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 **3**

1 Аналитическая часть **4**
1.1 Конвейерная обработка данных 4
1.2 Алгоритм Брезенхема с действительными коэффициентами 5

2 Конструкторская часть **7**
2.1 Схема алгоритма Брезенхема 7
2.2 Схема реализации конвейерной обработки данных для алгоритма Брезенхема 7

3 Технологическая часть **10**
3.1 Требования к программному обеспечению 10
3.2 Средства реализации программного обеспечения 10
3.3 Листинг кода 11
3.4 Тестирование программного продукта 15

4 Исследовательская часть **16**
4.1 Пример работы программного обеспечения 16
4.2 Технические характеристики 17
4.3 Время выполнения алгоритмов 17

Заключение **24**

Литература **24**

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.

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.

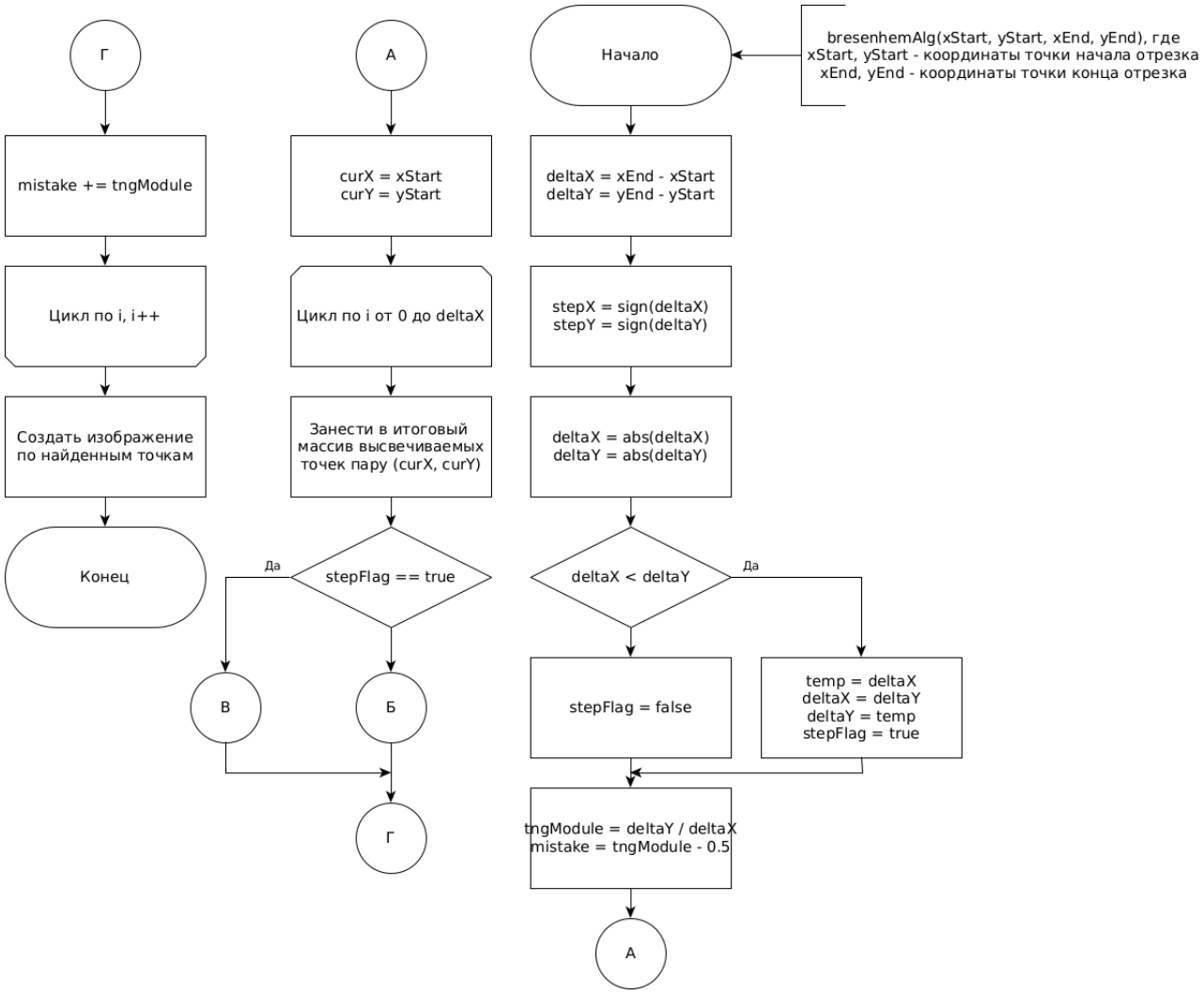

Рис. 2.1: Схема алгоритма Брезенхема.

