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 дисциплине "Анализ алгоритмов"

Тема Многопоточные вычисления

Студент Михаил Коротыч

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва 2021 г.

Оглавление

Введение

Поток или поток выполнения - базовая упорядоченная последовательность инструкций, которые могут быть переданы или обработаны одним ядром процессора [1]. Обычные программы поступают на обработку в виде одного потока. Многопоточная программа разбивается на части, за каждую из которых отвечает отдельный поток, эти потоки разделяют единое адресное пространство [2]. Если в системе больше одного процессора, то использование многопоточности фактически позволяет выполнять несколько действий одновременно [3].

Многие алгоритмы (умножение матриц, транспонирование матрицы, сортировка слиянием) допускают многопоточную реализацию [4]. При этом способы реализации многопоточности могут различаться.

Цель: сравнить производительность обычной и многопоточной версий алгоритма.

Для достижения поставленной цели необходимо выполнить следующие **задачи:**

- выбрать алгоритм;
- реализовать обычную версию алгоритма;
- реализовать многопоточную версию алгоритма;
- протестировать реализованные версии алгоритмов;
- провести сравнительный анализ двух версий алгоритмов;
- определить, всегда ли при задании количества потоков, равному удвоенному количеству логических ядер процессора, достигается наибольшая эффективность.

1 Аналитическая часть

1.1 Алгоритм цифрового дифференциального анализатора

Алгоритм цифрового дифференциального анализатора является тривиальным алгоритмом отрисовки отрезка, использующим инкрементальные значения для вычисления значений координат отрезка и операцию округления для аппроксимации ближайшим пикселем позиции точки отрезка на экране.

1.2 Алгоритм Брезенхэма

Алгоритм Брезенхэма построения отрезков используется в компьютерной графике для аппроксимации отрезков (т. к., учитывая структуру монитора, состоящего из пикселей, идеальными могут считаться только горизонтальные или вертикальные отрезки). Алгоритм Брезенхэма определяет, какие пиксели на экране нужно закрасить, чтобы получить приближённое изображение отрезка [5]. В отличие от алгоритма цифрового дифференциального анализатора алгоритм Брезенхэма уходит от тривиального способа вычисления позиций пикселей, учитывая дискретную структуру монитора и используя целочисленные вычисления, что позволяет добиться меньшего требуемого процессорного времени без потерь в качестве изображения.

1.3 Разбиение сектора на равные части

Для графического разбиения сектора на меньшие секторы равного размера можно использовать тривиальный алгоритм, который вычисляет инкрементальное значение угла, равное углу меньшего сектора, после чего рисует отрезки, используя один из алгоритмов отрисовки отрезков, отделяющие секторы.

1.4 Требования к вводу-выводу

На вход программе поступают следующие данные:

- абсцисса и ордината центра пучка;
- количество потоков;
- количество частей;
- длина отрезков;
- начальное смещение, в градусах;
- угол сектора, в градусах.

В результате своей работе программа отрисовывает заданную нами часть пучка.

1.5 Вывод

Рассмотрены алгоритм цифрового дифференциального анализатора построения отрезков, алгоритм Брезенхэма построения отрезков и алгоритм графического разбиения сектора на равные части.

2 Конструкторская часть

Исследуемый алгоритм – алгоритм разбиения сектора на равные секторы. В качестве алгоритма для отрисовки отрезков, отделяющих секторы, выбран алгоритм Брезенхэма. Схемы описанных алгоритмов приведены на рис. 2.1–2.4.

