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 №5 по курсу "Анализ алгоритмов"

Тема Конвейер

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Цель лабораторной работы

Изучение и реализация асинхронного взаимодействия потоков.

Задачи лабораторной работы

1. изучить асинхронное взаимодействие на примере конвейерной обработки данных;
2. спроектировать систему конвейерных вычислений;
3. реализовать систему конвейерных вычислений;
4. протестировать реализованную систему;
5. подготовить отчёт по проведенной работе.

1 | Аналитическая часть

В данном разделе описаны принцип и идея конвейерной обработки данных, а также алгоритм Брезенхема с действительными коэффициентами.

Работа алгоритма Брезенхема основывается на использовании понятия ошибка. Ошибкой здесь называется расстояние между действительным положением отрезка и ближайшим пикселем сетки растра, который аппроксимирует отрезок на очередном шаге.

На каждом шаге вычисляется величина ошибки и в зависимости от полученного значения выбирается пиксель, ближе расположенный к идеальному отрезку. Поскольку при реализации алгоритма на ЭВМ удобнее анализировать не само значение ошибки, а ее знак, то истинное значение ошибки смещается на -0,5.

Поскольку на первом шаге высвечивается пиксел с начальными координатами, то для него ошибка равняется 0, поэтому задаваемое предварительно значение этой ошибки:

1.1 Конвейерная обработка данных

Конвейерный принцип обработки данных подразумевает, что в каждый момент времени процессор работает над различными стадиями выполнения нескольких команд, причем на выполнение каждой стадии выделяются отдельные аппаратные ресурсы. Такая обработка оптимизирует использование ресурсов для заданного набора процессов, каждый из которых применяет эти ресурсы заранее предусмотренным способом. Идея конвейерной обработки данных заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции процессора представляются в виде последовательности более простых стадий. Вместо выполнения инструкций последовательно (ожидания завершения

конца одной инструкции и перехода к следующей), следующая инструкция может выполняться через несколько стадий выполнения первой инструкции. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца.

1.2 Алгоритм Брезенхема с действительными коэффициентами

Алгоритм Брезенхема — это алгоритм, определяющий, какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками [1].

Работа алгоритма Брезенхема основывается на использовании понятия ошибка. Ошибкой здесь называется расстояние между действительным положением отрезка и ближайшим пикселем сетки растра, который аппроксимирует отрезок на очередном шаге.

На каждом шаге вычисляется величина ошибки и в зависимости от полученного значения выбирается пиксель, ближе расположенный к идеальному отрезку. Поскольку при реализации алгоритма на ЭВМ удобнее анализировать не само значение ошибки, а ее знак, то истинное значение ошибки смещается на -0,5.

Поскольку на первом шаге высвечивается пиксел с начальными координатами, то для него ошибка равняется 0, поэтому задаваемое предварительно значение этой ошибки:

$$mistake = \frac{\Delta y}{\Delta x} - \frac{1}{2} \tag{1.1}$$

Выражение 1.1 фактически определяет ошибку для следующего шага.

В общем алгоритме Брезенхема большее по модулю из приращений принимается равным шагу растра, то есть единице, причем знак приращения совпадает со знаком разности конечной и начальной координат отрезка:

$$\Delta x = sign(x_e - x_s), \textbf{если } |x_e - x_s| \geq |y_e - y_s| \tag{1.2}$$

$$\Delta y = sign(y_e - y_s), \textbf{если } |y_e - y_s| \geq |x_e - x_s| \tag{1.3}$$

В выражениях 1.2, 1.3 x_e и y_e - координаты начала отрезка, а $sign$ - кусочно постоянная функция действительного аргумента.

Значение другой координаты идеального отрезка для следующего шага определяется как 1.4, поскольку приращение ординаты совпадает с величиной одного катета прямоугольного треугольника, а другой катет равен шагу сетки растра, то есть единице.

$$y_{ideal_i} = y_{ideal_{i+1}} + \frac{\Delta y}{\Delta x} \tag{1.4}$$

Ошибка на очередном вычисляется как:

$$mistake_{i+1} = y_{ideal_{i+1}} - y_{i+1} = y_{ideal_i} + \frac{\Delta y}{\Delta x} - y_i = mistake_i + \frac{\Delta y}{\Delta x} \tag{1.5}$$

В зависимости от полученного значения ошибки выбирается пиксел с той же ординатой (при ошибке <0) или пиксел с ординатой, на единицу большей, чем у предыдущего пиксела (при ошибке ≥ 0).

