



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 дисциплине: Анализ Алгоритмов

Тема: Конвейерные вычисления

Студент Чаушев А.К..

Группа ИУ7-56Б

Оценка (баллы) \_\_\_\_\_

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

# Введение

Имеется большое количество важнейших задач, решение которых требует использования огромных вычислительных мощностей, зачастую недоступных для современных вычислительных систем. Что привело к созданию параллельных вычислительных систем, т.е. систем, в которых предусмотрена одновременная реализация ряда вычислительных процессов, связанных с решением одной задачи.

Цель данной работы является изучение алгоритма работы конвейера с использованием методов распараллеливания процессов.

В данной работе стоит задача реализации алгоритма Винограда для умножения матриц, сравнение последовательной и конвейерной реализаций.

# 1 Аналитическая часть

В данном разделе приведено описание алгоритма конвейера.

## 1.1 Постановка задачи

Конвейеризация – это техника, в результате которой задача или команда разбивается на некоторое число подзадач, которые выполняются последовательно. Каждая подкоманда выполняется на своем логическом устройстве. Все логические устройства (ступени) соединяются последовательно таким образом, что выход  $i$ -ой ступени связан с входом  $(i + 1)$ -ой ступени, все ступени работают одновременно. Множество ступеней называется конвейером. Выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду [1].

В конвейере различают  $r$  последовательных этапов, так что когда  $i$ -я операция проходит  $s$ -й этап, то  $(i + k)$ -я операция проходит  $(s - k)$ -й этап.

## 1.2 Выводы

Таким образом, выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду. Однако работу конвейера тормозят зависимости по данным и конфликты по ресурсам.

## 2 Конструкторская часть

Рассмотрим алгоритм конвейерных вычислений для алгоритма Винограда.

### 2.1 Схемы алгоритмов

На рисунке 1 изображена схема алгоритма Винограда.