2.1 Схемы алгоритмов

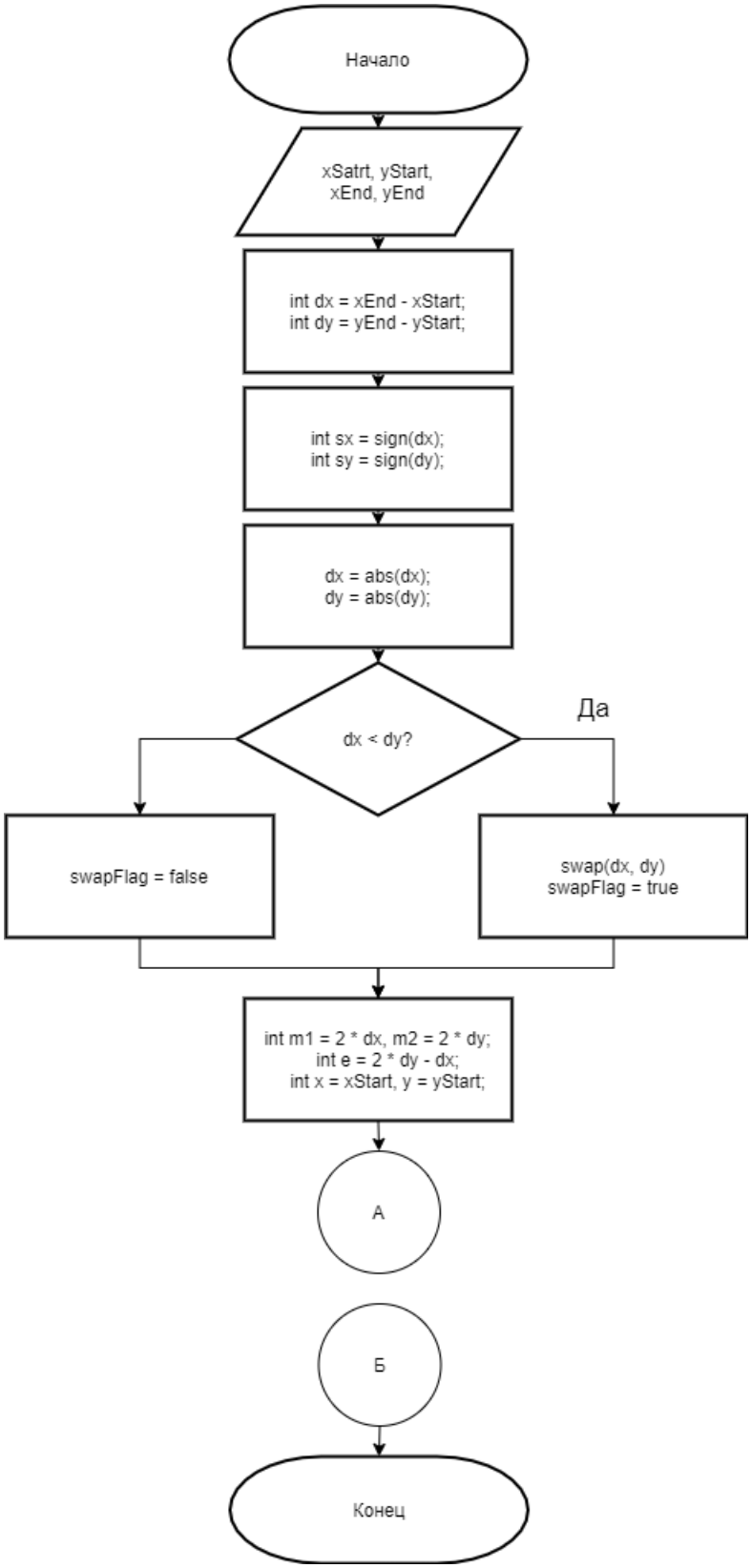

Рис. 2.1: Алгоритм Брезенхэма, ч. 1

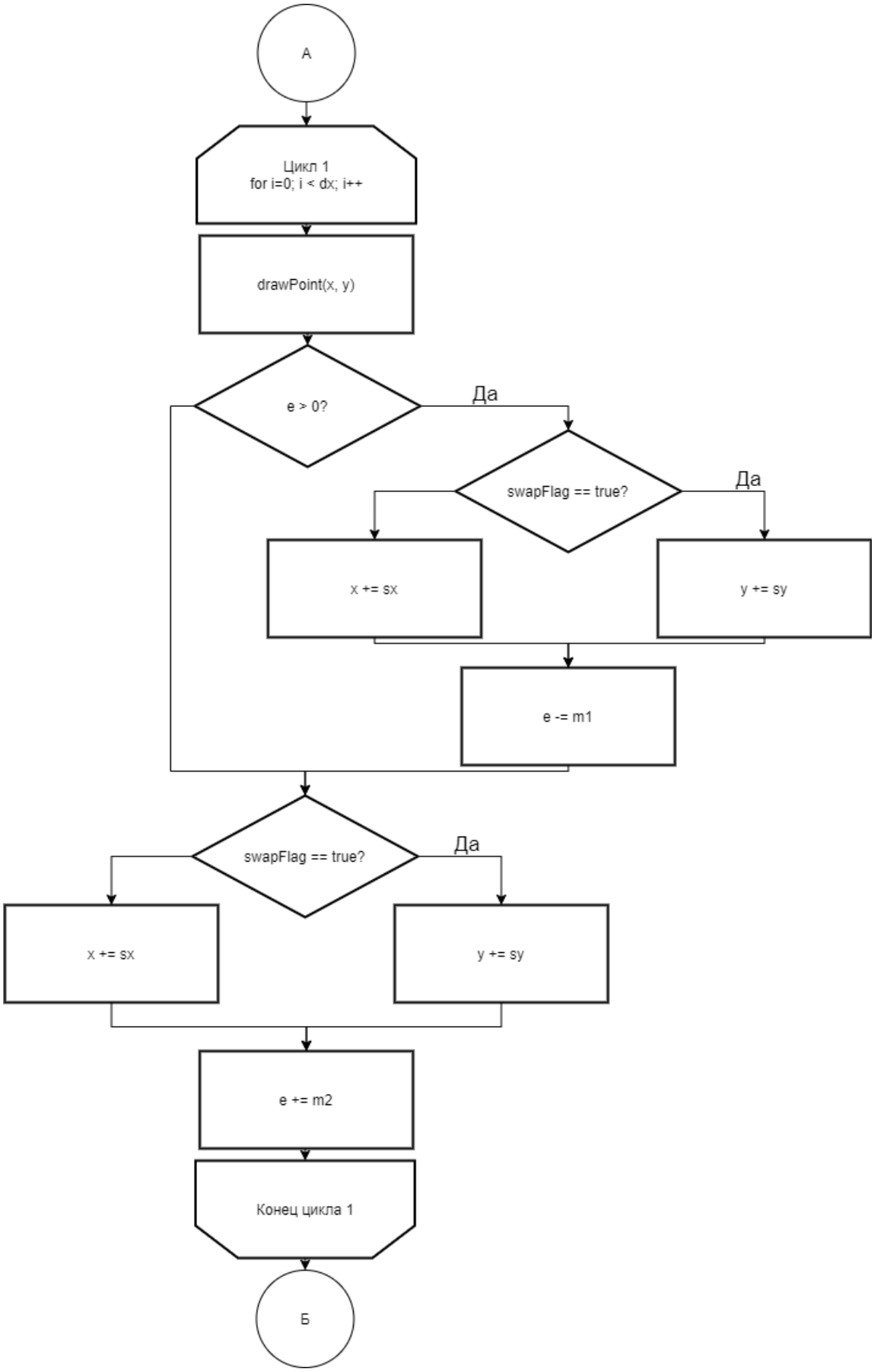

Рис. 2.2: Алгоритм Брезенхэма, ч. 2

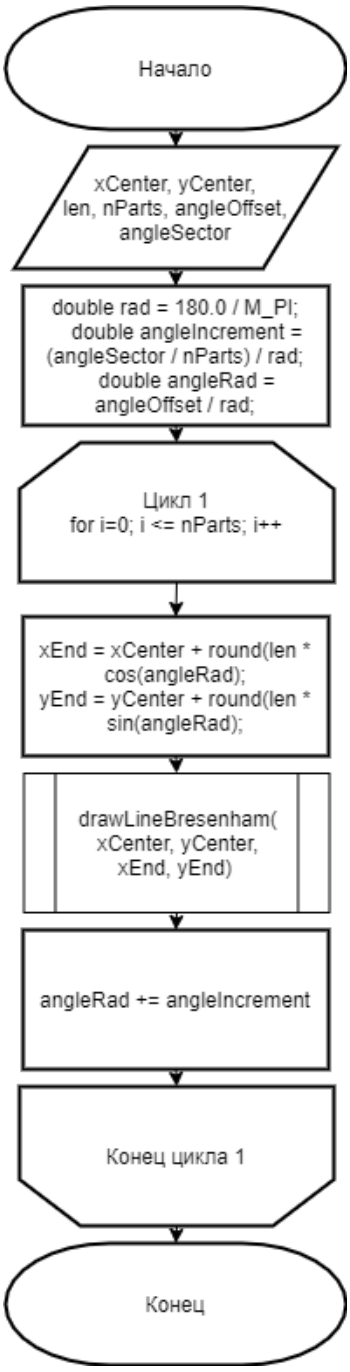

Рис. 2.3: Алгоритм отрисовки сектора, поделённого на равные части

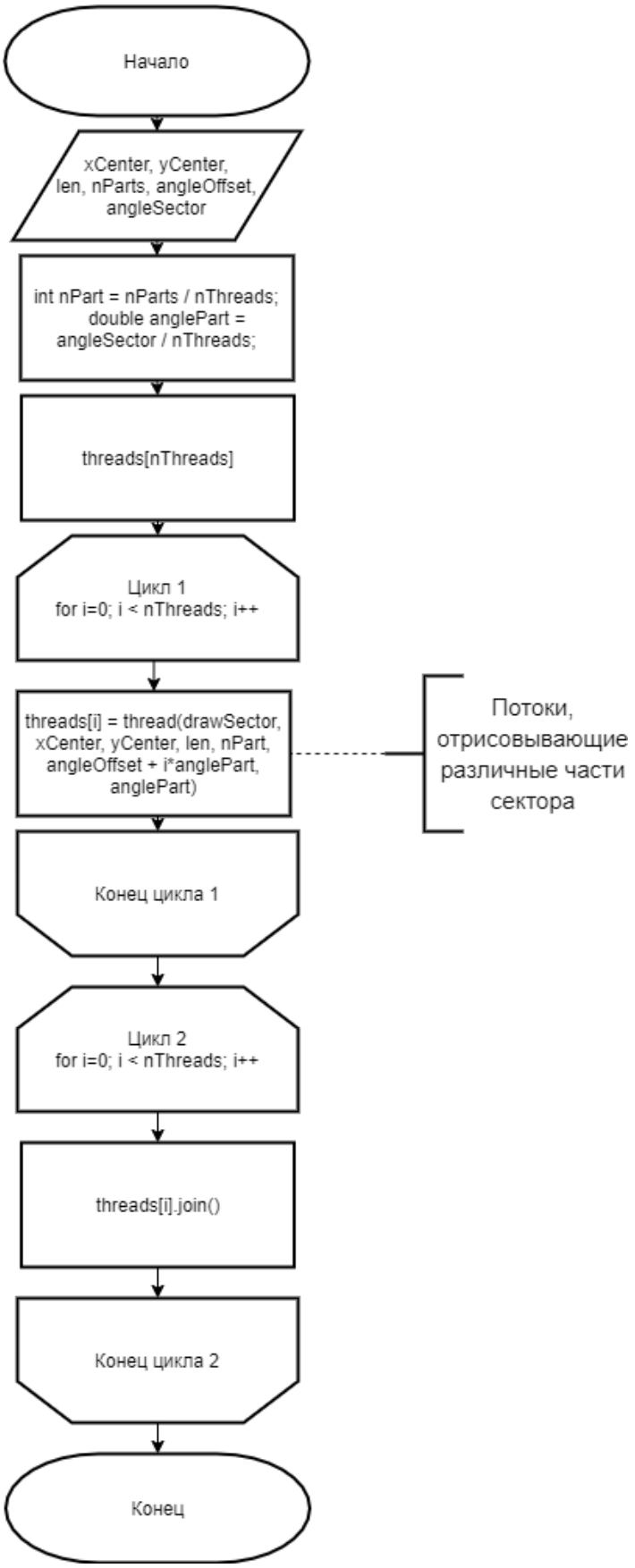

Рис. 2.4: Многопоточный алгоритм отрисовки сектора, поделённого на равные части

2.2 Использованные структуры данных

В лабораторной работе был использован фреймворк Qt. Для представления сцены была выбрана структуры `QGraphicsScene` и `QGraphicsPixmapItem`, для геометрических линий использовалась структура `QLineSeries`, для построения графиков – инструменты из `QtCharts`. Многопоточность программы реализована с помощью встроенной библиотеки `thread`. На рисунках 2.5 и 2.6 показан исходный код основных классов, используемых в программе.

```
class Interface
{
public:
    bool readInputInt(int &read, QLineEdit *line);
    bool readInputInt(int &read, QLineEdit *line, int leftBorder, int rightBorder);
};
```

Рис. 2.5: Класс `Interface`

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);

    ~MainWindow();

private slots:
    void on_btnDrawSector_clicked();

    void on_btnCls_clicked();

    void on_btnSavePlot_clicked();

private:
    Ui::MainWindow *ui;

    InputError readInput();

    void showMsgIncorrectInput(QString details);

    void initChart();

    void clearChart();

    void addPlot();

    void addPlot2();

    int xCenter = 0, yCenter = 0;

    int nParts = 1;

    int lineLength = 1;

    double angleOffset = 0;

    double angleSector = 0;

    int nThreads = 1;

    QImage mainImg;

    QImage drawImg;

    QImage bgImg;

    Interface user;

    QColor drawColor, bgColor;

    QGraphicsScene *scene = nullptr;