Поскольку предварительное значение ошибки вычисляется заранее, то во втором случае останется только вычесть единицу из значения ошибки, так как в этом случае $y_{i+1} = y_i + 1$, что не учитывалось при расчете.

Вывод

Были рассмотрены принцип и идея конвейерной обработки данных, а также алгоритм Брезенхема с действительными коэффициентами.

В данной работе стоит задача реализации системы конвейерной обработки данных для рассмотренного алгоритма.

2 | Конструкторская часть

В данном разделе представлены схемы алгоритма Брезенхема и реализации конвейерной обработки данных для алгоритма Брезенхема.

2.1 Схема алгоритма Брезенхема

Схема алгоритма Брезенхема предоставлена на рисунках 2.1, 2.2.

2.2 Схема реализации конвейерной обработки данных для алгоритма Брезенхема

На рисунке 2.3 предоставлена схема алгоритма работы функции, запускающей в требуемом количестве потоков функцию-аргумент.

Вывод

Были представлены схемы алгоритма Брезенхема, а также реализации конвейерной обработки данных для данного алгоритма.

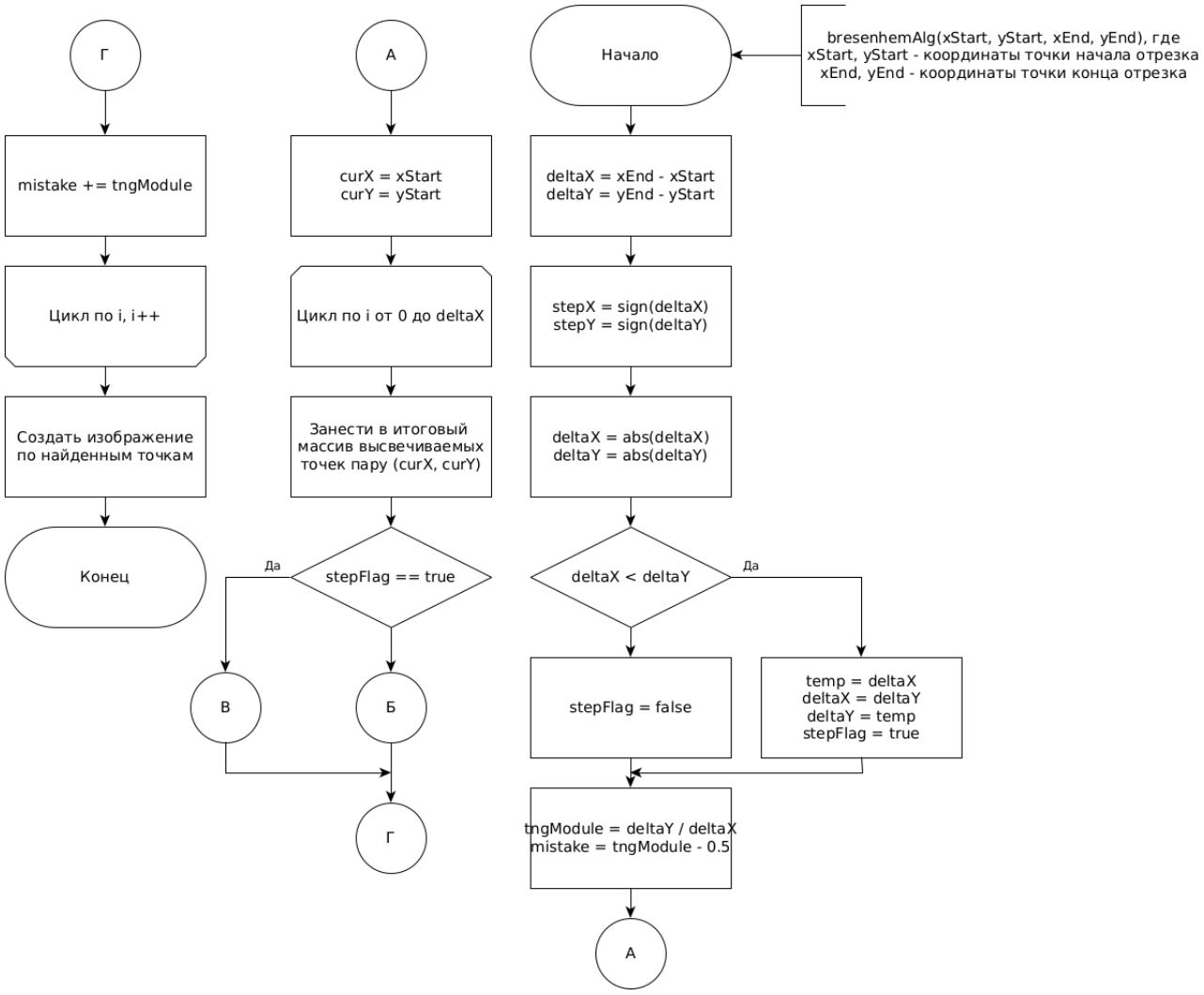

Рис. 2.1: Схема алгоритма Брезенхема.