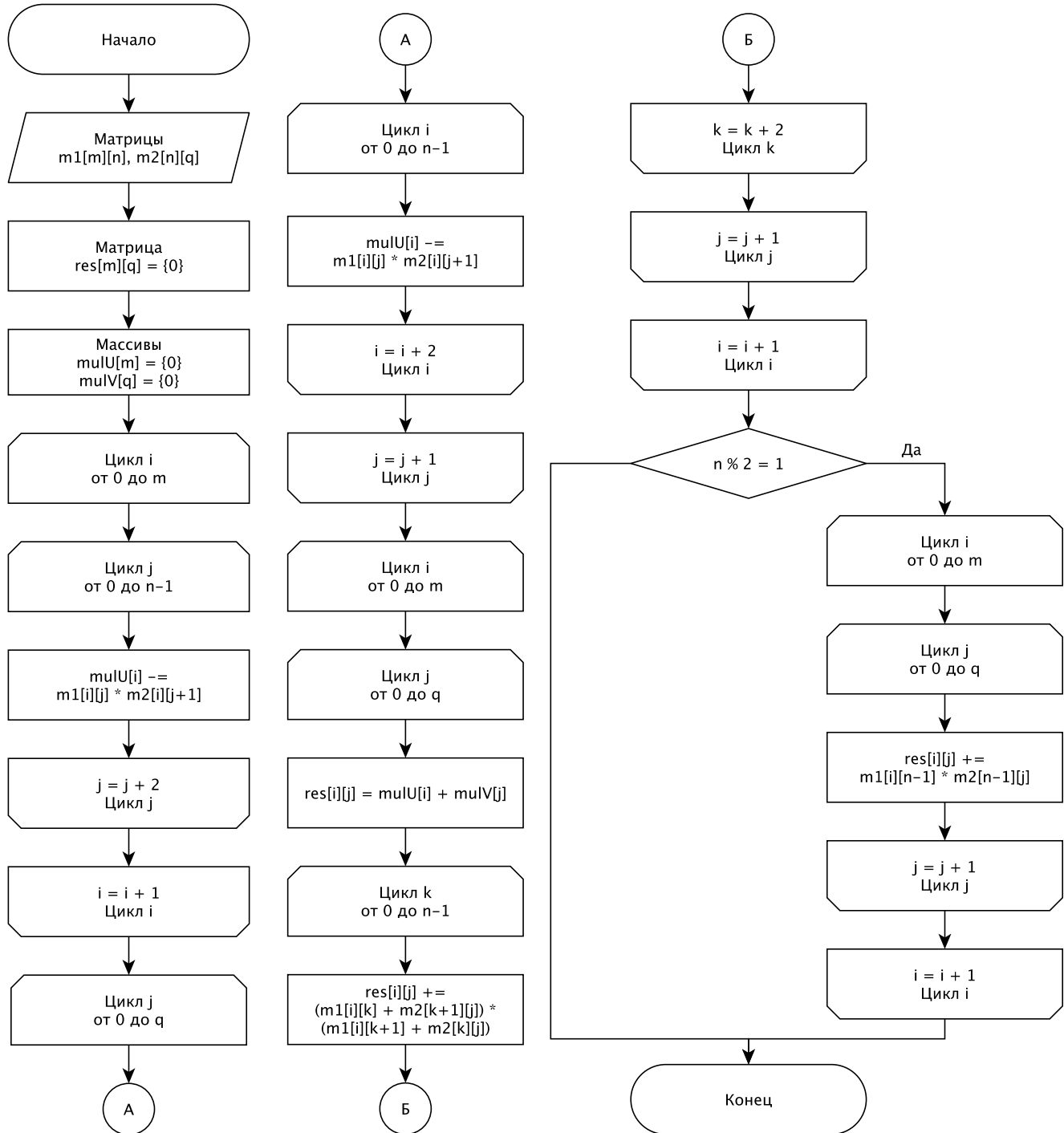


Рис. 1 – Схема алгоритма Винограда

На рисунке 2 представлена схема алгоритма, в котором действия разделены на конвейеры, выполняющиеся в отдельных потоках.