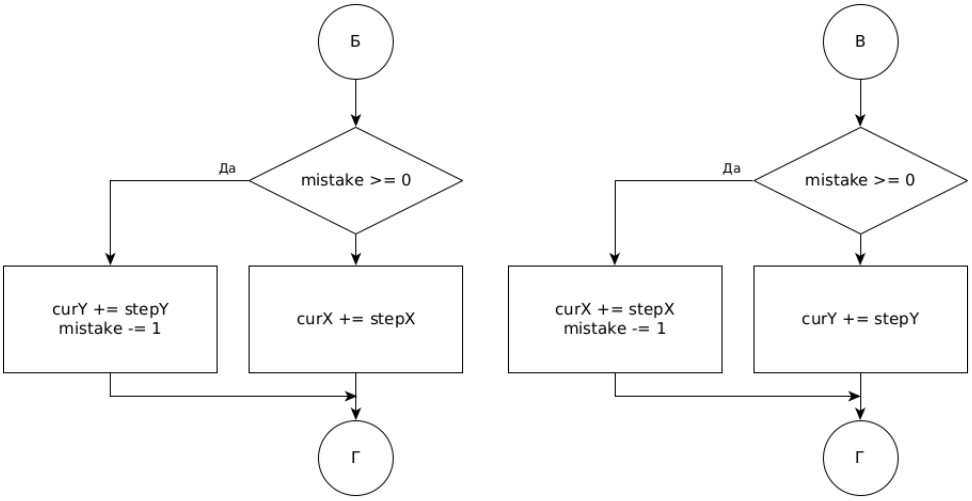

Рис. 2.2: Схема алгоритма Брезенхема.

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++.

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.

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 milliseconds =
11         td.total_milliseconds() - ((hours * 3600 + minutes * 60 +
               seconds) * 1000);
12
13     char buf[40];
14     sprintf(buf, "%02ld:%02ld:%02ld.%03ld", hours, minutes,
           seconds, milliseconds);
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 = NULL;
31  }
32
33  int sign(float num)
34  {
35      return (num < -__FLT_EPSILON__) ? -1 : ((num >
            __FLT_EPSILON__) ? 1 : 0);
36  }
37
38  void SegmentRasterizator::prepareConstantsForRB(int index)
39  {
40      std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
            BEGIN %s\n" ANSI_RESET, index, now_str().c_str());
41
42      deltaX = xEnd - xStart;
43      deltaY = yEnd - yStart;
44
45      stepX = sign(deltaX);
46      stepY = sign(deltaY);
47
48      deltaX = std::abs(deltaX);
49      deltaY = std::abs(deltaY);
50
51      if (deltaX < deltaY)
52      {
53          std::swap(deltaX, deltaY);
54          stepFlag = true;
55      }
56      else
57          stepFlag = false;
58
59      tngModule = deltaY / deltaX;
60      mistake = tngModule - 0.5;
61
62      std::printf(ANSI_BLUE_BRIGHT "From START worker: task %d
            ENDED %s\n" ANSI_RESET, index, now_str().c_str());
63  }
64
65  void SegmentRasterizator::rastSegment(int index)
66  {
67      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            BEGIN %s\n" ANSI_RESET, index, now_str().c_str());
68
```