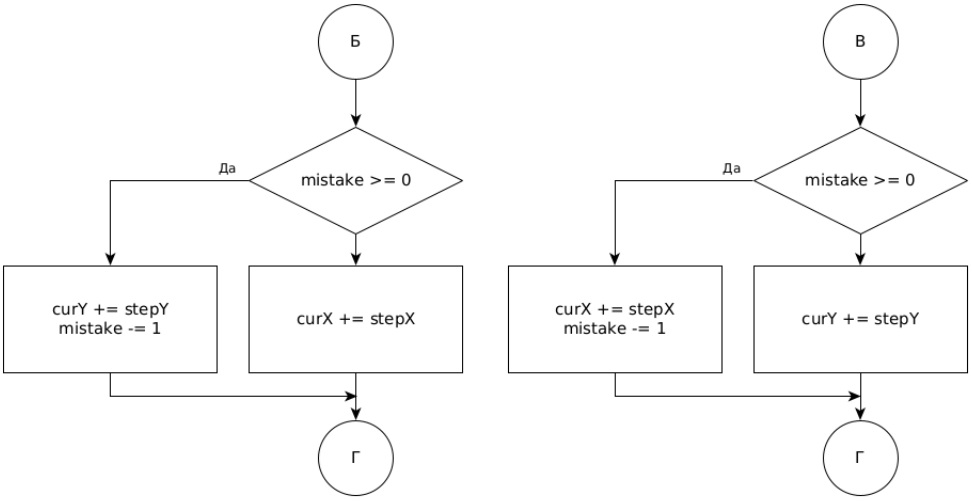

Рис. 2.2: Схема алгоритма Брезенхема.

Рис. 2.3: Схема реализации конвейерной обработки данных для алгоритма Брезенхема.

3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации программного обеспечения, а также листинг кода.

3.1 Требования к программному обеспечению

- входные данные - количество выполняемых задач (количество растеризуемых отрезков);
- выходные данные - записи времени прихода и ухода обрабатываемых заявок для каждого реализованного конвейера.

3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования C++ [2].

Данный выбор обусловлен следующими факторами:

- Данный язык программирования преподавался в рамках курса объектно-ориентированного программирования;
- Высокая вычислительная производительность;
- Большое количество справочной и учебной литературы в сети Интернет.

При написании программного продукта использовалась среда разработки QT Creator [3].

Данный выбор обусловлен следующими факторами:

- Основы работы с данной средой разработки преподавался в рамках курса программирования на Си;
- QT Creator позволяет работать с расширением QtDesign.