    QGraphicsPixmapItem *mainImgPixmap = nullptr;

    QChartView *chartView = nullptr;

    QChart *chart = nullptr;

    QChart *defaultChart = nullptr;

    QChart *firstChart = nullptr;

    QLogValueAxis *xAxis = nullptr;

    QValueAxis *xValueAxis = nullptr;

    QValueAxis *yAxis = nullptr;

    int chartMaxX = 0, chartMaxY = 0;
};
```

Рис. 2.6: Центральный класс `MainWindow`

Перегруженный метод `readInputInt` проверяет введённое число на корректность. Класс `MainWindow` является служебным для фреймворка и описывает класс графического интерфейса.

2.3 Выделенные классы эквивалентности

Классы эвивалентности были выделены на основании того, что наибольшая эффективность достигается при 8 или 16 потоках.

- 1 поток и 1 часть;
- 1 поток и большое количество частей (> 1);
- > 1 потока и 1 часть;
- > 1 потока и большое количество (> 1);
- > 8 потоков и большое количество частей (> 1).

2.4 Вывод

Приведены схемы алгоритма Брезенхэма отрисовки отрезка и алгоритма разбиения сектора на равные части.

3 Технологическая часть

3.1 Аппаратные характеристики и используемая операционная система

- Операционная система: Windows 10 Домашняя для одного языка, версия 21H1 (поддерживает многопоточные приложения [2, 3]);
- Процессор: Intel(R) Core(TM) i7-8550U [1];
- Количество ядер процессора: 4;
- Количество логических ядер: 8.

3.2 Используемый язык программирования

Выбранный язык программирования: C++. Причины:

- реализация многопоточности стандартной библиотекой C++ [6];
- мультипарадигмальность;
- большое количество справочной информации и литературы.

Используемая среда: Qt Creator 4.14. Причины:

- большое количество встроенных модулей;
- удобство.

3.3 Описание функций

- `drawLineBresenham` – отрисовка отрезка алгоритмом Брезенхэма
- `drawSector` – отрисовка сектора алгоритмом Брезенхэма
- `drawSectorMultiThread` – многопоточный алгоритм Брезенхэма

3.4 Реализация алгоритмов

В листингах 3.1–3.3 представлена реализация описанных в главе 2 алгоритмов.

Листинг 3.1: Алгоритм отрисовки отрезка Брезенхэма

```
void drawLineBresenham(int &xStart, int &yStart, int &xEnd, int &
   yEnd, QPainter &painter)
{
    int dx = xEnd - xStart;
    int dy = yEnd - yStart;
    int sx = sign(dx);
    int sy = sign( dy);
    dx = abs(dx);
    dy = abs(dy);

    bool swapFlag = false;
    if (dx < dy)
    {
        int tmp = dx;
        dx = dy;
        dy = tmp;
        swapFlag = true;
    }
    int  m1 = 2 * dx, m2 = 2 *
         dy;
    int  e = 2 * dy - dx;
    int  x = xStart, y = yStart;

    for  (int i = 0; i < dx; ++i)
    {
        painter.drawPoint(x, y);

        if (e > 0)
        {
            if (swapFlag)
                x += sx;
            else
                y += sy;
            e -= m1;
        }
        if (swapFlag)
            y += sy;
        else
            x += sx;
        e += m2;
    }
}
```

Листинг 3.2: Обычный алгоритм отрисовки сектора

```
void drawSector(int xCenter, int yCenter, int len, int nParts,
   double angleOffset, double angleSector, QPainter &painter)
{
    double rad = 180.0 / M_PI;
    double angleIncrement = (angleSector / nParts) / rad;
    double angleRad = angleOffset / rad;
    int xEnd, yEnd;

    for (int i = 0; i <= nParts; ++i) {
        xEnd = xCenter + std::round(len * cos(angleRad));
        yEnd = yCenter - std::round(len * sin(angleRad));
        drawLineBresenham(xCenter, yCenter, xEnd, yEnd, painter);
        angleRad += angleIncrement;
    }
}
```

Листинг 3.3: Многопоточный алгоритм отрисовки сектора

```
void drawSectorMultiThread(int xCenter, int yCenter, int len, int
   nParts, double angleOffset, double angleSector, QPainter &
   painter, int nThreads)
{
    int nPart = nParts / nThreads;
    double anglePart = angleSector / nThreads;
    std::vector<std::thread> threads(nThreads);
    for (int i = 0; i < nThreads; ++i) {
        threads[i] = std::thread(drawSector, xCenter, yCenter, len,
           nPart, angleOffset + i*anglePart, anglePart, std::ref(
           painter));
    }