```
69      float curX = xStart, curY = yStart;
70      for (int i = 0; i <= deltaX; i++)
71      {
72          dotsOfSegment.push_back(std::pair<int, int>(curX, curY));
73          if (stepFlag)
74          {
75              if (mistake >= 0)
76                  (curX += stepX, mistake--);
77              curY += stepY;
78          }
79          else
80          {
81              if (mistake >= 0)
82                  (curY += stepY, mistake--);
83              curX += stepX;
84          }
85          mistake += tngModule;
86      }
87
88      std::printf(ANSI_MAGENTA_BRIGHT "From MIDDLE worker: task %d
            ENDED %s\n" ANSI_RESET, index, now_str().c_str());
89  }
90
91  void SegmentRasterizator::createImg(int index)
92  {
93      std::printf(ANSI_CYAN_BRIGHT"From END worker: task %d BEGIN %
            s\n" ANSI_RESET, index, now_str().c_str());
94
95      if (image)
96          delete image;
97      image = new QImage(WIDTH, HEIGHT, QImage::Format_RGB32);
98      image->fill(Qt::white);
99
100     for (auto iter = dotsOfSegment.begin(); iter < dotsOfSegment.
            end(); iter++)
101         image->setPixel(iter->first, iter->second, Qt::black);
102
103     std::printf(ANSI_CYAN_BRIGHT " From END worker: task %d ENDED
            %s\n" ANSI_RESET, index, now_str().c_str());
104 }
```

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 Пример работы программного обеспечения

Ниже на рисунках 4.1-4.2 предоставлены примеры работы каждого из алгоритмов на введённых пользователем данных.

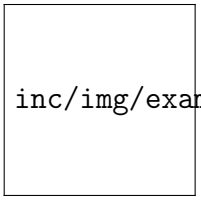

Рис. 4.1: Пример работы ПО.

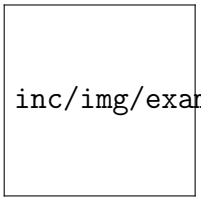

Рис. 4.2: Пример работы ПО.

4.2 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.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались на данных, сгенерированных случайным образом.

Результаты замеров времени приведены в таблицах 4.1 - 4.5. На рисунках 4.3 и 4.7 приведены графики зависимостей времени работы алгоритмов от размерности обрабатываемых матриц при 1, 2, 4, 8, 16 потоках. В таблице КВ - последовательный алгоритм Копперсмита-Винограда, Парал1КВ - реализация схемы **??** параллельного алгоритма Копперсмита-Винограда, Парал2КВ - реализация схемы **??** параллельного алгоритма.

Таблица 4.1: Замеры времени для различных размеров массивов на 1 потоке.

	Время работы, нс		
Размеры матрицы	КВ	Парал1КВ	Парал2КВ