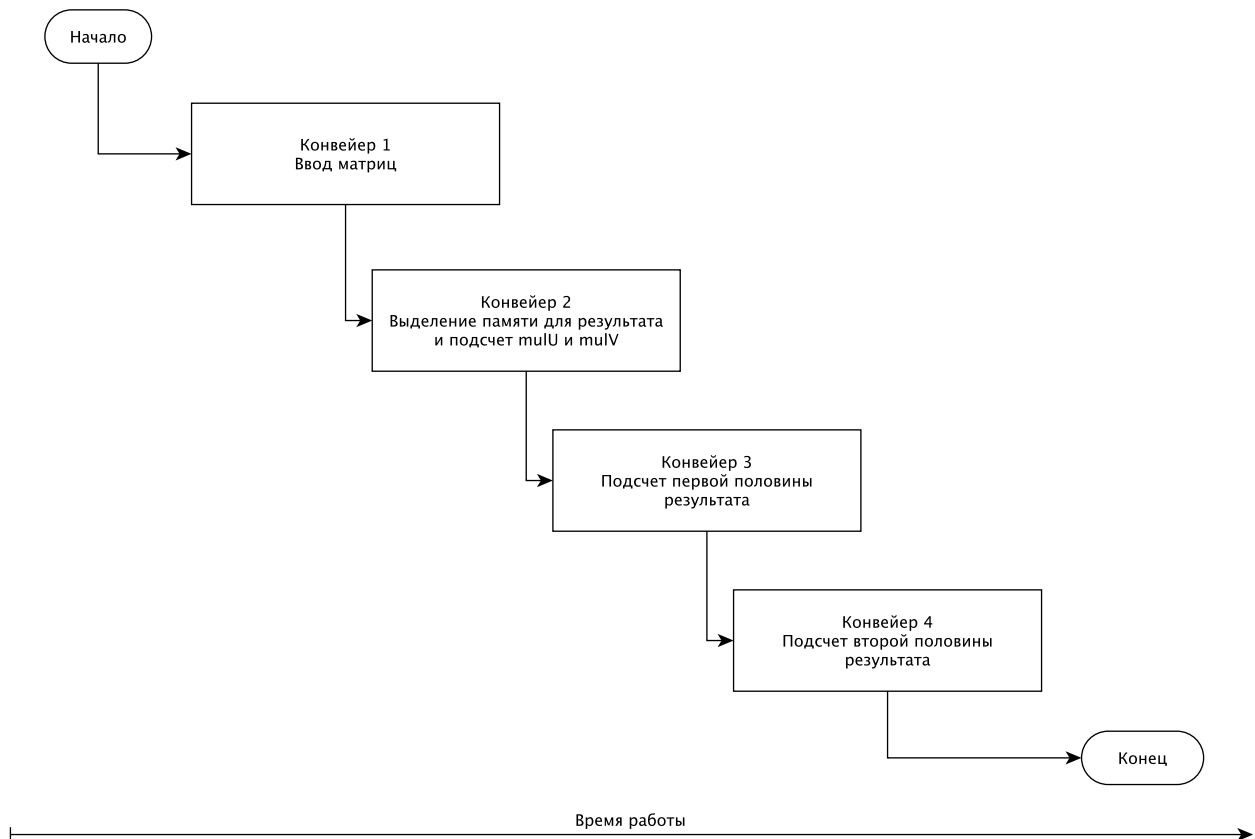


Рис. 2 – Схема конвейера

## 2.2 Выводы

Описанный принцип построения процессора действительно напоминает конвейер сборочного завода, на котором изделие последовательно проходит ряд рабочих мест. На каждом из этих мест над изделием производится новая операция. Эффект ускорения достигается за счет одновременного выполнения частей алгоритма.

## 3 Технологическая часть

Необходимо изучить и реализовать конвейерную разработку для умножения матриц.

### 3.1 Требования к программному обеспечению

Программное обеспечение должно обеспечивать замер процессорного времени выполнения каждого алгоритма. Проводятся замеры для случайно генерируемых квадратных матриц размерности до 1000.

### 3.2 Средства реализации

В качестве языка программирования был выбран **Kotlin**. Данный язык имеет полную совместимость с Java. Как и Java, C и C++, Kotlin[2] — это статически типизированный язык. Он поддерживает как объектно-ориентированное, так и процедурное программирование. Программа, написанная на **Kotlin**, будет доступна на всех платформах.

### 3.3 Методы замера времени в программе

#### 3.3.1 Время

Время замерялось с помощью функции `measureTimeMillis`, которая измеряет процессорное время в миллисекундах.

Листинг 1 – Функция замера времени.

```
1 inline fun measureTimeMillis(block: () -> Unit): Long
```

Для распараллеливания вычислений была использована функция `thread` [3].

Листинг 2 – Функция создания потока.

```
1 fun thread(  
2     start: Boolean = true,  
3     isDaemon: Boolean = false,  
4     contextClassLoader: ClassLoader? = null,  
5     name: String? = null,  
6     priority: Int = -1,  
7     block: () -> Unit  
8 ): Thread
```

Тестирование проводится на процессоре с количеством логических потоков равным 4.

### 3.4 Листинг кода

На листингах 1, 2 представлен код для ввода двух матриц.

Листинг 1 – Ввод первой матрицы

```
1 fun inputFirstMatrix(scanner: Scanner) {
2     n1 = scanner.nextInt()
3     m1 = scanner.nextInt()
4
5
6     firstMatrix = Array(n1) {IntArray(m1)}
7
8     for (i in 0 until n1 - 1) {
9         for (j in 0 until m1 - 1) {
10             firstMatrix[i][j] = scanner.nextInt()
11         }
12     }
13 }
```

Листинг 2 – Ввод второй матрицы

```
1 fun inputSecondMatrix(scanner: Scanner) {
2     n2 = scanner.nextInt()
3     m2 = scanner.nextInt()
4
5     mulU = IntArray(n1 * n2)
6     mulV = IntArray(m1 * m2)
7
8     for (i in 0 until n2 - 1) {
9         for (j in 0 until m2 - 1) {
10             secondMatrix[i][j] = scanner.nextInt()
11         }
12     }
13 }
```

На листингах 3, 4 представлен код для вычисления массивов mulU и mulV .