    for (int i = 0; i < nThreads; ++i) {
        threads[i].join();
    }
}

```

1.2 Вывод

Реализован алгоритм разбиения сектора на равные секторы с использованием алгоритма Брезенхэма для отрисовки отрезков. Реализована многопоточная версия с использованием стандартной библиотеки C++ [6, 3].

4 Исследовательская часть

4.1 Тестирование

При тестировании проверялась корректность изображений, полученных при работе программы с различными данными. Тесты и соответствующие им данные для построения секторов представлены на рис. 4.1–4.3.

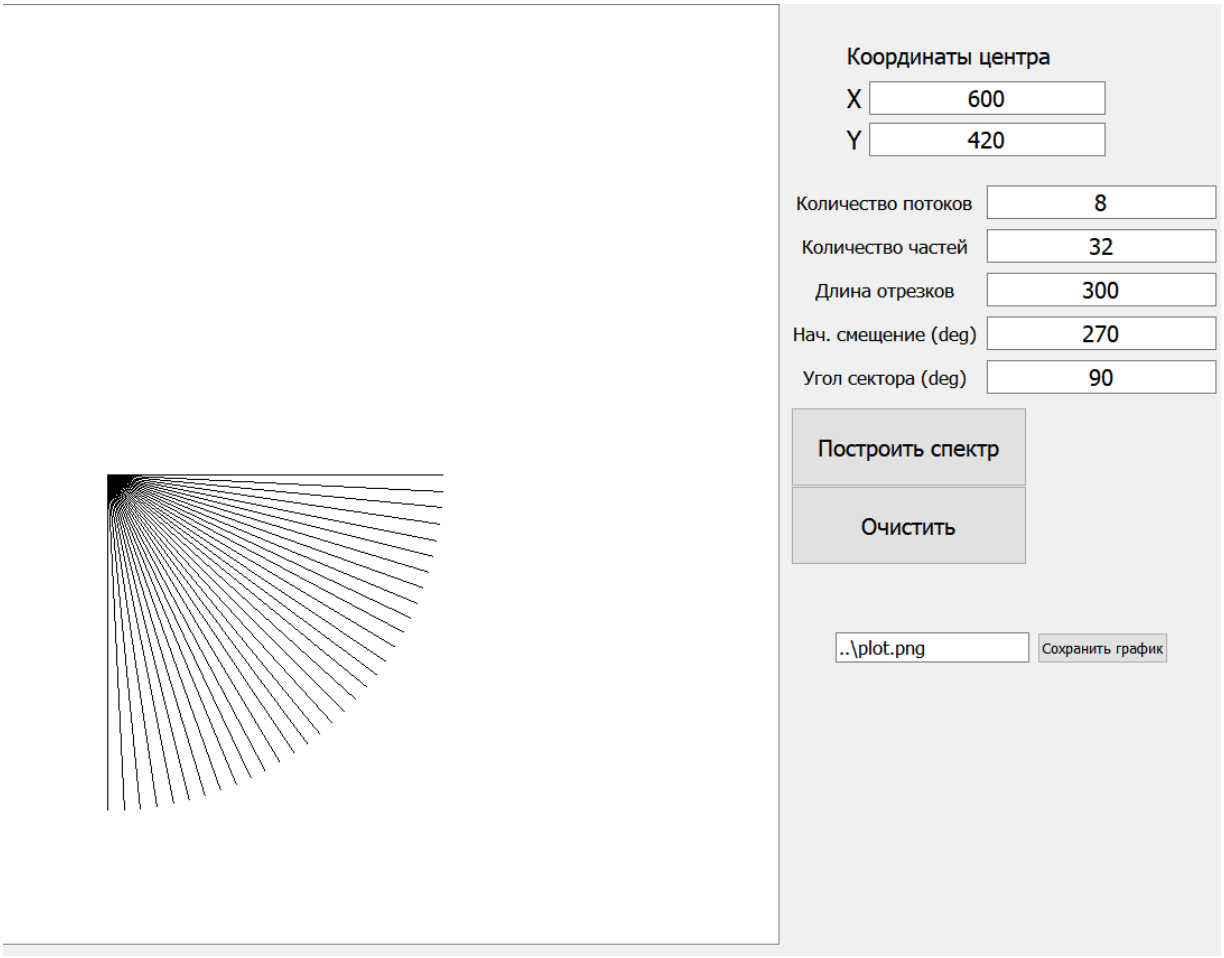

Рис. 4.1: Тест 1

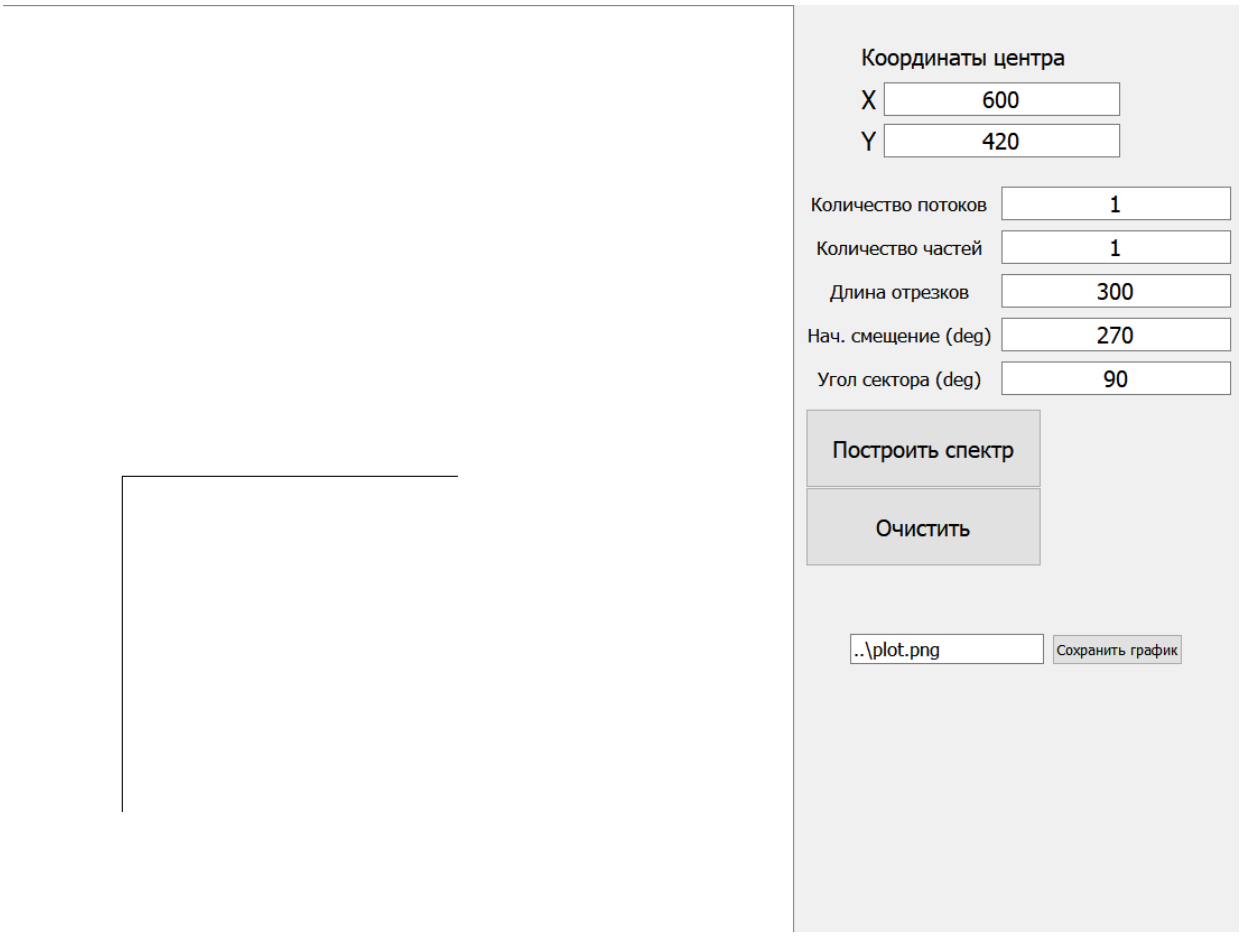
Координаты центра
X
600
Y
420
Количество потоков
1
Количество частей
1
Длина отрезков
300
Нач. смещение (deg)
270
Угол сектора (deg)
90
Построить спектр
Очистить
..\plot.png
Сохранить график

Рис. 4.2: Тест 2

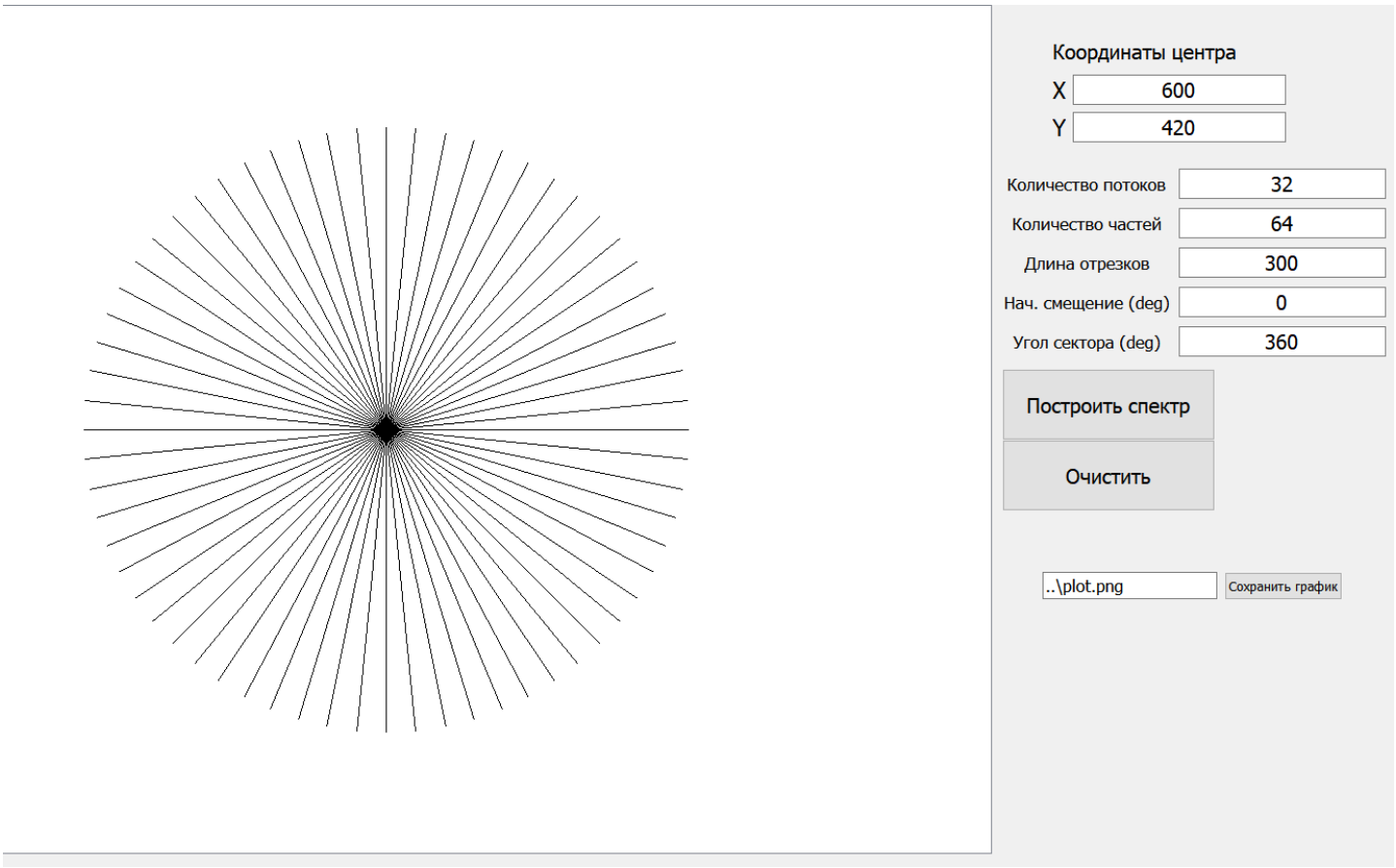

Рис. 4.3: Тест 3

Тестирование пройдено успешно.

1.3 Демонстрация работы программы

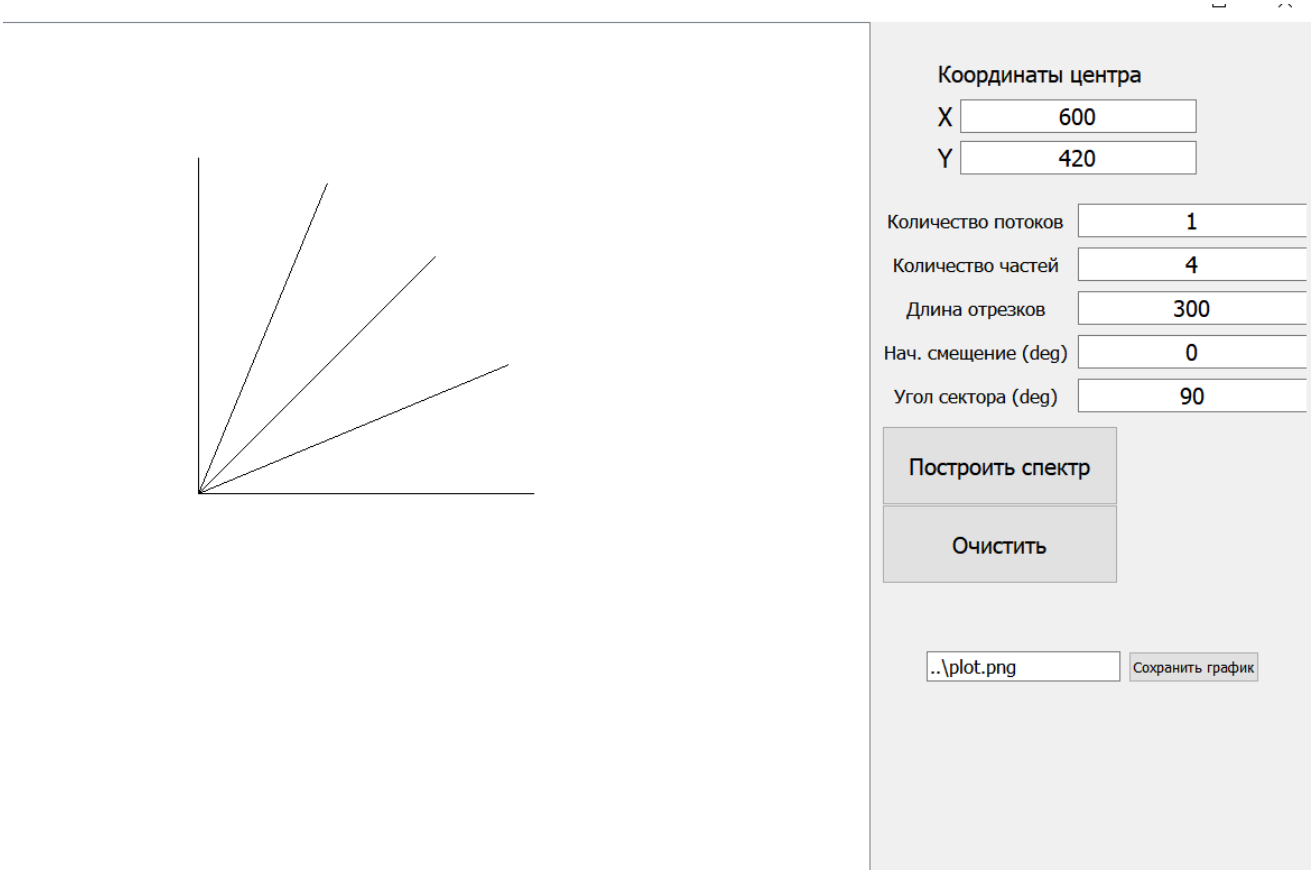

Рис. 4.4: Демонстрация работы программы

4.2 Анализ общего затрачиваемого времени