100	4692982	6144734	6125329
200	49620266	56025172	58077386
300	178730542	199925329	206518482
400	429056319	456832433	561695647
500	856058247	737473554	673711435

Таблица 4.2: Замеры времени для различных размеров массивов на 2 потоках.

	Время работы, нс		
Размеры матрицы	КВ	Парал1КВ	Парал2КВ
100	4250527	4315274	2910312
200	50728798	34809356	18273241
300	186202130	120880876	61885504
400	436860056	248104438	137340969
500	870833440	576761740	267199177

Таблица 4.3: Замеры времени для различных размеров массивов на 4 потоках.

	Время работы, нс		
Размеры матрицы	КВ	Парал1КВ	Парал2КВ
100	4485824	4239416	2969518
200	51678241	28600648	10260636
300	176448578	80224609	31277241
400	436787067	208705553	70302333
500	850999256	485612161	151094601

Таблица 4.4: Замеры времени для различных размеров массивов на 8 потоке.

	Время работы, нс		
Размеры матрицы	КВ	Парал1КВ	Парал2КВ
100	4403638	5304634	3712704
200	52234883	26568390	11046163
300	177331754	107137650	31637714
400	442171563	337629280	58383234
500	877219464	883596315	117636357

Вывод

При сравнении результатов замеров по времени видно, что рассматривать скорость работы распараллеленных алгоритмов на одном потоке смысла нет, так как они работают медленнее на $\approx 16\%$. Связанно это с тем, что на создание потоков тратится время.

На 2-х потоках обе схемы себя показывают уже лучше. Схема распараллеливания с разделением показывает себя хуже, чем вторая схема, на $\approx 181\%$ (на размерности 300). При этом последовательная реализация будет работать медленнее схемы без разделения на $\approx 326\%$.

На 4-х потоках схемы вновь показывают себя лучше. При этом они работают быстрее, чем на 2-х потоках. Схема распараллеливания с разделением показывает себя хуже, чем вторая схема, на $\approx 297\%$.

Таблица 4.5: Замеры времени для различных размеров массивов на 16 потоках.

	Время работы, нс		
Размеры матрицы	КВ	Парал1КВ	Парал2КВ
100	4463337	7381390	4981564
200	50528949	32361454	17479130
300	175677695	110593792	44960321
400	433177913	342768739	68839512
500	831773652	887656714	100236284

На 8-и потоках первая схема показывает себя уже хуже, чем на 4-х потоках. Таким образом 4 потока - это оптимальный вариант решения задачи для первой схемы. Такое увеличение времени выполнения может связано со временем, затрачиваемым на создание потоков. Схема распараллеливания с разделением показывает себя хуже, чем вторая схема на $\approx 578\%$.