Листинг 3 – Подсчет mulU

```
1 fun calcMulU() {
2     for (i in 0 until n1 - 1) {
3         for (j in 0 until n2 step 2)
4             mulU[i] -= firstMatrix[i][j] * firstMatrix[i][j + 1]
5         }
6     }
```

Листинг 4 – Подсчет mulV

```
1 fun calcMulV() {
2     for (i in 0 until m2 - 1) {
3         for (j in 0 until n2 step 2)
4             mulV[j] -= secondMatrix[i][j] * secondMatrix[i + 1][j]
5         }
6     }
```

На листингах 5, 6 вычисляется результат.

Листинг 5 – Подсчет первой половины результата

```
1 void Multiplication::calculate1()
2 {
3     for (i in 0 until (n1 shr 1) + 1) {
4         for (j in 0 until m2) {
5             result[i][j] = mulU[i] + mulV[j]
6             var k = 0
7             while (k < n2 - 1) {
8                 result[i][j] += (firstMatrix[i][k] + secondMatrix[k + 1][j]) *
9                     (firstMatrix[i][k + 1] + secondMatrix[k][j])
10                k += 2
11            }
12        }
13    }
14 }
```

Листинг 6 – Подсчет второй половины результата

```
1 void Multiplication::calculate2()
2 {
3     for (i in (n1 shr 1) + 1 until n1) {
4         for (j in 0 until m2) {
5             result[i][j] = mulU[i] + mulV[j]
6             var k = 0
7             while (k < n2 - 1) {
8                 result[i][j] += (firstMatrix[i][k] + secondMatrix[k + 1][j]) *
9                     (firstMatrix[i][k + 1] + secondMatrix[k][j])
10                k += 2
11            }
12        }
13    }
14 }
```

На листингах 7, 8, 9 и 10 проводятся конвейерные вычисления умножения матриц.

Листинг 7 – Первый конвейер

```
1 fun functionPipeline1() {
2     while (true) {
3         var scanner = Scanner(inputStream)
4
5         if (!scanner.hasNext()) {
6             stopQueue1 = true
7             break
8         }
9         var time = measureTimeMillis {
10             var mult = Mult()
11             mult.inputFirstMatrix(scanner)
12             mult.inputSecondMatrix(scanner)
13
14             synchronized(mutex2) {
15                 queue2.push(mult)
16             }
17         }
18         timeQueue1 += time
19     }
20 }
```



Листинг 8 – Второй конвейер

```

1 fun functionPipeline2() {
2     while (true) {
3         if (stopQueue1 && queue2.isEmpty()) {
4             stopQueue2 = true
5             break
6         }
7
8         if (queue2.isEmpty()) continue
9
10        var mult: Mult
11        synchronized(mutex2) {
12            mult = queue2.first
13            queue2.pop()
14        }
15        mult.createResult()
16        mult.calcMulU()
17        mult.calcMulV()
18
19        synchronized(mutex3) {
20            queue3.push(mult)
21        }
22    }
23    timeQueue2 += time
24 }
25 }
```

Листинг 9 – Третий конвейер

```

1 fun functionPipeline3() {
2     while (true) {
3         if (stopQueue2 &&
4             queue3.isEmpty()) {
5             stopQueue3 = true;
6             break;
7         }
8
9         if (queue3.isEmpty()) continue;
10        var time = measureTimeMillis {
11            var mult: Mult
12            synchronized(mutex3) {
13                mult = queue3.first
14                queue3.pop()
15            }
16
17            mult.calculate1();
18
19            synchronized(mutex4) {
20                queue4.push(mult)
21            }
22        }
23        timeQueue3 += time
24    }
25 }
```

```

1 fun functionPipeline4() {
2     while (true) {
3         if (stopQueue3 && queue4.isEmpty()) {
4             break
5         }
6
7         if (queue4.isEmpty()) continue
8
9         var time = measureTimeMillis {
10
11             var mult: Mult
12             synchronized(mutex4) {
13                 mult = queue4.first
14                 queue4.pop()
15             }
16
17             mult.calculate2()
18             mult.check()
19             queueResult.push(mult)
20         }
21
22         timeQueue4 += time
23     }
24 }