На рис. 4.5 представлена зависимость требуемого алгоритму общего времени от количества потоков. Для замера времени использовался библиотечный класс `QElapsedTimer` [7].

Данные, использованные при построении графика:

- количество точек: 6;
- количество потоков: 2^{i-1}, где i - порядковый номер точки;
- количество частей, на которые разбивается сектор: 352;
- начальное угловое смещение сектора (в градусах): 0;
- величина разбиваемого сектора (в градусах): 352;
- длины отрезков: 1000;
- количество повторных измерений: 1000.

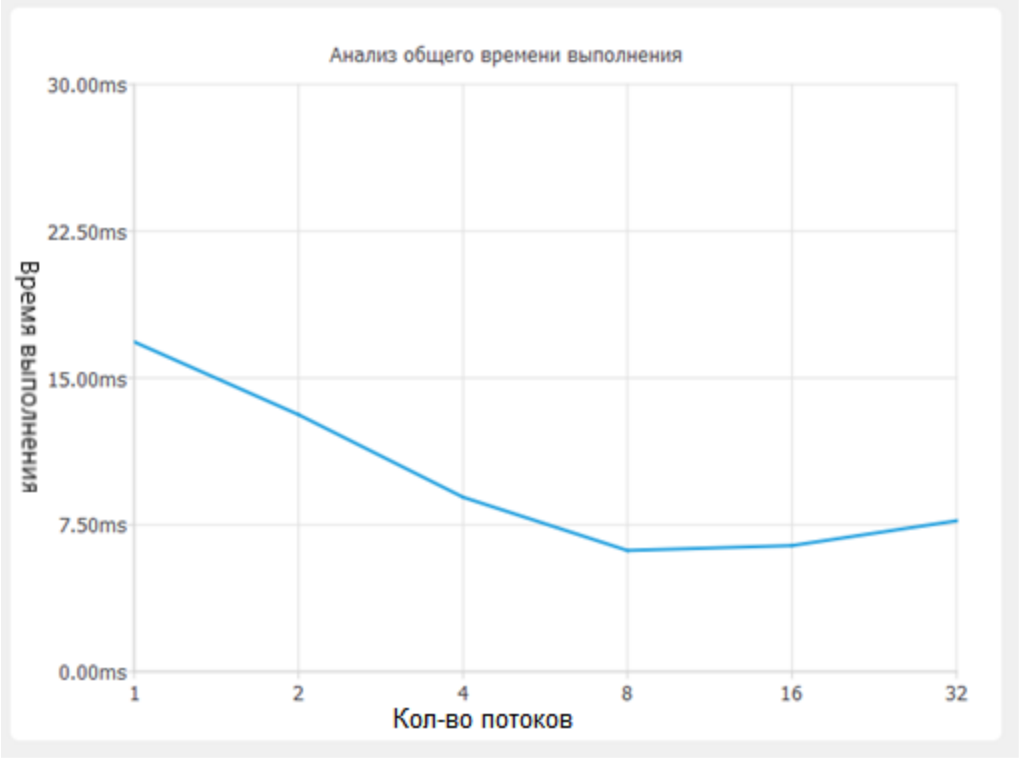

Рис. 4.5: Зависимость времени выполнения программы от количества потоков. Значения осей по вертикали время выполнения в миллисекундах, по горизонтали количество потоков

По представленному на рис. 4.5 графику видно, что наибольшая эффективность достигается при 8 или 16 потоках. Затраты времени снижаются более, чем в 2 раза.

4.3 Анализ распределения времени выполнения

При исследовании графика на рис. 4.5 установлено, что наибольшая эффективность достигается при использовании 8 или 16 потоков. Для более подробного анализа проведено дополнительное исследование распределения времени выполнения для 8 и 16 