На 16-ти потоках схемы работают с ещё меньшей эффективностью. Схема распараллеливания с разделением показывает себя хуже второй реализации на $\approx 500\%$. Таким образом, вторая реализация параллельной схемы уже начинает несколько уменьшать своё преимущество, но при этом она всё ещё имеет свой потенциал.

Таким образом, параллельная реализация действительно работает быстрее последовательной. Но важно отметить факт того, что требуется ответственно относиться к выделяемым ресурсам.

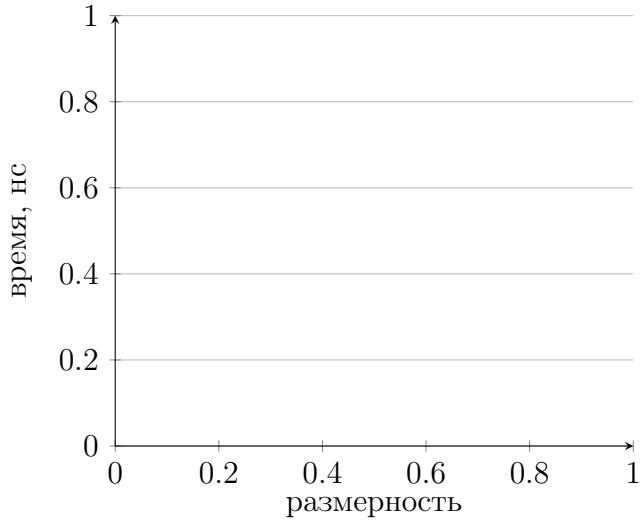

Рис. 4.3: Зависимость времени работы от размерности матриц (при 1 потоке).

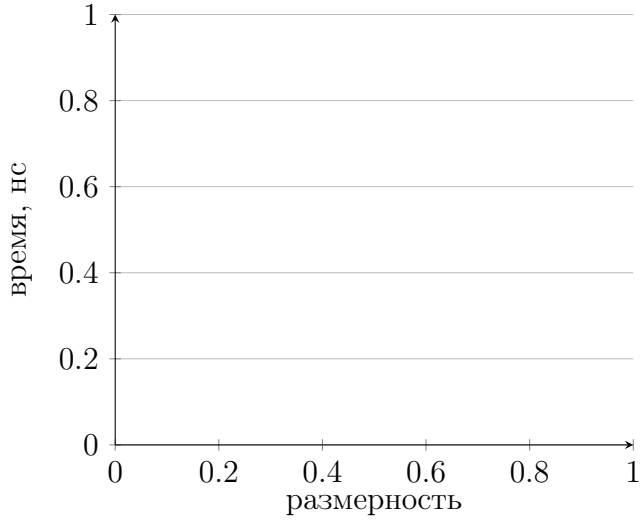

Рис. 4.4: Зависимость времени работы от размерности матриц (при 2 потоках)

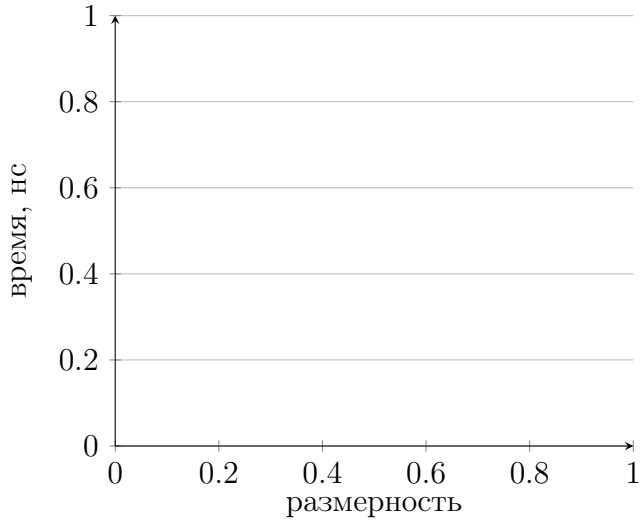

Рис. 4.5: Зависимость времени работы от размерности матриц (при 4 потоках)

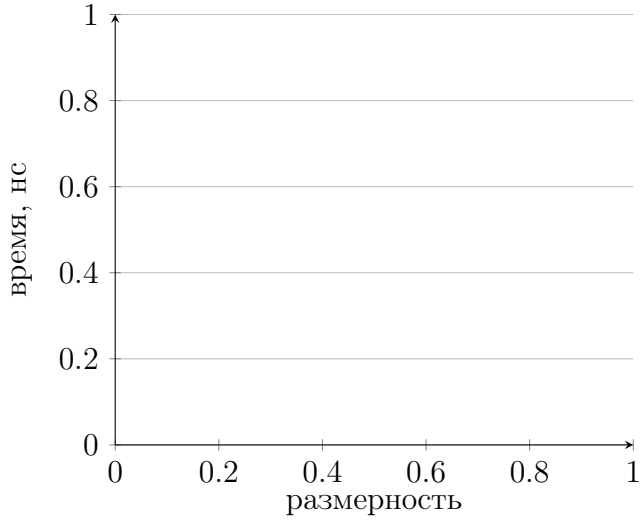

Рис. 4.6: Зависимость времени работы от размерности матриц (при 8 потоках)

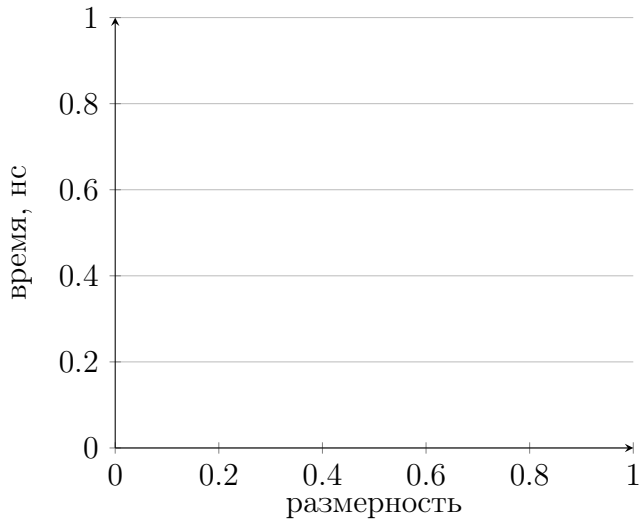

Рис. 4.7: Зависимость времени работы от размерности матриц (при 16 потоках)

Заключение

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

1. были изучены последовательный и два параллельных реализаций алгоритмов Копперсмита-Винограда;
2. были реализованы последовательный и два параллельных алгоритма Копперсмита-Винограда;
3. был проведён сравнительный анализ алгоритмов на основе экспериментальных данных;
4. был подготовлен отчёт по лабораторной работе;
5. были получены практические навыки реализации алгоритмов на ЯП Nim.

Исследования показали, что первая схема параллельной реализации показывает себя в среднем на $\approx 445\%$ хуже реализации с неразделённым вычислением элементов итоговой матрицы. Связано это с повторным выделением потоков для решения дополнительной задачи. Также стало известно, что в случае рассмотрения разницы между последовательной реализацией и параллельной с неразделённым вычислением элементов итоговой матрицы, первая покажет себя хуже в среднем на $\approx 322\%$

Литература

[1] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.

[2] Погорелов Дмитрий Александрович Таразанов Артемий Михайлович Волкова Лилия Леонидовна. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

[3] Nim documentation [Электронный ресурс]. Режим доступа: `https://nim-lang.org/documentation.html` (дата обращения 09.10.2020).