3.3 Листинг кода

В листингах 3.1 и 3.2 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Разбиение алгоритма Брезенхема

```
1  std::string now_str()
2  {
3      const boost::posix_time::ptime now = boost::posix_time::
           microsec_clock::local_time();
4
5      const boost::posix_time::time_duration td = now.time_of_day()
           ;
6
7      const long hours = td.hours();
8      const long minutes = td.minutes();
9      const long seconds = td.seconds();
10     const long nanoseconds =
11         td.total_nanoseconds() - ((hours * 3600 + minutes * 60 +
               seconds) * 1000000000);
12
13     char buf[40];
14     sprintf(buf, "%02ld:%02ld:%02ld.%09ld", hours, minutes,
           seconds, nanoseconds);
15
16     return buf;
17 }
18
19 SegmentRasterizator::SegmentRasterizator(int xStart_, int yStart_
       , int xEnd_, int yEnd_)
20 {
21     xStart = xStart_;
22     yStart = yStart_;
23     xEnd = xEnd_;
24     yEnd = yEnd_;
25     if (xStart == xEnd)
26         xEnd += 1;
27     else if (yStart == yEnd)
```

```
28          yEnd += 1;
29
30      image = new QImage(WIDTH, HEIGHT, QImage::Format_RGB32);
31      image->fill(Qt::white);
32  }
33
34  int sign(float num)
35  {
36      return (num < -__FLT_EPSILON__) ? -1 : ((num >
            __FLT_EPSILON__) ? 1 : 0);
37  }
38
39  void SegmentRasterizator::prepareConstantsForRB(int index)
40  {
41      std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
            BEGIN %s" ANSI_RESET "\n", index, now_str().c_str());
42
43      deltaX = xEnd - xStart;
44      deltaY = yEnd - yStart;
45
46      stepX = sign(deltaX);
47      stepY = sign(deltaY);
48
49      deltaX = std::abs(deltaX);
50      deltaY = std::abs(deltaY);
51
52      if (deltaX < deltaY)
53      {
54          std::swap(deltaX, deltaY);
55          stepFlag = true;
56      }
57      else
58          stepFlag = false;
59
60      tngModule = deltaY / deltaX;
61      mistake = tngModule - 0.5;
62
63      std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
            ENDED %s" ANSI_RESET "\n", index, now_str().c_str());
64  }
65
66  void SegmentRasterizator::rastSegment(int index)
67  {
68      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            BEGIN %s" ANSI_RESET "\n", index, now_str().c_str());
```

```
69
70      float curX = xStart, curY = yStart;
71      for (int i = 0; i <= deltaX; i++)
72      {
73          dotsOfSegment.push_back(std::pair<int, int>(curX, curY));
74          if (stepFlag)
75          {
76              if (mistake >= 0)
77                  (curX += stepX, mistake--);
78              curY += stepY;
79          }
80          else
81          {
82              if (mistake >= 0)
83                  (curY += stepY, mistake--);
84              curX += stepX;
85          }
86          mistake += tngModule;
87      }
88
89      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            ENDED %s" ANSI_RESET "\n", index, now_str().c_str());
90  }
91
92  void SegmentRasterizator::createImg(int index)
93  {
94      std::printf(ANSI_CYAN_BRIGHT"From END worker: task %d BEGIN %
            s" ANSI_RESET "\n", index, now_str().c_str());
95
96      for (auto iter = dotsOfSegment.begin(); iter < dotsOfSegment.
            end(); iter++)
97          image->setPixel(iter->first, iter->second, Qt::black);
98
99      std::printf(ANSI_CYAN_BRIGHT "From END worker: task %d ENDED
            %s" ANSI_RESET "\n", index, now_str().c_str());
100 }
101
102 std::vector<std::pair<int, int>> SegmentRasterizator::
        getDotsOfSegment()
103 {
104     return dotsOfSegment;
105 }
```