```

### 3.5 Тестирование

Для тестирования программы были заготовлены следующие тесты в таблице 1.

Таблица 1 – Тесты для алгоритмов

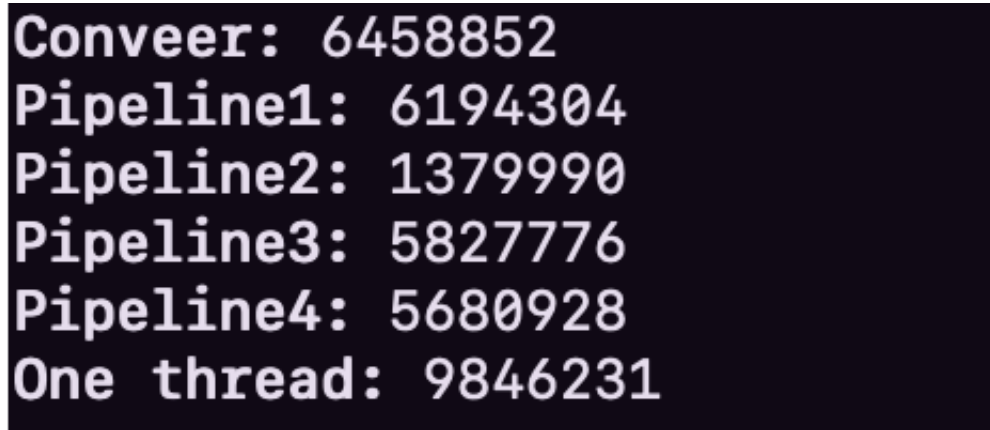
Первая матрица	Вторая матрица	Ожидаемый результат
1 2 3 4	1 2 3 4	7 10 15 22
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	30 36 42 66 81 96 102 126 150
1 2 3 4 5 6	1 2 3	14 32

## 4 Экспериментальная часть

Проведем тестирование и сравним алгоритмы по времени работы.

### 4.1 Примеры работ

На рисунке 3 изображены примеры работ.



```
Conveer: 6458852
Pipeline1: 6194304
Pipeline2: 1379990
Pipeline3: 5827776
Pipeline4: 5680928
One thread: 9846231
```

Рис. 3 – Успешное выполнение

### 4.2 Результаты тестирования

Для тестирования были использованы тесты в таблице 1. Результаты продемонстрированы в таблице 2.

Таблица 2 – Результаты тестирования

Первая матрица	Вторая матрица	Результат
1 2 3 4	1 2 3 4	7 10 15 22
1 2 3 4 5 6 7 8 9	1 2 3 4 5 6 7 8 9	30 36 42 66 81 96 102 126 150
1 2 3 4 5 6	1 2 3	14 32

Все тесты пройдены успешно.

### 4.3 Замеры времени

На рисунке 4 видно сравнение времени работы одного потока против конвейра на примере количества матриц от 100 до 1000.