потоков, соответствующие графики представлены на рис. 4.6-4.7. Данные, использованные для построения графиков, аналогичны данным из раздела 4.3, но только для 8 и 16 потоков, количество повторений 10000.

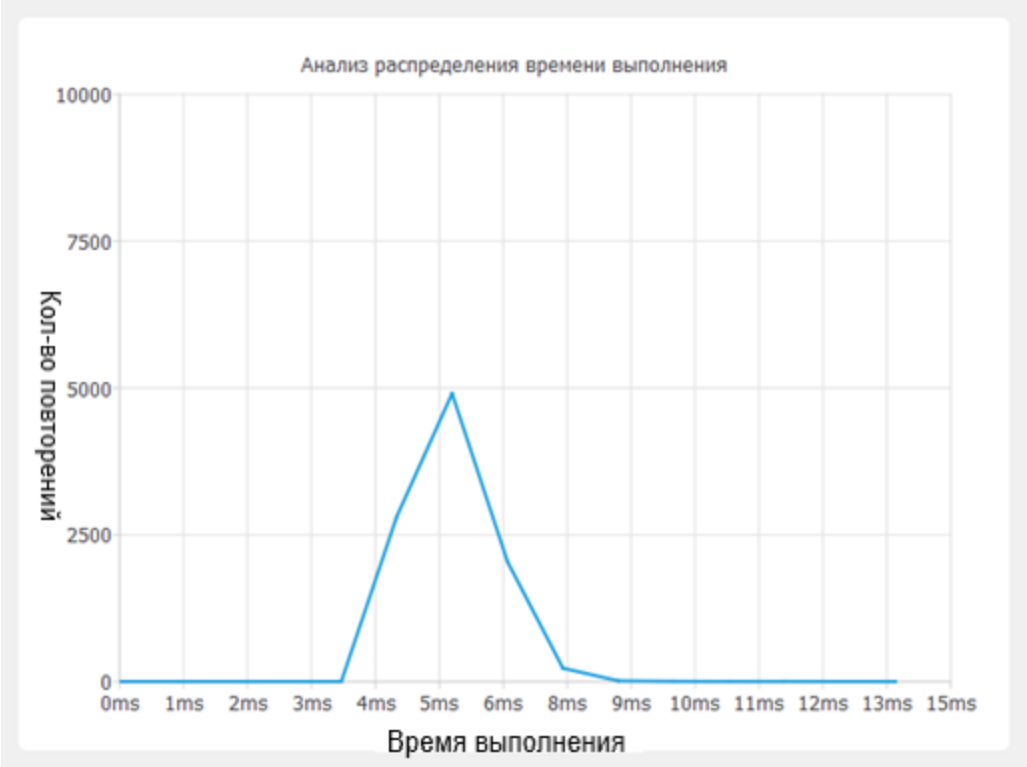

Рис. 4.6: Распределение времени выполнения при 8 потоках

Рис. 4.7: Распределение времени выполнения при 16 потоках

По представленным графикам на рис. 4.6 и 4.7 видно, что, в среднем, при использовании

8 потоков, следует ожидать большей эффективности, чем при использовании 16 потоков.

4.4 Вывод

Алгоритмы, реализованные в главе 3, успешно протестированы. Анализ общего затрачиваемого времени показал эффективность использования многопоточной версии алгоритма по сравнению с однопоточной (затраты времени снизились более, чем в 2 раза). Наибольшая эффективность достигается при использовании 8 потоков, что соответствует количеству логических ядер процессора [1] машины, на которой проводились исследования.

Заключение

Цель работы достигнута, все задачи выполнены:

- выбран алгоритм разбиения сектора на равные части;
- реализована обычная версия алгоритма;
- реализована многопоточная версия алгоритма;