Листинг 3.2: Менеджер потоков

```
1  Director::Director(std::queue<SegmentRasterizator> &startQueue_)
2  {
3      startQueue = startQueue_;
4  }
5
6  void Director::processPrepare()
7  {
8      int i = 0;
9      for (SegmentRasterizator curSeg(startQueue.front());
           startQueue.size();
10          startQueue.pop(), curSeg = startQueue.front())
11     {
12
13         curSeg.prepareConstantsForRB(i++);
14         middleQueue.push(curSeg);
15     }
16 }
17
18 void Director::processRast()
19 {
20     int i = 0;
21     while (startQueue.size() || middleQueue.size())
22     {
23         if (middleQueue.empty())
24             continue;
25         SegmentRasterizator curSeg(middleQueue.front());
26
27         curSeg.rastSegment(i++);
28
29     endQueue.push(curSeg);
30         middleQueue.pop();
31     }
32 }
33
34 void Director::processCreate()
35 {
36     int i = 0;
37     while (startQueue.size() || middleQueue.size() || endQueue.
           size())
38     {
39         if (endQueue.empty())
40             continue;
41         SegmentRasterizator curSeg(endQueue.front());
42
43         curSeg.createImg(i++);
```

```
44          endQueue.pop();
45          final.push_back(curSeg);
46      }
47  }
48
49  void Director::initWork()
50  {
51      workers[0] = std::thread(&Director::processPrepare, this);
52      workers[1] = std::thread(&Director::processRast, this);
53      workers[2] = std::thread(&Director::processCreate, this);
54
55      workers[0].join();
56      workers[1].join();
57      workers[2].join();
58  }
59
60  std::vector<SegmentRasterizator> Director::getFinal() { return
        final; }
```

3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций, реализующих алгоритм Брезенхема. Тесты пройдены успешно.

Таблица 3.1: Тестирование функций

Точка начала отрезка (x, y)	Точка конца отрезка (x, y)	Ожидаемый результат
(1, 1)	(3, 3)	(1, 1), (2, 2), (3, 3)
(1, 1)	(1, 3)	(1, 1), (1, 2), (1, 3)
(1, 1)	(2, 1)	(1, 1), (2, 1)
(3, 3)	(1, 1)	(1, 1), (2, 2), (3, 3)

Вывод

Спроектированные алгоритмы были реализованы и протестированы.

4 | Исследовательская часть

4.1 Технические характеристики

Технические характеристики ЭВМ, на котором выполнялись исследования:

- ОС: Manjaro Linux 20.1.1 Mikah
- Оперативная память: 16 Гб
- Процессор: Intel Core i7-10510U

При проведении замеров времени ноутбук был подключен к сети электропитания.

4.2 Пример работы программного обеспечения

На рисунке 4.1 приведен пример работы программы для 7 визуализируемых отрезков.