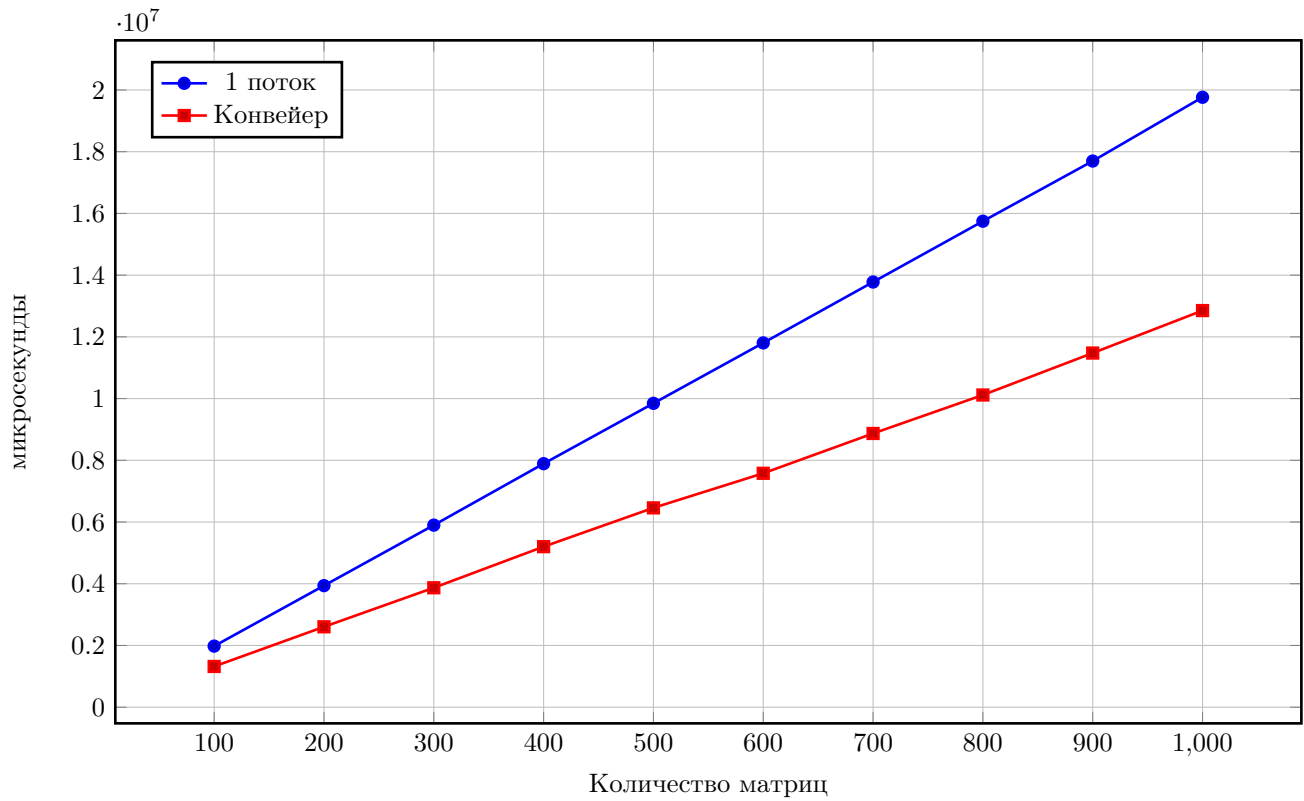


Рис. 4 – Сравнение конвейра с 1 потоком

### 4.4 Выводы

Конвейерная реализация выигрывает у обычной примерно в два раза (рисунок 4).

## Заключение

Таким образом, выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду.

В результате проведенных исследований был реализован алгоритм Винограда, распараллеленный по конвейрам. В результате такой алгоритм оказался быстрее в два раза, чем обычный. Ускорение произошло благодаря одновременного выполнения частей алгоритма.

При выполнении лабораторной работы цель была достигнута и выполнены все задачи.

## Список литературы

- [1] Конвеерные вычисления: [ЭЛ. РЕСУРС] Режим доступа: <http://www.myshared.ru/slide/674082>. (дата обращения: 14.12.2020).
- [2] JetBrains Kotlin [ЭЛ. РЕСУРС] Режим доступа: <https://kotlinlang.org/>. (дата обращения: 21.10.2020).
- [3] Документация по Thread [ЭЛ. РЕСУРС] Режим доступа: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/> (дата обращения: 21.10.2020).