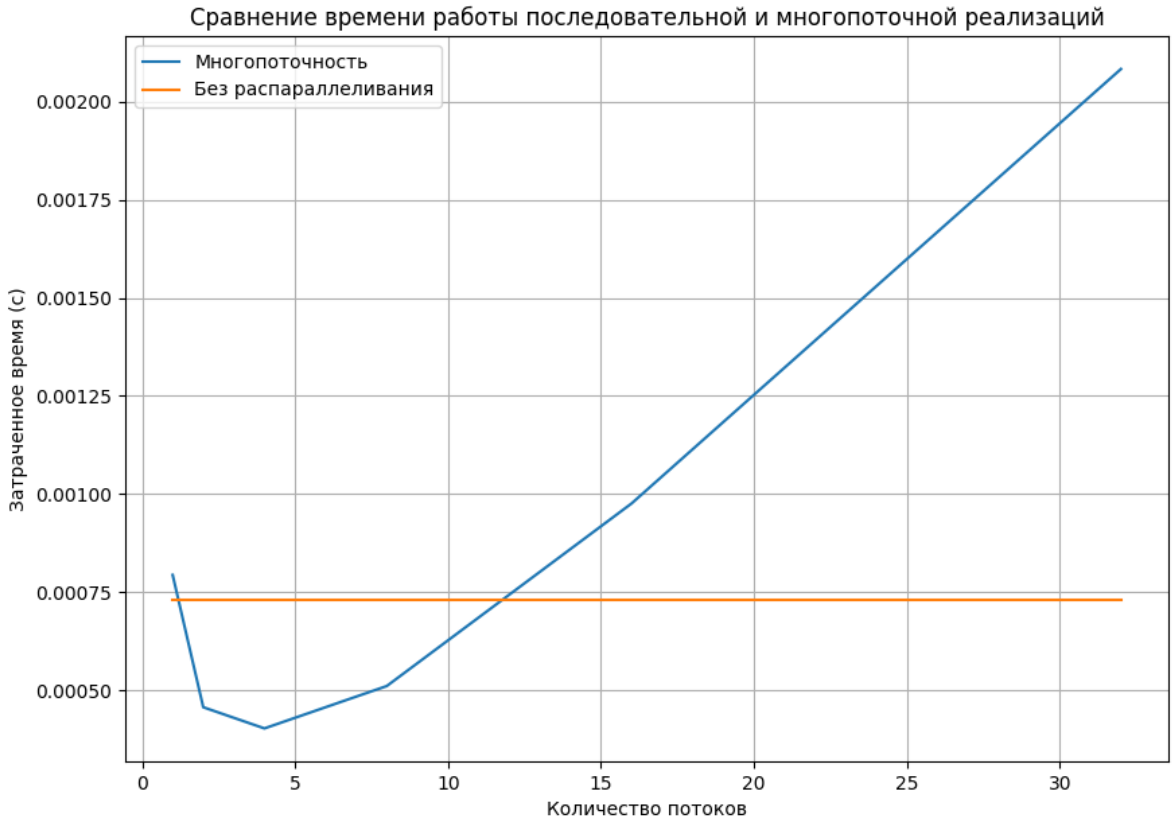

Рисунок 4.2 – Сравнение времени работы последовательной и многопоточной реализаций алгоритмов построения спектра отрезков по Брезенехему

На рисунке 4.3 представлено сравнение времени работы реализаций без многопоточности и с 4 потоками алгоритмов построения спектра отрезков по Брезенехему на разных длинах отрезков в спектре.