- алгоритм успешно протестирован;
- проведён сравнительный анализ общего затрачиваемого времени в зависимости от количества потоков;
- проведён дополнительный сравнительный анализ распределения времени выполнения для 8 и 16 потоков.

Оказалось, что многопоточная версия алгоритма позволяет сократить общие затраты времени более, чем в 2 раза. Оптимальное количество потоков – восемь - соответствует количеству логических ядер процессора.

Список литературы

[1] Процессор Intel Core i7-1065G7 [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/196597/intel-core-i7-1065g7-processor- 8m-cache-up-to-3-90-ghz.html. 2021. Дата обращения: 03.10.2021.

[2] Внутреннее устройство Windows / Марк Руссинович, Дэвид Соломон, Алекс Ионеску [и др.]. 7 изд. 2018.

[3] Рихтер Джеффри. Windows для профессионалов. 2004.

[4] Cormen Thomas H. Introduction to Algorithms. 3 изд. 2009.

[5] Д. Роджерс. Алгоритмические основы машинной графики. 1989.

[6] Stroustrup Bjarne. The C++ Programming Language. 4 изд. 2013.

[7] QElapsedTimer Class [Электронный ресурс]. Режим доступа: https://doc.qt.io/qt-5/qelapsedtimer.html. 2021. Дата обращения: 28.09.2021.