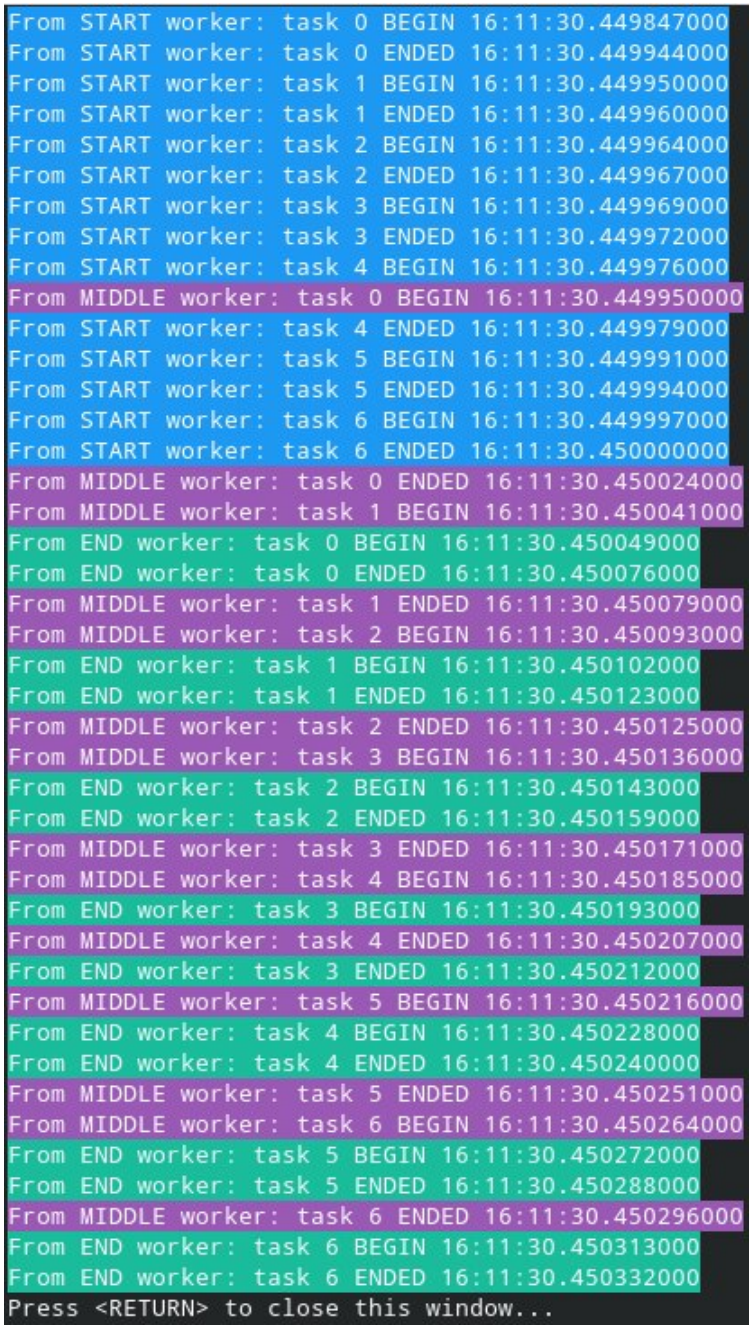

Рис. 4.1: Пример работы ПО.

4.3 Время выполнения алгоритмов

Алгоритм тестировался на данных, сгенерированных случайным образом.

В таблице 4.1 предоставлено время работы над каждым отрезком в предоставленном примере каждого из выделенных этапов.

Из таблицы видно, что среднее время выполнения этапа 1 составляет ≈ 17428.6 наносекунд. Среднее время выполнения этапа 2 составляет ≈ 38285.7 наносекунд. Среднее время выполнения этапа 3 составляет ≈ 18571.4 наносекунд. Таким образом, этап 1 сравним по среднему времени выполнения с этапом 2. Но после выполнения этапа 1 заметно, что последующие вызовы функции работают за константное время, равное 3000 наносекунд, что при наличии начальных "прогревочных"запусков вылилось бы в факт того, что этап 1 не был бы сопоставим по среднему времени выполнения с этапом 3. Этап 2 является самым долго выполняющимся.

Таблица 4.1: Замеры времени для выполнения выделенных этапов.

	Время обработки, нс		
Номер отрезка	Этап 1	Этап 2	Этап 3
0	97000	74000	27000
1	10000	38000	21000
2	3000	32000	16000
3	3000	35000	19000
4	3000	22000	12000
5	3000	35000	16000
6	3000	32000	19000

Вывод

При сравнении результатов замеров по времени стало известно, что самым быстрым этапом конвейера оказался этап 1. При этом, самым медленным из трех рассмотренных - этап 3.

В среднем этап 1 работает быстрее этапа 2 на ≈ 20857.1 наносекунд. При этом, при четвертой обработке отрезка разница в скорости выполнения составила 32000 наносекунд.

Этап 3 в среднем работает быстрее этапа 2 на ≈ 19714.3 наносекунд. При этом, при шестой обработке отрезка разница в скорости выполнения составила 19000 наносекунд.

Таким образом, среднее время выполнения алгоритма для каждого отрезка составило ≈ 74285.71 наносекунд.

Заключение

В ходе выполнения лабораторной работы была выполнена цель и следующие задачи:

1. было изучено асинхронное взаимодействие на примере конвейерной обработки данных;
2. была спроектирована система конвейерных вычислений;
3. была реализована система конвейерных вычислений;
4. была протестирована реализованная система;
5. был подготовлен отчёт по проведенной работе.

Исследования показали, что в среднем:

1. этап 1 работает быстрее этапа 2 на ≈ 20857.1 наносекунд;
2. этап 3 работает быстрее этапа 2 на ≈ 19714.3 наносекунд;
3. среднее время выполнения алгоритма составило ≈ 74285.71 наносекунд.

Литература

[1] Роджерс Дэвид. Алгоритмические основы машинной графики. Мир, 1989. с. 512.

[2] C++ standart [Электронный ресурс]. Режим доступа: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf` (дата обращения 01.12.2020).

[3] Qt documentation [Электронный ресурс]. Режим доступа: `https://doc.qt.io/` (дата обращения 01.12.2020).