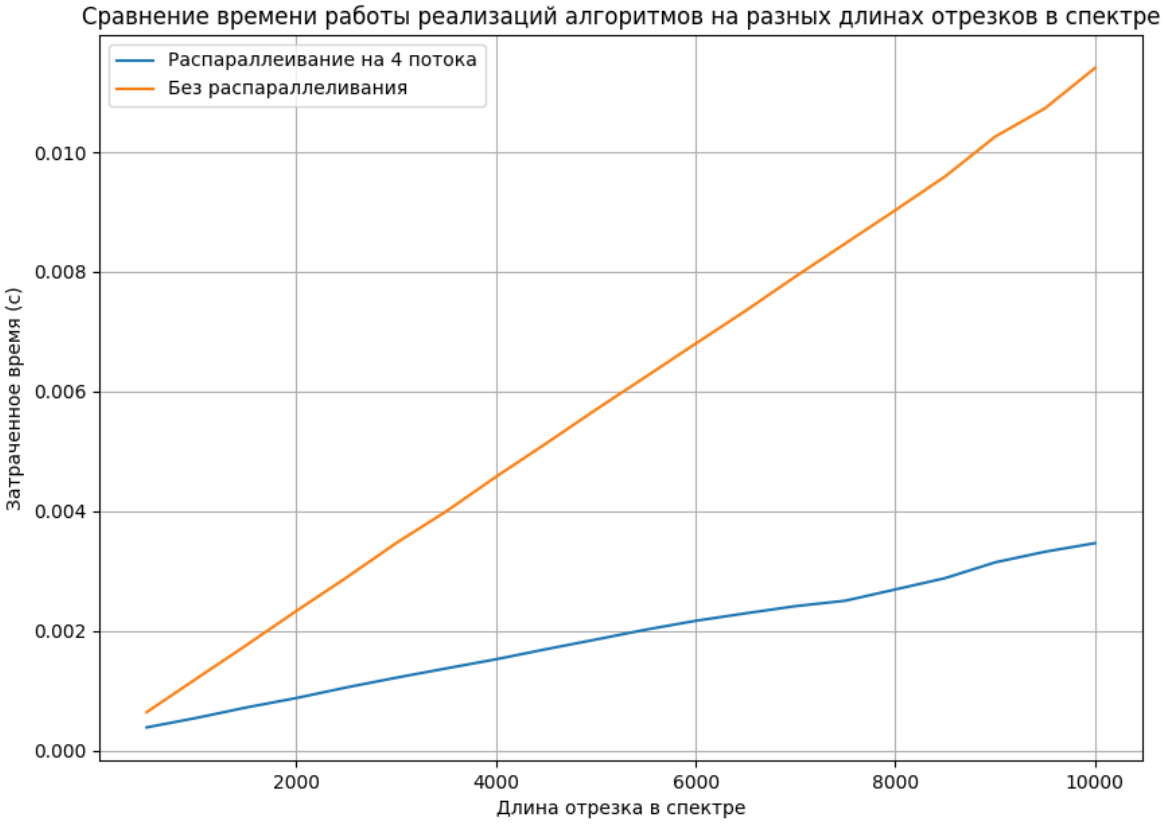

Рисунок 4.3 – Сравнение времени работы реализаций алгоритмов построения спектра отрезков по Брезенехему на разных длинах отрезков в спектре

Вывод

В результате эксперимента было получено, что при использовании 4 потоков, многопоточная реализация алгоритма построения спектра отрезков по Брезенехему лучше реализации без многопоточности почти в 2 раза на длине отрезка в спектре, равной 500. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводились замеры времени, имеется всего 4 логических ядра. Таким образом, рекомендуется применять 4 потока, многопоточная реализация с 4 потоками алгоритма построения спектра отрезков достигает наибольшей эффективности по времени.

Последовательная реализация алгоритма построения спектра отрезков по Брезенхему чуть эффективнее по времени, чем многопоточная реализация с 1 рабочим потоком из-за дополнительных затрат по времени на создание рабочего потока и диспетчеризацию.

Также при проведении эксперимента было выявлено, что при увеличении длины отрезка в спектре, преимущество многопоточной реализации (с 4 потоками) над последовательной растет. Так, при длине отрезка, равной 1000 многопоточная реализация с 4 потоками эффективней по времени реализации без многопоточности в 2.2 раза, а на длине отрезка, равной 10000 — в 3.3 раза. Следовательно, чем больше длина отрезка в спектре, тем выгоднее использовать многопоточную реализацию.

Важно отметить, что при реализации алгоритма построения спектра отрезков пришлось воспользоваться мьютексом, так как при многопоточной реализации может возникнуть ситуация, когда несколько потоков в один момент времени пытаются закрасить разные пиксели. Таким образом, происходит неопределенное поведение — часть пискселей закрашивается, а часть — пропадает. Поэтому была реализована блокировка части кода, которая отвечает за помещение пикселя на канвас с помощью мьютекса. Все вычисления выполняются параллельно, но потокам придется встать в очередь, чтобы закрасить пиксель на экране.

Заключение

В результате выполнения лабораторной работы цель достигнута: изучена организация параллельных вычислений на базе алгоритма построения спектра отрезков по Брезенхему.

В ходе выполнения данной работы были решены все задачи:

- изучены основы параллельных вычислений (многопоточности);
- изучен алгоритм Брезенхема для построения отрезков;
- разработаны параллельная и последовательная версии алгоритма построения спектра отрезков по Брезенхему (то есть с использованием многопоточности и без нее);
- реализованы параллельная и последовательная версии алгоритма построения спектра отрезков по Брезенхему;
- выполнены замеры процессорного времени работы реализаций алгоритмов сортировки (гномья сортировка, поразрядная сортировка, сортировка выбором);
- проведен сравнительный анализ времени выполнения реализаций последовательной и параллельной (при различном количестве потоков) версий алгоритма построения спектра отрезков по Брезенхему;
- проведен сравнительный анализ времени выполнения реализаций с использованием многопоточности и без при разной длине отрезка в спектре.

В результате лабораторной работы можно сделать вывод, что самой эффективной по времени является многопоточная реализация (с 4 потоками) алгоритма построения спектра отрезков по Брезенхему, так как ноутбук, на котором производились замеры времени, имеет 4 логических ядра.

Список использованных источников

[1] Многопоточность [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C (дата обращения: 28.10.2022).

[2] Растровый алгоритм Брезенхема построения линии [Электронный ресурс]. Режим доступа: http://grafika.me/node/8 (дата обращения: 29.10.2022).

[3] Алгоритм Брезенхема для рисования наклонных отрезков [Электронный ресурс]. Режим доступа: https://prog-cpp.ru/brezenham/ (дата обращения: 29.10.2022).

[4] Справочник по языку C++ [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170 (дата обращения: 29.10.2022).

[5] Date and time utilities [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono (дата обращения: 29.10.2022).

[6] Qt Creator Manual [Электронный ресурс]. Режим доступа: https://doc.qt.io/qtcreator/ (дата обращения: 29.10.2022).

[7] Лутц Марк. Изучаем Python, том 1, 5-е изд. Пер. с англ. — СПб.: ООО “Диалектика”, 2019. с. 832.

[8] Windows 10 [Электронный ресурс]. Режим доступа: https://learn.microsoft.com/ru-ru/windows/ (дата обращения: 20.09.2022).